DL_MESO Technical Manual

Release 2.7

M. A. Seaton and W. Smith

Mar 21, 2022

CONTENTS:

CHAPTER

ONE

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, UKRI STFC Daresbury Laboratory
- Rob English, North East Wales Institute
- Andrea Ferrante, Novidec
- Ian Halliday, Sheffield Hallam University
- John Harding, University of Sheffield
- Sebastian Reich, Imperial College
- Bill Smith, UKRI 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, John Purton, Ilian Todorov, David Bray, Annalaura Del Regno, Olga Lobanova, Antoine Schlijper, Andrea Ferrante, Massimo Noro, Ian Stott, Neil George, John Hone, Peter Dowding, Kai Luo and the UK Consortium for Mesoscale Engineering Sciences (UKCOMES), Luke Mason, Sergi Siso and Terry Hewitt.

Particular thanks go to Rongshan Qin at The Open University as the original author of the Lattice Boltzmann Equation source code (DL_MESO_LBE) and the DL_MESO graphical user interface, Jianping Meng at UKRI STFC Daresbury Laboratory for his contributions to DL_MESO_LBE, Richard Anderson at UKRI STFC Daresbury Laboratory, Ard van Bergen at Novidec and Bill Swope at IBM for their contributions to the Dissipative Particle Dynamics source code (DL_MESO_DPD), and Michael Johnston and Leopold Grinberg at IBM for their extensive optimisation work on both codes.

CHAPTER

TWO

DL_MESO GENERAL INFORMATION

2.1 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) and the High-End Computing UK Consortium on Mesoscale Engineering Sciences (UKCOMES). The package is the property of the UKRI 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, UKRI 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)

2.2 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 and UKCOMES strongly influence developments, but other input would be welcome nevertheless.

2.2.1 Lattice Boltzmann Equation

DL_MESO_LBE can simulate fluid (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 [154]
- Collisions: Bhatnagar-Gross-Krook (BGK) single-relaxation-time [10], Two-Relaxation-Time (TRT) [42], Multiple-Relaxation-Time (MRT) [73][159][134] or cascaded LBE (CLBE) [35][33]
- Forcing methods: Martys/Chen [91], Equal Difference Method (EDM) [69], Guo [49], He [54]
- Rheological models: Newtonian, power law, Bingham plastic [11], Herschel-Bulkley plastic [55], Casson [19], Carreau-Yasuda [17][152]
- Boundary conditions: Periodic, bounce-back (including stationary objects), constant pressure/velocity at planar surfaces [158][63][74][5]
- Mesoscale interactions: Shan-Chen pseudopotential method [118][119], Lishchuk continuum-based method [81], Swift free-energy method [136][135]

- Initial conditions can either be determined by DL_MESO_LBE or specified by the user

2.2.2 Dissipative Particle Dynamics

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

- Choice of integrators/thermostats: standard Velocity Verlet, DPD Velocity Verlet [40], Lowe-Andersen [83], Peters [100] and Stoyanov-Groot [131]
- Constant volume (NVT) or constant pressure (NPT) simulations with Berendsen [8] or Langevin [64] 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 [46], density-dependent (many-body) DPD [97][140], Lennard-Jones [66], Weeks-Chandler-Andersen [147]
- Electrostatic potentials between charged beads using modified Ewald summations [44][146], optionally using Smooth Particle Mesh Ewald [31]
- Boundaries: Periodic, hard reflecting walls with optional short-range repulsions (DPD [105] or Weeks-Chandler-Andersen), frozen particle walls, Lees-Edwards periodic shearing boundaries [76]
- Initial conditions can either be determined by DL_MESO_DPD or specified by the user

2.3 Requirements

2.3.1 Software requirements

- Standard C++ Compiler for LBE source code, DL_MESO_LBE
- Standard Fortran (2003 or later) Compiler for DPD source code, DL_MESO_DPD
- GNU Make (included in standard Unix/Linux distributions; can be installed for Windows)
- Message Passing Interface version 2 (MPI-2) or higher (if parallel execution required)
- JAVA 2 Version 1.4 or higher (if GUI is to be used)

Versions of the codes exist that use Open Multi-Processing (OpenMP) to divide up calculations on each processor core among threads, which require compilers that can link in OpenMP libraries: the majority of recent standard C++ and Fortran compilers are able to do this. For Smooth Particle Mesh Ewald calculations in the DPD code, either the FFTW 3.x or IBM ESSL Fast Fourier Transform (FFT) libraries may be used in place of the internal FFT solver.

2.3.2 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/7, IBM p690+ HPCx, PowerPC 450 Blue Gene/P, PowerPC A2 Blue Gene/Q, Cray XT4/XT6 HECToR, Cray XC30 ARCHER and Intel Xeon E5-2670 (Sandy Bridge) machines.

2.4 The DL_MESO Directory Structure

The supplied version of DL_MESO is a zip file **dl_meso_2.x**, where x is a generation number: this unpacks as a directory **dl_meso** 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'

2.5 Disclaimer

2.6 Copyright

2.7 Authors

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

2.8 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

THREE

INTRODUCTION TO DL_MESO TECHNICAL MANUAL

This chapter gives an overview of the DL_MESO Technical Manual, including its purpose and how it is structured.

3.1 Purpose

This Technical Manual is intended as a guide on how DL_MESO has been written and structured, and how its codes can be modified. It is intended to be used by code developers and advanced users (e.g. user-developers) who wish to augment DL_MESO_LBE or DL_MESO_DPD to include new functionalities or modify existing ones, primarily to carry out or disseminate new scientific research.

This manual is intended to be complementary to the DL_MESO User Manual. The User Manual is intended to help users get their Lattice Boltzmann Equation (LBE) and Dissipative Particle Dynamics (DPD) simulations running and to set out the features available in each code (including their theoretical background), while the Technical Manual is focussed more on how the codes implement the algorithms and features. Some details on algorithmic implementations are included in the User Manual primarily to enable users to get a simulation running, particularly on larger computing platforms, while this manual includes further details to enable user-developers to implement new features.

3.2 Structure

The manual is divided broadly into four sections:

- DL_MESO_LBE (LBE code written in C++)
- DL_MESO_DPD (DPD code written in Fortran)
- DL_MESO GUI (written in Java)
- DL_MESO Utilities

The DL_MESO_LBE and DL_MESO_DPD sections both include chapters providing:

- Programming background
- Detailed code description
- Input and output file formats
- Advice on how to develop the code
- Additional information for the code

The programming background includes the algorithm in use and the basic concepts for coding it up, the strategies used to parallelise calculations, what is stored in memory during a calculation (and how it is laid out), details on how key parts of the algorithm (including various functionalities) have been implemented, and how files are read into the code and simulation data are written.

The detailed code description for each code goes through the various modules, listing the functions, subroutines and any variables contained inside them, which are later described in more detail, including how functions and subroutines are called in the code and the input and/or output variables. These descriptions are based on automated documentation for DL_MESO_LBE and DL_MESO_DPD created using Doxygen, which has been converted into reStructuredText (reST) for Sphinx to generate the Technical Manual.

The chapters on input and output file formats are augmented forms of the equivalent chapters in the DL_MESO User Manual, which go into further detail about the file formats required for user-developers to create utilities or scripts to read and/or manipulate them. While some details about the available options for input files are included in these chapters, the user-developer is directed to the User Manual for complete lists of e.g. keywords.

The advice chapters provide some contextual information on how the codes can be modified to include new functionalities or modify currently-available ones. These include suggestions on where changes need to be made or where new code can be added, as well as which pre-existing subroutines and functions can be used as ‘templates’ for writing new code.

The additional information chapter includes information that may be relevant for the user-developer for implementing their changes, which are modified versions of appendices in the DL_MESO User Manual. For DL_MESO_LBE, this includes the lattice schemes used in the codes, including how each lattice link is identified and consequently how e.g. Multiple-Relaxation-Time (MRT) collision operator matrices are defined. For DL_MESO_DPD, a list of available error and warning messages is provided.

The DL_MESO GUI section consists of a single chapter containing a detailed code description for the Graphical User Interface (GUI), primarily to help user-developers make minor changes to the GUI to incorporate their changes made in DL_MESO_LBE or DL_MESO_DPD. The DL_MESO Utilities chapter consists of a single chapter listing the available utilities for preparing DL_MESO_LBE and DL_MESO_DPD simulations and processing their results: while these are not as rigorously described as the main codes and GUI, sufficient information about the utilities (and file formats) has been provided to enable user-developers to create their own.

CHAPTER

FOUR

DL_MESO_LBE PROGRAMMING BACKGROUND

4.1 Basic concepts

DL_MESO_LBE solves the Lattice Boltzmann Equation:

$$f_i\left(\vec{x}+\hat{e}_i\Delta t, t+\Delta t\right) - f_i\left(\vec{x}, t\right) = C_i + F_i\Delta t \tag{4.1}$$

where $f_i\left(\vec{x}, t\right)$ is a distribution function describing the probability of finding particles at time t and position $\vec{x}$ with a particular momentum moving them along a lattice link vector $\hat{e}_i$ to neighbouring grid points in a time period of Δt. The available lattice link vectors, including the number of Cartesian dimensions they cover, define several of the parameters used in LBE calculations and form a *lattice scheme* that is often described by DnQm, where n is the number of dimensions (normally 2 or 3) and m is the total number of available link vectors, including any rest vector with zero distance[1].

The macroscopic density of the fluid at a given position and time is equal to the sum of the associated distribution functions over all lattice links:

$$\rho\left(\vec{x}, t\right) = \sum_i f_i,$$

while the fluid momentum is given as the first moment of distribution functions (summed product of distribution functions and link vectors):

$$\vec{p}\left(\vec{x}, t\right) = \rho\left(\vec{x}, t\right)\vec{u}\left(\vec{x}, t\right) = \sum_i f_i \hat{e}_i.$$

The governing equation (4.1) can be divided into separate processes for *collisions*:

$$f_i\left(\vec{x}, t^+\right) = f_i\left(\vec{x}, t\right) + C_i + F_i\Delta t \tag{4.2}$$

where a collision operator C_i and forcing term to apply forces to the fluid F_i can be applied, resulting in post-collisional distribution functions at time t^+, and *propagation*:

$$f_i\left(\vec{x}+\hat{e}_i\Delta t, t+\Delta t\right) = f_i\left(\vec{x}, t^+\right) \tag{4.3}$$

which moves collided distribution functions along lattice links to neighbouring grid points. It is worth noting that (excluding calculations of forces) the collision process operates entirely locally to each grid point, while propagation of distribution functions only extends to nearest neighbouring grid points.

The collision process can take many different forms, although the simplest is the Bhatnagar-Gross-Krook (BGK) approximation [10]:

$$C_i = -\frac{\Delta t}{\tau_f}\left[f_i\left(\vec{x}, t\right) - f_i^{eq}(\rho(\vec{x}, t), \vec{u}(\vec{x}, t))\right], \tag{4.4}$$

where τ_f is a single relaxation time that can be related to the kinematic viscosity of the fluid, and $f_i^{eq}\left(\rho, \vec{u}\right)$ is the local equilibrium distribution function for the given fluid density and velocity. More complex collision

[1] The lattice schemes currently available in DL_MESO_LBE are D2Q9, D3Q15, D3Q19 and D3Q27. Most but not all features are available for each lattice scheme: the main exceptions are no Swift free-energy interactions available for D3Q27 and no cascaded LBE collisions are available for D3Q15.

operators are available with additional relaxation times or frequencies to improve the numerical stability of LBE calculations and/or to control additional hydrodynamic properties, while it is also possible to apply non-constant viscosity rheological models with local values of τ_f for each grid point calculated from shear rates (velocity gradients).

The forcing term F_i can be used to apply forces to the fluid. As well as constant body forces (e.g. gravity), these can include interaction forces calculated at each grid point related to gradients of values such as fluid density: the gradients can be calculated from values in surrounding grid points by using stencils.

Propagation is achieved by moving the post-collisional distribution functions along the lattice links to neighbouring grid points. No further calculation is required for this step, although care is needed to ensure the distribution functions reach their destination grid points without being overwritten and remain with the same lattice links.

The LBE method can be extended to multiple fluid systems by defining distribution functions for individual fluids existing on the same grid: interaction forces between fluids can be calculated and (depending on the interaction scheme) the distribution functions for individual fluids can be collided separately and propagated. A similar approach can be taken to include diffusion of solutes in a fluid and/or heat transfer: the additional distribution functions represent solute concentration or temperature respectively, using the fluid velocity to calculate local equilibrium distribution functions and the relaxation time (τ_c, τ_t) to control the mass or thermal diffusivity.

Boundary conditions are applied by devising distribution functions going back into the bulk system for lattice points representing a boundary. The simplest class of boundary condition include bounce back - reflection of distribution functions - that can be applied to any lattice point to provide a no-slip (zero velocity) condition. Outflow boundary conditions use copies of distribution functions in nearby grid poiints, while other schemes exist to apply constant values of fluid velocity, densities, solute concentrations and temperature by calculating ‘missing’ distribution functions based on the other available values. Each lattice point can be assigned a number to specify the type of boundary condition it includes and the direction in which it is applied.

4.2 Parallelisation strategies

DL_MESO_LBE consists of two methods to divide computational work among the available processor cores and threads:

- Equipartition domain decomposition
- Multithreading of main calculation loops

Domain decomposition involves the division of computational work among the available processor cores, with each core carrying out a section of the calculation with as little input from other cores as possible. This strategy is particularly effective for LBE calculations, since the application of collisions at each grid point is generally the most computationally intensive part and does not require input from other grid points. As such, each processor core is assigned its own section of the entire lattice - described as a *subdomain* - and applies collisions solely to its own grid points. To ensure each processor core carries out similar amounts of computational work, the lattice is divided as equally as possible among the available cores - *fArrangeProcessors()* - with each core’s subdomain selected based on its number: *fDefineDomain()*. This form of *equipartition* domain decomposition works well, particularly since the number of grid points held by each processor core can remain constant throughout the simulation.

While propagation and interaction force calculations do require information originating from surrounding grid points, including those found on neighbouring processor cores, these can be achieved by defining a *boundary halo* of additional grid points around the subdomain. The information for the boundary halo - distribution functions, interaction forces etc. - can be shared between pairs of processor cores using MPI (Message Passing Interface) communications. In the case of DL_MESO_LBE, these are normally carried out after defining MPI derived data types to specify vectors of data being sent and received by each core - *fDefineMessage()*[2] - and determining which cores are nearest neighbours in each direction: *fDefineNeighbours()*. The MPI communications themselves are carried out using a series of non-blocked send/receive calls applied sequentially in each Cartesian direction ($\pm x$, $\pm y$ and, if needed, $\pm z$) to ensure edges and corners are also dealt with correctly. As well as dealing with internal

[2] There is a compile-time option available in DL_MESO_LBE, `DPackbuf`, to alternatively pack the data being communicated into an array, communicate (send/receive) the data and then unpack the data into the appropriate places in memory. This approach can exploit OpenMP multithreading to speed up packing and unpacking the arrays. There is also another compile-time option, `DMPIold`, that makes use of older names for MPI routines (subsequently changed from MPI-2 onwards) to create derived data types.

boundaries between subdomains, these communications to boundary halos also enable periodic boundaries across opposite sides of the entire simulation grid.

For each simulation time step, the distribution functions $f_i(\vec{x}, t)$ - *fNonBlockCommunication()* - need to be communicated to boundary halos at the start to enable properties for interaction forces (e.g. fluid densities) to be calculated. Interaction models that require force calculations based on gradients of phase indices or densities will require these properties to be communicated - *fIndexNonBlockCommunication()* - before the interaction forces themselves are communicated: *fForceNonBlockCommunication()*. These communications and local calculations of body forces provide all the information needed to apply collisions to all grid points, including those in the boundary halo, and no further communications will then be needed for the propagation stage. The boundary condition values for grid points in the boundary halo can also be communicated - *fBoundNonBlockCommunication()* - although this is generally only required once at the start while setting up the simulation.

Compared to the serial version, the parallel version of DL_MESO_LBE includes an additional module - *lbpMPI.cpp* - which includes functions and subroutines to avoid using MPI calls directly in the main body of the code. There are separate main source code files for the two versions of DL_MESO_LBE - *plbe.cpp* for parallel running, *slbe.cpp* for serial running - as well as parallel and serial versions of modules with the main calculation loops - *lbpRUNPAR.cpp and lbpRUNSER.cpp* - and subroutines to aggregate data for file writing: *lbpIOAGGPAR.cpp and lbpIOAGGSER.cpp*.

When running DL_MESO_LBE in serial (i.e. on a single processor core), the boundary halo is not essential as the distribution functions and other data values for all lattice points are already available. By default, DL_MESO_LBE will not include a boundary halo for serial running and will instead use modulo functions to find neighbouring grid points that cross periodic boundaries. A customisable version of the serial DL_MESO_LBE code - *slbecombine.cpp* - enables the use of the boundary halo when running on a single processor core with additional subroutines - *fsPeriodic()*, *fsIndexPeriodic()*, *fsForcePeriodic()* and *fsBoundPeriodic()* - to copy the relevant values into the additional grid points.

The main sections for each timestep - collisions, propagation and calculation of interaction forces - can additionally be sped up by dividing loop iterations among available threads. This has been enabled in DL_MESO_LBE by adding OpenMP directives to the main calculation loops, instructing the available threads to divide up the lattice points among themselves. A single loop is used wherever possible (particularly for collisions and at least part of propagations) over all available grid points to make division between threads as simple as possible. In the cases where nested loops are required (e.g. to go through all grid points apart from those in the boundary halo), an OpenMP directive to collapse these loops for division among threads is used. This form of optimisation can be used with both the parallel and serial versions of DL_MESO_LBE: in the case of the parallel version, OpenMP multithreading can be used alongside domain decomposition with MPI to speed up the computational performance of each individual core. No alternative source code files are needed as the OpenMP directives do not require changes to how the fundamental calculations in LBE are performed and are ignored if the compile-time option (normally a compiler flag such as `-fopenmp`) is not invoked.

4.3 Data storage

DL_MESO_LBE uses one-dimensional arrays to store its main calculational properties for individual grid points:

- *lbf* - Distribution functions f_i
- *lbphi* - Boundary condition numbers
- *lbneigh* - Indicators of boundaries in neighbouring grid points
- *lbboundnorm* - Surface normals for solid-fluid interfacial forces
- *lbft* - Interaction values (pseudopotentials for Shan-Chen interactions, interfacial normals for Lishchuk interactions, density and concentration gradients for Swift interactions)
- *lbinterforce* - Interaction forces acting on fluids due to fluid/phase interfaces
- *lbheatforce* - Convective heat forces acting on fluids (calculated using Boussinesq approximation)
- *lbomega* - Fluid relaxation frequencies

Each array is either the size of the number of available grid points, `lbdm.touter`, or an integer multiple of this number. This total number of grid points for a processor's subdomain is the product of `lbdm.xouter` (N_x), `lbdm.youter` (N_y) and `lbdm.zouter` (N_z), which include any boundary halo points. The coordinate number of an individual grid point given as Cartesian coordinates (x, y, z) (where $0 \leq x < N_x$, $0 \leq y < N_y$ and $0 \leq z < N_z$) can be calculated[3] as:

$$l = (xN_y + y)\, N_z + z$$

with z always equal to zero and $N_z = 1$ for two-dimensional simulations. This means the z-coordinate changes most quickly with grid point location, followed by the y-coordinate and then the x-coordinate.

Multiplying l by the number of values per grid point for the given array provides the starting location for data at that point.

The distribution functions f_i in *lbf* are stored in blocks ordered by link vector i, with the distribution functions for each fluid, solute and temperature field stored in that order. If a phase field ϕ is in use[4], this value is placed at the end. For instance, if two fluids (numbered 0 and 1), a temperature field and a phase field are being modelled on a D2Q9 lattice (with $0 \leq i < 9$), the distribution functions at each grid point would be stored as follows:

$$f_0^0, f_0^1, h_0, f_1^0, f_1^1, h_1, f_2^0, f_2^1, h_2, f_3^0, f_3^1, h_3, f_4^0, f_4^1, h_4, f_5^0, f_5^1, h_5, f_6^0, f_6^1, h_6, f_7^0, f_7^1, h_7, f_8^0, f_8^1, h_8, \phi$$

The total number of distribution functions per grid point is given as *lbsitelength*, equal to the product of the number of lattice links per grid point `lbsy.nq` (N_q) and the sum of the numbers of fluids (`lbsy.nf`, N_f), solutes (`lbsy.nc`, N_c) and temperature fields (`lbsy.nt`, N_t), plus the number of phase fields (`lbsy.pf`, N_p). The above choice of ordering in memory has been made for better exploitation of cache when applying propagation, particularly when multiple fluids, solutes and temperature fields are in use, while a subsequent link's distribution function for a given fluid/solute/temperature field can be found by increasing the array index by $N_f + N_c + N_t$. Since macroscopic properties - e.g. velocity, fluid density - can be calculated directly from distribution functions immediately prior to collisions or when writing output files, no additional arrays are required to store values of these properties.

The boundary condition numbers in *lbphi* are stored in an integer array: only one integer is required per grid point, the values of which are given in the input file *lbin.spa*. Similarly, the indicators of boundary conditions in neighbouring grid points in *lbneigh* also only require one integer per grid point, although these are calculated in the subroutine *fNeighbourBoundary()* based on a search of values of *lbphi* and used to describe how derivatives of properties should be calculated for each lattice point.

The array with fluid relaxation frequencies *lbomega* includes blocks with values of $\omega_f = \frac{1}{\tau_f}$ for all fluids at each grid point. The convective heat force array *lbheatforce* includes the Boussinesq approximation (temperature-dependent) forces calculated for all fluids at each grid point, with the x-, y- and z-components of force for each fluid together in memory, e.g. for three fluids at a given grid point:

$$F_x^0, F_y^0, F_z^0, F_x^1, F_y^1, F_z^1, F_x^2, F_y^2, F_z^2$$

The size of the interaction force array *lbinterforce* depends upon the fluid interaction model in use. It can be identical to the size of $lbheatforce$ (with each grid point requiring three values per fluid) when Shan-Chen pseudopotential interactions are in use, as forces for individual fluids are calculated. If Lishchuk continuum-based interactions are selected, however, only a single force is needed per grid point, which is applied to all fluids simultaneously during collisions.

Similarly, the size of the array *lbft* will depend on the fluid interaction model, with each grid point storing:

- N_f values for Shan-Chen interactions to store pseudopotentials for each fluid;
- $\frac{N_f(N_f-1)}{2}$ values for Lishchuk interactions to store interfacial normals for each pair of fluids (not including each fluid with itself); or
- $4N_f$ values for Swift interactions to store first-order and second-order derivatives of densities and concentrations (three Cartesian coordinates for first-order, a single value for second-order).

[3] This formula is used by the function *fGetNodePosi()*, while the *fGetCoord()* subroutine can be used to obtain the inverse (Cartesian coordinates) by using integer division and modulo functions.

[4] No available interaction model in DL_MESO_LBE currently makes use of it: it has been provided as an option for further functionality expansion, but users currently should specify zero for the number of phase fields in the input file *lbin.sys*.

A single triple of Cartesian coordinates for surface normals are stored in the array $lbboundnorm$ for each grid point, which are used to calculate solid-fluid interactions for the Lishchuk continuum-based algorithm.

4.4 Collisions

Four general forms of particle collisions are available in DL_MESO_LBE:

- Bhatnagar-Gross-Krook (BGK) single relaxation time
- Two-Relaxation-Time (TRT)
- Multiple Relaxation Time (MRT) based on raw moments of distribution functions
- Cascaded Lattice Boltzmann Equation (CLBE) based on central moments of distribution functions

Each of these are available in modules - *lbpBGK.cpp*, *lbpTRT.cpp*, *lbpMRT.cpp* and *lbpCLBE.cpp* - which include the subroutines required to carry out collisions at all available grid points for each processor core. There are two main classes of subroutines in each module:

- Subroutines with loops over all grid points, e.g. *fCollisionBGK()*
- Subroutines to carry out collisions for all fluids, solutes or temperature fields at a single grid point, e.g. *fSiteFluidCollisionBGK()*

with the former calling the latter for each lattice site that contains fluid[5], sending in calculated values of (variable) fluid densities and velocities, as well as a pointer indicating the first distribution function value for the required lattice site. The site collision subroutines include calls to other subroutines to calculate values required for collisions, e.g. local equilibrium distribution functions f_i^{eq} in *fGetEquilibriumF()*, forcing terms in *fGetMomentForceGuo()* etc., apply (4.2) and overwrite the initial distribution functions $f_i\left(\vec{x},t\right)$ in the *lbf* array with post-collisional values, $f_i\left(\vec{x},t^{+}\right)$.

There are several variations for each collision scheme based on whether or not fluids are compressible, the type of fluid interactions in use and the chosen method of applying forces to fluids.

The site subroutines include variants for compressible and incompressible fluids, which use different local equilibrium distribution functions f_i^{eq} and (in the case of incompressible fluids) can use both constant and variable fluid densities. The subroutines with loops generally include two loops for compressible and incompressible fluids apart from Swift free-energy based interactions and CLBE collisions, both of which can only be applied to compressible fluids.

Shan-Chen interactions between fluids require a slightly different definition for the overall velocity of the fluids that takes their relaxation times into account: this is dealt with in the loop subroutines by using the *fGetSpeedShanChenAllMassSite()* and *fGetSpeedShanChenIncomAllMassSite()* subroutines to calculate fluid densities and velocity instead of the usual *fGetSpeedAllMassSite()* and *fGetSpeedIncomAllMassSite()* subroutines.

The collisions for Lishchuk interactions are applied to all fluids simultaneously by calculating 'achromatic' distribution functions, i.e.

$$f_i = \sum_a f_i^a,$$

applying the collision and then re-segregating the distribution functions along interfacial normals between pairs of fluids. The process of carrying out achromatic collisions and segregation are available in site collision subroutines, e.g. *fSiteFluidCollisionBGKLishchuk()*, that can take interfacial normals between pairs of fluids as an additional input sent from loop subroutines, e.g. *fCollisionBGKLishchuk()*. An alternative to applying interaction forces between fluids is also available in the form of forcing terms with interfacial normals, which are enacting in both site and loop subroutines such as *fSiteFluidCollisionBGKLishchukLocal()* and *fCollisionBGKLishchukLocal()* respectively.

[5] Boundary points for constant fluid velocity/density, solute concentration and temperature conditions must be included in collisions for the boundary schemes currently available in DL_MESO_LBE and are referred to as 'wet nodes' due to their inclusion of fluid. Blank and bounce back boundary sites do not need to be collided as their distribution functions are either set to zero or given post-collisional values from neighbouring grid points.

Collisions for Swift interactions can be applied either to one or two fluid systems: in the case of two fluid systems, the distribution functions for total fluid density and concentration (related to difference in densities between fluids) are both collided. For both one- and two-fluid systems, gradients of density (and concentration) need to be passed into the site collision subroutines, e.g. *fSiteFluidCollisionBGKSwiftOneFluid()*, to make use of more complex local equilibrium distribution functions that include these gradients for interfacial tension terms.

Each type of collision can incorporate forces acting on the fluids using one of four different forcing schemes:

- Martys-Chen: adjustment of velocity used in local equilibrium distribution functions f_i^{eq} for collisions to include forces
- Equal Difference Method (EDM): additional forcing term F_i based on differences in f_i^{eq} with and without the effect of forces on fluid velocity
- Guo method: adjusted fluid velocity in f_i^{eq} and additional forcing term F_i designed to give correct fluid behaviour
- He method: adjusted fluid velocity in f_i^{eq} and additional forcing term F_i determined from derivative of Maxwell-Boltzman local equilibrium distribution functions

and each of these is applied in both site and loop subroutines: the default forcing scheme is Martys-Chen.

4.5 Propagation

The propagation subroutines in DL_MESO_LBE enact (4.3), moving post-collisional distribution functions $f_i\left(\vec{x}, t^+\right)$ along the relevant lattice links $\hat{e}_i$ to neighbouring grid points in preparation for the next simulation timestep. The distribution functions in the array *lbf* for all fluids, solutes and temperature fields are moved: any phase field values remain in place as these are not specific to particular lattice links.

The simplest form of propagation is enacted in the subroutine *fPropagationTwoLattice()*. In this instance, the distribution functions in *lbf* for each fluid/solute/temperature field and link vector are copied into a temporary array equal in size to at least the total number of grid points in the subdomain[6], each value from position $\vec{x}$ being placed at the intended destination grid point $\vec{x} + \hat{e}_i \Delta t$. The values in the temporary array are then copied back to *lbf* in the appropriate places. While this method is clear, easy to understand and can be applied to the lattice points in any order, its use of two loops over all grid points per distribution function to move them and the copying of large numbers of values between arrays requires a lot of memory accesses and data storage. While the subroutine remains in DL_MESO_LBE as its original implementation of propagation, it is not used by default.

The other two propagation subroutines are based on a *swap algorithm* [93]. This approach carries out two sets of swaps of distribution functions: the first set of swaps is applied to distribution functions for each lattice point between conjugate lattice links, i.e. $f_i\left(\vec{x}\right)$ is swapped for $f_j\left(\vec{x}\right)$ when $\hat{e}_j = -\hat{e}_i$, while the second set of swaps is a reversal but with the neighbouring point, i.e. $f_j\left(\vec{x}\right)$ is swapped with $f_i\left(\vec{x} + \hat{e}_i\right)$. To make it easier to carry out swaps between conjugal lattice links, the links for each lattice scheme are arranged in memory so that $\hat{e}_j = -\hat{e}_i$ for $1 \leq i \leq \frac{N_q - 1}{2}$ (assuming link zero is a ‘rest’ link, $\hat{e}_0 = \vec{0}$), where $j = i + \frac{N_q - 1}{2}$.

The two sets of swaps can either be carried out in two separate loops as in *fPropagationSwap()* or combined in a single loop over all grid points (including the boundary halo) as in *fPropagationCombinedSwap()*. The combined simulataneous swaps can only work if a boundary halo of grid points is included and the lattice links are ordered so the first half ($1 \leq i \leq \frac{N_q - 1}{2}$) are all directed to lattice points that have previously been through at least the first swap stage. As such, the combined swap implementation is the default for parallel running that uses a boundary halo but no OpenMP multithreading (as this can break the required sequence of swaps), while the other implementations of DL_MESO_LBE make use of the two separate loops in *fPropagationSwap()*, which can have OpenMP applied to each loop in turn to speed them up by assigning each iteration to different threads.

[6] The subroutine reuses the *lbft* array, overwriting any pseudopotentials, interfacial normals or density/concentration gradients that are now no longer needed after collisions have taken place.

4.6 Fluid interactions

The module *lbpFORCE.cpp* includes all of the subroutines required for DL_MESO_LBE to calculate interaction forces and other related properties among the fluids in the simulation. These forces and other properties typically rely upon gradient calculations of properties related to fluid densities in some way:

- Pseudopotentials for Shan-Chen interactions
- Phase indices for Lishchuk interactions, used to calculate interfacial normals
- Fluid density (and concentration) for Swift interactions

While the gradient calculations for each interaction method differ, they each rely upon using values of the property in surrounding grid points. As such, DL_MESO_LBE identifies grid points on the very edge of the subdomain in the array *lbouter* (the number of these points given by *lboutersize*), which require modulo functions to find neighbouring lattice points across periodic boundaries. These points are located within layers of thickness `lbdm.owidx`, `lbdm.owidy` and `lbdm.owidz`, which are set to either the boundary halo size `lbdm.bwid` or 1 (whichever is larger) for all active Cartesian coordinates (i.e. `lbdm.owidz` is set to 0 for two-dimensional simulations), while the array *lbouter* stores both the one-dimensional coordinate number l and the Cartesian coordinates (i, j, k) for each point. As such, two sets of subroutines exist to calculate gradients at a given grid point: one type works on grid points near the edge of the subdomain and uses modulo functions to find neighbouring points across periodic boundaries, while the other works on grid points away from the subdomain edge and does not require modulo functions to find neighbours.

As with collisions, the module includes a number of subroutines with loops through the available grid points - excluding any in the boundary halo - to calculate forces and/or gradients. For instance, the forces for Shan-Chen interactions can be calculated using *fInteractionForceShanChen()* for parallel calculations - which excludes grid points in the boundary halo - or *fsInteractionForceShanChen()* for serial calculations that include grid points right at the edges of the lattice. In the case of the parallel version, the values of forces or gradients for lattice points in the boundary halo are obtained by applying a communication - *fForceNonBlockCommunication()* for forces, *fIndexNonBlockCommunication()* for gradient-based properties (interfacial normals for Lishchuk interactions, density/concentration gradients for Swift interactions) - to copy in values from neighbouring processor cores.

The subroutines in this module that do not require separated calculations for grid points near and far from the edge of the subdomain (and therefore do not subsequently require communications for running in parallel) include:

- *fInteractionForceZero()* - zeros all interaction and convective heat forces
- *fCalcPotential_ShanChen()* - calculates pseudopotential values for Shan-Chen interactions for all grid points (including the boundary halo)[7]
- *fCalcPhaseIndex_LishchukLocal()* - calculates interfacial normals for Lishchuk interactions in an entirely local manner for all grid points (using an approximation for the phase index gradient based on distribution functions for each grid point)
- *fCalcForce_Boussinesq()* - calculates temperature-dependent heat convection forces using the Boussinesq approximation at a given grid point
- *fConvectionForceBoussinesq()* - calculates temperature-dependent heat convection forces using the Boussinesq approximation at all grid points

All three interaction methods can include interactions between fluids and solid surfaces represented by lattice sites with specified boundary conditions. The array *lbneigh* is used for these interactions to identify the sites where they take place (at grid points adjacent to solid boundaries) and how the gradients need to be calculated (e.g. using one-sided gradient approximations to use fluid points further away from the boundary).

[7] The pseudopotentials in Shan-Chen interactions can be selected to give specific equation of states as functions of fluid density ρ and temperature T. Four of these equations of state - Redlich-Kwong, Soave-Redlich-Kwong, Peng-Robinson and Carnahan-Starling-Redlich-Kwong - include more complex dependences on temperature, which can either be fixed across the system (in the input file *lbin.sys*) or vary locally at each grid point using temperature field distribution functions: as such, these equations come in both constant and variable temperature forms. (This is also the case for calculations of bulk pressures p_b used for Swift interactions, although these are mainly calculated during the collision step.)

4.7 Rheological models

While the default hydrodynamic behaviour in DL_MESO_LBE is for constant kinematic viscosities ν for each fluid - represented as relaxation times τ_f - DL_MESO_LBE allows for alternative rheological models to be applied. The array *lbomega* is used to store relaxation frequencies (reciprocals of relaxation times) for all fluids at each grid point (including any boundary halo points), which can be updated with appropriate values based on specified rheological models for local shear rates (velocity gradients).

The shear rates at each grid point are calculated from the non-equilibrium part of the momentum flux tensor:

$$\dot{\gamma} = \sqrt{2 \sum_{\alpha,\beta} S_{\alpha\beta} S_{\alpha\beta}}$$

which is itself calculated using distribution functions and a collision matrix $\mathbf{\Lambda}$ that includes the relaxation frequency calculated during the previous timestep:

$$S_{\alpha\beta} = -\frac{3}{2\rho\Delta t} \sum_i e_{i,\alpha} e_{i,\beta} \sum_j \Lambda_{ij} \left(f_j - f_j^{eq} \right).$$

Subroutines in *lbpRHEOLOGY.cpp* are available to calculate the shear rate $\dot{\gamma}$ at individual grid points based on the collision scheme in use, e.g. *fGetShearRateBGK()* for BGK collisions. These are called by the subroutine *fGetSystemOmega()* before it calls *fGetRelaxationFrequency()* to calculate a new relaxation frequency for each grid point and fluid based on the rheological model and the shear rate. The relaxation frequencies in *lbomega* are included when writing the restart file *lbout.dump* to ensure the simulation can be resumed from where it left off.

4.8 Boundary conditions

The array *lbphi* is used by DL_MESO_LBE to identify grid points with boundary conditions: the values in this array are read in from the input file *lbin.spa*, which identifies each boundary condition by a number indicating its type, the direction in which it is applied and which properties are being specified (e.g. fluid velocity or densities, solute concentrations and/or temperature). A full list of available boundary codes is given in Chapter 6 of the DL_MESO User Manual, while an overview is given in Table 6.4 and Table 6.5.

The default value for the boundary code in *lbphi* is 0, which indicates no boundary condition is applied at that grid point, other than periodic boundaries if the point is at the outside of the simulation grid. If a boundary halo is in use, a boundary code of 10 is used at each of its grid points to indicate its location if no other boundary conditions need to be applied: this code ensures the grid point is involved in collisions but is omitted from sums of fluid densities and momenta used in diagnostic messages printed by DL_MESO_LBE.

A boundary code of 11 is used to indicate a blank site, e.g. one that is inside a solid object but not at a grid point adjacent to fluid. A code of 12 applies an *on-grid bounce-back* boundary condition, which provides a no-slip condition (zero fluid velocity) at a wall and is obtained by reversing distribution functions at the grid point once propagation has taken place. A *mid-grid bounce-back* condition is indicated by a boundary code of 13 and is implemented by copying and reflecting post-collisional distribution functions going into the boundary grid point. All three boundary types here can be applied at any arbitrary lattice point and their implementations do not depend upon direction (i.e. where the nearest fluid points happen to be).

All other boundary condition types available in DL_MESO_LBE include indications of the direction in which they are applied. Outflow conditions are implemented directly in DL_MESO_LBE, since either only four or six directions need to be considered as they are limited to those orthogonal to the outer surfaces or edges of the simulation grid. (First and second order accurate schemes are available, which make use of distribution functions from one or two fluid points beyond the boundary point respectively: the choice for these is made by the user in the *lbin.sys* input file.)

Other directional boundaries are treated in DL_MESO_LBE based on their types: planar surfaces, concave edges and concave corners. Only one direction for each of these needs to be implemented directly for each boundary condition scheme and each lattice scheme, as the inputs and outputs parameters for the subroutines (specifically distribution functions, velocities, forces and density/concentration gradients for Swift interactions) can be rearranged when called to give the correct results for other directions. The codes used in *lbphi* for these boundaries are represented as numbers between 100 and 899: the last two digits indicate direction, while the first digit specifies whether a constant fluid velocity or density condition is needed along with constant solute concentration/temperature or bounce back for these properties. The quantities being held constant at these boundaries are specified in *lbin.sys* for planar surfaces in three-dimensional simulations or concave edges in two-dimensional simulations. Corners and edges use one fluid property (velocity or densities/concentrations/temperature) from intersecting surface/edge boundaries and either take fluid densities or other properties from a nearby fluid point or assume zero fluid velocity.

Four boundary condition schemes are currently available in DL_MESO_LBE for constant fluid velocity and density conditions:

- Zou-He: *lbpBOUNDZouHe.cpp*
- Inamuro: *lbpBOUNDInamuro.cpp*
- Regularised: *lbpBOUNDRegular.cpp*
- Kinetic: *lbpBOUNDKinetic.cpp*

and each scheme has its own module with subroutines to implement the boundaries for each available lattice scheme. For instance, constant velocity conditions using the Zou-He scheme with a D2Q9 lattice are found in *fD2Q9VFZouHe()*, which calls *fD2Q9VCEZouHe()* for edges (as defined for the bottom edge) and *fD2Q9VCCZouHe()* for corners (defined for the bottom-left corner): the selection of lattice and boundary scheme for this kind of condition is made in the *fFixedSpeedFluid()* subroutine. Boundary conditions for constant solute concentrations and temperatures are currently limited to the Zou-He and Inamuro schemes.

4.9 Reading input files

Four input files can be read by DL_MESO_LBE at the start of a simulation: *lbin.sys* with simulation parameters, *lbin.spa* with boundary conditions, *lbin.init* with initial conditions and *lbout.dump* to restart a previous simulation. The first three of these are (human-readable) text files, while the *lbout.dump* file is written in binary (see below). All processor cores read each of these files in the current version of DL_MESO_LBE, although the assignment of boundary conditions, initial conditions and restart data in each core will depend on whether or not the specified grid points exist in its subdomain.

The *lbin.sys* consists of lines, each with a word at the start and a value (either a number or another word) separated by spaces and/or tab characters. The subroutines reading this file - *fDefineSystem()* and *fInputParameters()* - use the C++ function `getline` to extract each line in turn as a string, before applying the *fReadString()* function to find the word and value as individual strings. The word string is then compared with specific key words for simulation properties, and the value string is either compared with other key words or parsed using the function *fStringToNumber()* to give a number for assignment to a variable or array. The *fDefineSystem()* subroutine restricts itself to searching for fundamental properties for the simulation: spatial dimensions and discrete speeds (number of lattice links) to give the lattice scheme, numbers of fluids, solutes, temperature and phase fields, the total size of the lattice, the boundary halo size, whether or not the fluids are exactly incompressible, if the simulation is being restarted, the collision/forcing type, interaction type, and output file format. Many of these values are used to define array sizes, for storing the data used directly for simulations (e.g. distribution functions, interaction forces) and for parameters used to describe each fluid/solute/temperature field (e.g. relaxation times). These, along with initial and boundary conditions, are then read in by the *fInputParameters()* subroutine.

Both the *lbin.spa* and *lbin.init* files consist of lines of numbers. Each line of each file starts with three integers indicating integer Cartesian coordinates of a grid point, which are followed by the required data at that point (boundary code in *lbin.spa*, velocity, fluid densities, solute concentrations and temperatures in *lbin.init*). These numbers are read in directly from the filestream for the opened file and the value(s) assigned to the grid point if it exists in the current subdomain. (When running a two-dimensional simulation, only grid points where the z-coordinate is equal to 0 will be accepted.) The subroutines *fReadSpaceParameter()* and *fReadInitialState()* are used to read in the *lbin.spa* and *lbin.init* files respectively.

If a simulation restart is requested in the *lbin.sys* file and a *lbout.dump* can be found, the subroutine *fReadRestart()* will read the latter file. Some basic information about the simulation is read in first as a series of integers and compared with the information provided in *lbin.sys*: mismatches in lattice scheme, grid size, numbers of fluids, solutes, temperature and phase fields cannot be reconciled and will cause DL_MESO_LBE to halt with at least one error message stating the mismatch(es). The timestep at which the simulation previously stopped and the number of output files previously written are also among the basic information, as well as whether or not incompressible fluids were used: while a mismatch in this latter property is not fatal for restarting a DL_MESO_LBE simulation, the value in *lbout.dump* indicates that constant densities ρ_0 then immediately follow. After these data, the Cartesian coordinates identifying all grid points are then read in one set at a time: if the grid point is found inside the current processor core's subdomain, the distribution functions and fluid relaxation frequencies for that grid point are then read in and assigned to the arrays. All of the data in the *lbout.dump* is provided in binary format using the endianness of the computer that previously ran the simulation, although utilities also exist in DL_MESO to read and manipulate this data.

4.10 Writing output files

The main output files written by DL_MESO_LBE are in one of three different formats: XML-based structured grid VTK (*lbout*.vts*), structured grid Legacy VTK (*lbout*.vtk*) or Plot3D (*lbout*.q*). While the exact formats of the files differ, the details about how they are prepared and written are very similar.

By default, each processor core will write its own file for each trajectory frame (a snapshot of the simulation at a given timestep). This option can be overridden using the `output_combine` keywords in *lbin.sys*, which tell DL_MESO_LBE to combine data from processor cores along specific Cartesian axes onto a single processor core, which then produces a single output file for that group of processor cores. If all dimensions are requested for combination of output data, DL_MESO_LBE will switch on the option to use MPI-IO instead of full combination of data: multiple groups of processor cores are still created to combine data in all but one dimension (z-dimension for three-dimensional simulations, y-dimension for two-dimensional simulations)in slices, but each group's root core writes concurrently to a single output file at unique locations to ensure the data are written in the required order.

The *fCreateIOGroups()* subroutine sets up groups of processor cores that will share their data, determines the extent of the grid that each group will cover and creates MPI communicators to enable each group to gather their data on to a root core that will end up writing to its output file. A series of subroutines, e.g. *fGroupDensities()* for fluid densities, are available to gather together data required for file-writing and are called by subroutines specifying what is written to each file for the given format (e.g. *fOutputVTK()* for XML-based VTK files). In these subroutines, each processor core puts together the required values into an array (swapping bits if requested) before this data is gathered together into a larger array in the groups' root cores and then reordered with the x-coordinate as the fastest changing coordinate (followed by the y- and z-coordinates). Subroutines are then called, e.g. *fWriteVTKFloatBinaryData()*, to get the root processor core for each group to write the data to the file as a stream of numbers: if MPI-IO is in use, the MPI subroutine `MPI_File_write_at` is used to place the data in the appropriate place in the file. (If any text is required before or after the data, e.g. XML tags, this can also be added by the appropriate I/O groups.)

The *lbout.dump* restart file is prepared in a similar fashion to other output files. The subroutine *fGroupGather-RestartData()* - as called by the *fWriteRestart()* subroutine - brings together the grid point coordinates for each individual processor core or each I/O group of processor cores, as well as the distribution functions and relaxation frequencies for all fluids at each grid point. After writing the important simulation data required for a restart (lattice scheme, grid size etc.) and constant fluid densities if incompressible fluids are in use, the grid point co-ordinates are written by every core or I/O group (using MPI-IO to do this concurrently if running in parallel) to form a single block of integer numbers, before the distribution functions and relaxation frequencies are written as a single block of double precision floating-point numbers. Unlike other output files, only a single *lbout.dump* file is ever produced at a time regardless of the number of processor cores used to run the simulation.

CHAPTER

FIVE

DL_MESO_LBE CODE DESCRIPTION

This chapter lists and describes the subroutines, functions, variables, datatypes etc. in DL_MESO_LBE, based on output generated using Doxygen with annotations in the code.

5.1 lbe.hpp

5.1.1 Summary

Common variables and arrays when running DL_MESO_LBE in both serial and parallel. Required variables and arrays for LBE calculations that are applicable for both serial and parallel running of DL_MESO_LBE (made globally accessible with global.hpp).

5.1.2 Classes

- *struct sSystem*

  Structure for system information.

- *struct sDomain*

  Structure for domain information.

5.1.3 Enumerations

- `enum` *BoundaryType* :: {

  PST = 21, PSD = 22, PSL = 23, PSR = 24, PSF = 25, PSB = 26, CCTRB = 27, CCTLB = 28, CCDLB = 29, CCDRB = 30, CCTRF = 31, CCTLF = 32, CCDLF = 33, CCDRF = 34, CETR = 43, CETL = 44, CEDL = 45, CEDR = 46, CETF = 47, CELF = 48, CEDF = 49, CERF = 50, CETB = 51, CELB = 52, CEDB = 53, CERB = 54

  }

  Types of boundary conditions.

5.1.4 Variables

- *sSystem lbsy*

  System information for LBE simulation.

- *sDomain lbdm*

  Domain information for LBE simulation.

- `double` *lbiniv* `[3]`

  Initial fluid velocities.

- `double` *lbtopv* `[3]`

  Fluid velocity at top boundary.

- `double` *lbbotv* `[3]`

  Fluid velocity at bottom boundary.

- `double` *lbfrov* `[3]`

  Fluid velocity at front boundary.

- `double` *lbbacv* `[3]`

  Fluid velocity at back boundary.

- `double` *lblefv* `[3]`

  Fluid velocity at left boundary.

- `double` *lbrigv* `[3]`

  Fluid velocity at right boundary.

- `double` *lbtopvoscil* `[3]`

  Oscillating fluid velocity at top boundary.

- `double` *lbbotvoscil* `[3]`

  Oscillating fluid velocity at bottom boundary.

- `double` *lbfrovoscil* `[3]`

  Oscillating fluid velocity at front boundary.

- `double` *lbbacvoscil* `[3]`

  Oscillating fluid velocity at back boundary.

- `double` *lblefvoscil* `[3]`

  Oscillating fluid velocity at left boundary.

- `double` *lbrigvoscil* `[3]`

  Oscillating fluid velocity at right boundary.

- `double` *lbtopvfq*

  Angular frequency of oscillating fluid velocity at top boundary.

- `double` *lbbotvfq*

  Angular frequency of oscillating fluid velocity at bottom boundary.

- `double` *lbfrovfq*

  Angular frequency of oscillating fluid velocity at front boundary.

- `double` *lbbacvfq*

  Angular frequency of oscillating fluid velocity at back boundary.

- `double` *lblefvfq*

  Angular frequency of oscillating fluid velocity at left boundary.
- `double` *lbrigvfq*

  Angular frequency of oscillating fluid velocity at right boundary.
- `int` *lbtopvbc*

  Flag indicating kind of velocity boundary condition at top boundary.
- `int` *lbbotvbc*

  Flag indicating kind of velocity boundary condition at bottom boundary.
- `int` *lbfrovbc*

  Flag indicating kind of velocity boundary condition at front boundary.
- `int` *lbbacvbc*

  Flag indicating kind of velocity boundary condition at back boundary.
- `int` *lblefvbc*

  Flag indicating kind of velocity boundary condition at left boundary.
- `int` *lbrigvbc*

  Flag indicating kind of velocity boundary condition at right boundary.
- `double *` *lbincp*

  Constant fluid densities.
- `double *` *lbinip*

  Initial fluid densities.
- `double *` *lbtopp*

  Fluid densities at top boundary.
- `double *` *lbbotp*

  Fluid densities at bottom boundary.
- `double *` *lbfrop*

  Fluid densities at front boundary.
- `double *` *lbbacp*

  Fluid densities at back boundary.
- `double *` *lblefp*

  Fluid densities at left boundary.
- `double *` *lbrigp*

  Fluid densities at right boundary.
- `double *` *lbinic*

  Initial solute concentrations.
- `double *` *lbtopc*

  Solute concentrations at top boundary.
- `double *` *lbbotc*

  Solute concentrations at bottom boundary.

- `double` ⋆ *lbfroc*

  Solute concentrations at front boundary.

- `double` ⋆ *lbbacc*

  Solute concentrations at back boundary.

- `double` ⋆ *lblefc*

  Solute concentrations at left boundary.

- `double` ⋆ *lbrigc*

  Solute concentrations at right boundary.

- `double` *lbsyst*

  Constant system temperature.

- `double` *lbinit*

  Initial temperature.

- `double` *lbtopt*

  Temperature at top boundary.

- `double` *lbbott*

  Temperature at bottom boundary.

- `double` *lbfrot*

  Temperature at front boundary.

- `double` *lbbact*

  Temperature at back boundary.

- `double` *lbleft*

  Temperature at left boundary.

- `double` *lbrigt*

  Temperature at right boundary.

- `double` *lbsysdt*

  System-wide heating rate.

- `double` *lbtopdt*

  Heating rate at top boundary.

- `double` *lbbotdt*

  Heating rate at bottom boundary.

- `double` *lbfrodt*

  Heating rate at front boundary.

- `double` *lbbacdt*

  Heating rate at back boundary.

- `double` *lblefdt*

  Heating rate at left boundary.

- `double` *lbrigdt*

  Heating rate at right boundary.

- `int` *lbtotstep*

  Total number of simulation timesteps.

- `int` *lbequstep*

  Number of equilibration timesteps.

- `int` *lbsave*

  Output file writing frequency.

- `int` *lbdump*

  Simulation restart file writing frequency.

- `int` *lbcurstep*

  Current timestep number.

- `int` *lbsteer*

  Flag for computational steering.

- `int` *lbmpiio*

  Flag for writing output files using MPI-IO.

- `int` *lbrestart*

  Flag for restarting a previous simulation.

- `double` *lbcalctime*

  Total specified calculation time (in seconds) for simulation.

- `double` *lbnoise*

  Noise intensity for fluid densities.

- `double` *lbtrtmagic*

  Two relaxation time (TRT) 'magic number'.

- `double` *lbgasconst*

  Universal gas constant.

- `double *` *lbtf*

  Fluid relaxation frequencies.

- `double *` *lbtfbulk*

  Fluid bulk relaxation frequencies.

- `double *` *lbtfclb3*

  Third-order relaxation frequencies for CLBE collisions.

- `double *` *lbtfclb4*

  Fourth-order relaxation frequencies for CLBE collisions.

- `double *` *lbtc*

  Solute relaxation frequencies.

- `double *` *lbtt*

  Thermal relaxation frequencies.

- `double` *lbtmob*

  Free-energy concentration relaxation frequency for mobility between two fluid species.

- `int *` *lbscpot*

  Pseudopotential types for fluids.

- `int *` *lbwet*

  Solid-fluid wetting type.

- `double` *lbfewet* `[4]`

  Surface free energy parameters for Swift free-energy interactions.

- `int` *lbfeeos*

  Fluid equation of state for Swift free-energy interactions.

- `int` *lbfepot*

  Chemical potential type for Swift free-energy interactions.

- `int` *lbgradord*

  Accuracy of gradient approximation for solid-fluid interactions.

- `int` *lbbctyp*

  Type of constant velocity or density boundary conditions.

- `int` *lbsbctyp*

  Type of constant solute concentration boundary conditions.

- `int` *lbtbctyp*

  Type of constant temperature boundary conditions.

- `double` *lbkappa*

  Surface tension parameter.

- `double` *lbfemob*

  Fluid mobility parameter.

- `double *` *lbg*

  Fluid-fluid interaction parameters.

- `double *` *lbgwall*

  Solid-fluid interaction parameters.

- `double *` *lbseg*

  Fluid-fluid segregation parameters for Lishchuk continuum-based interactions.

- `double *` *lbpsi0*

  Parameters for 1994 Shan-Chen pseudopotential model.

- `double *` *lbcritt*

  Fluid critical temperatures.

- `double *` *lbcritp*

  Fluid critical pressures.

- `double *` *lbeosa*

  Fluid attraction coefficients for equations of state.

- `double *` *lbeosb*

  Fluid finite volume coefficients for equations of state.

- `double ⋆` *lbacentric*

  Fluid acentric factors for equations of state.

- `double ⋆` *lbscquad*

  Weighting factors for quadratic Shan-Chen pseudopotential forces.

- `double ⋆` *lbbdforce*

  Constant body forces on fluids.

- `double ⋆` *lboscilforce*

  Amplitudes of oscillating forces on fluids.

- `double ⋆` *lbbousforce*

  Boussinesq buoyancy forces on fluids.

- `double ⋆` *lbheatforce*

  Heat convection forces.

- `double ⋆` *lbinterforce*

  Interfacial forces.

- `double` *lboscilfreq*

  Angular frequency of oscillating forces acting on fluids.

- `int` *lbbdforcetyp*

  Flag indicating kind of body forces acting on fluids.

- `double ⋆` *lbomega*

  Fluid relaxation frequencies at lattice points.

- `int ⋆` *lbrheo*

  Rheological models for fluids.

- `double ⋆` *lbrheoa*

  Parameters a for rheological models.

- `double ⋆` *lbrheob*

  Parameters b for rheological models.

- `double ⋆` *lbrheoc*

  Parameters c for rheological models.

- `double ⋆` *lbrheod*

  Parameters d for rheological models.

- `double ⋆` *lbrheopower*

  Power-law indices for rheological models.

- `double ⋆` *lbf*

  Distribution functions.

- `double ⋆` *lbft*

  Pseudopotentials, interfacial normals or density/concentration gradients.

- `double ⋆` *lbfeq*

  Local equilibrium distribution functions.

- `int *` *lbphi*

  Boundary condition properties (phase fields).

- `int *` *lbneigh*

  Neighbouring point properties.

- `double *` *lbboundnorm*

  Surface normals for lattice points.

- `int *` *lbvx*

  Link vectors (x-components).

- `int *` *lbvy*

  Link vectors (y-components).

- `int *` *lbvz*

  Link vectors (z-components).

- `int` *lbfevx* `[19]`

  Gradient stencil for Swift free-energy interactions (x-component).

- `int` *lbfevy* `[19]`

  Gradient stencil for Swift free-energy interactions (y-component).

- `int` *lbfevz* `[19]`

  Gradient stencil for Swift free-energy interactions (z-component).

- `int *` *lbopv*

  Conjugate lattice links.

- `int *` *lbspair*

  List of fluid pairs.

- `double *` *lbw*

  Link weights.

- `double *` *lbvwx*

  Products of x-components of link vector and weights.

- `double *` *lbvwy*

  Products of y-components of link vector and weights.

- `double *` *lbvwz*

  Products of z-components of link vector and weights.

- `double *` *lbwi*

  Velocity link weights for Swift free-energy interaction local equilibrium distribution functions.

- `double *` *lbw0*

  Density link weights for Swift free-energy interaction local equilibrium distribution functions.

- `double *` *lbwpt*

  Bulk pressure/surface tension link weights for Swift free-energy interaction local equilibrium distribution functions.

- `double *` *lbwxx*

  Surface tension (xx-direction) link weights for Swift free-energy interaction local equilibrium distribution functions.

- `double` ⋆ *lbwyy*

  Surface tension (yy-direction) link weights for Swift free-energy interaction local equilibrium distribution functions.

- `double` ⋆ *lbwzz*

  Surface tension (zz-direction) link weights for Swift free-energy interaction local equilibrium distribution functions.

- `double` ⋆ *lbwxy*

  Surface tension (xy-direction) link weights for Swift free-energy interaction local equilibrium distribution functions.

- `double` ⋆ *lbwxz*

  Surface tension (xz-direction) link weights for Swift free-energy interaction local equilibrium distribution functions.

- `double` ⋆ *lbwyz*

  Surface tension (yz-direction) link weights for Swift free-energy interaction local equilibrium distribution functions.

- `double` ⋆ *lbwgam*

  Galilean invariance link weights for Swift free-energy interaction local equilibrium distribution functions.

- `double` ⋆ *lbwdel*

  Galilean invariance link weights for Swift free-energy interaction local equilibrium distribution functions.

- `double` ⋆ *lbtr*

  Multiple relaxation time (MRT) transformation matrix.

- `double` ⋆ *lbtrinv*

  Multiple relaxation time (MRT) inverse transformation matrix.

- `double` *lbmrts* `[8]`

  Tuneable collision frequencies for multiple relaxation time (MRT) collisions.

- `double` *lbmrtw* `[3]`

  Tuneable parameters for moments for multiple relaxation time (MRT) collisions.

- `int` *lbsitelength*

  Number of distribution functions per lattice point.

- `double` *lbdx*

  Grid spacing in ‘real world’ units.

- `double` *lbdt*

  Timestep in ‘real world’ units.

- `double` *lbcs*

  Lattice speed of sound.

- `double` *lbcssq*

  Square of lattice speed of sound.

- `double` *lbrcssq*

  Reciprocal of the square of lattice speed of sound.

- `double` *lbsoundv*

  Speed of sound of fluid 0.

- `double` *lbreynolds*

  Reynolds number for LBE simulation.
- `double` *lbkinetic*

  Kinematic viscosity of fluid 0.
- `double` *lbbousth*

  High temperature for Boussinesq approximation.
- `double` *lbboustl*

  Low temperature for Boussinesq approximation.
- `double` *lbevaplim*

  Evaporation limit for fluid density.
- `unsigned long *` *lbouter*

  List of lattice points at outer edge of subdomain.
- `int` *lboutersize*

  Number of lattice points in outer edge region of subdomain.
- `int` *bigend*

  Flag for big endianness of system.
- `double` *timetotal*

  Total calculation time in seconds.
- `int` *qVersion*

  Output file number.
- `int` *collide*

  Collision type.
- `int` *interact*

  Interaction type.
- `int` *incompress*

  Flag for incompressible fluids.
- `int` *nonnewtonian*

  Flag for variable fluid relaxation times (non-Newtonian rheology).
- `int` *outformat*

  Output file format.
- `int` *postequil*

  Flag for post-equilibration state.

5.1.5 Class Documentation

struct sSystem

Structure for system information (e.g. numbers of dimensions and lattice links per grid point, fluids, solutes, temperature fields, size of lattice).

Table 5.1: Class Members

int	nc	Number of solutes in LBE simulation: helps specify the total number of lattices for the simulation, as each solute exists on its own lattice. If at least one solute is in use, only one fluid can currently be used.
int	nd	Number of dimensions for lattice and simulation: used to specify lattice scheme
int	nf	Number of fluids in LBE simulation: helps specify the total number of lattices for the simulation, as each fluid exists on its own lattice
int	nq	Number of lattice link vectors from each grid point: used to specify lattice scheme
int	nt	Number of temperature fields in LBE simulation, which can either be 0 or 1: helps specify the total number of lattices for the simulation, as the temperature field exists on its own lattice
int	nx	Number of grid points in x-direction for LBE simulation
int	ny	Number of grid points in y-direction for LBE simulation
int	nz	Number of grid points in z-direction for LBE simulation. This value needs to be set to 1 for two-dimensional simulations.
int	pf	Number of phase fields in LBE simulation: this property is currently not used in DL_MESO_LBE for any of its interaction models, but only one value per phase field per grid point is required

struct sDomain

Structure for domain information (e.g. processor number, numbers of processors, boundary halo size, dimensions of lattice subdomain).

Table 5.2: Class Members

int	bwid	Size of boundary halo to receive communications from neighbouring processors.
int	owidx	Size of boundary region at edge of lattice (used to apply modulo functions for periodic boundaries) in x-direction. This value is set to the default of 1.
int	owidy	Size of boundary region at edge of lattice (used to apply modulo functions for periodic boundaries) in y-direction. This value is set to the default of 1.
int	owidz	Size of boundary region at edge of lattice (used to apply modulo functions for periodic boundaries) in z-direction. This value is set to 0 for two-dimensional simulations and 1 for three-dimensional simulations.
int	rank	Number (rank) of current processor: used to identify processor.
int	size	Total number of processors involved in LBE calculation.
int	touter	Total number of grid points (including the boundary halo) held by the current processor for its lattice subdomain.
int	xcor	Position of current processor within system given as the x-component of a Cartesian coordinate.
int	xdim	Total number of processors for current LBE calculation in x-direction.
int	xe	Highest x-coordinate for lattice subdomain of current processor (excluding boundary halo).
int	xin-ner	Number of grid points in x-direction for the lattice subdomain of current processor excluding the boundary halo.
int	xouter	Number of grid points in x-direction for the lattice subdomain of current processor including the boundary halo.
int	xs	Lowest x-coordinate for lattice subdomain of current processor (excluding boundary halo).
int	ycor	Position of current processor within system given as the y-component of a Cartesian coordinate.
int	ydim	Total number of processors for current LBE calculation in y-direction.
int	ye	Highest y-coordinate for lattice subdomain of current processor (excluding boundary halo).
int	yin-ner	Number of grid points in y-direction for the lattice subdomain of current processor excluding the boundary halo.
int	youter	Number of grid points in y-direction for the lattice subdomain of current processor including the boundary halo.
int	ys	Lowest y-coordinate for lattice subdomain of current processor (excluding boundary halo).
int	zcor	Position of current processor within system given as the z-component of a Cartesian coordinate.
int	zdim	Total number of processors for current LBE calculation in z-direction. This value is set to 1 for two-dimensional simulations.
int	ze	Highest z-coordinate for lattice subdomain of current processor (excluding boundary halo).
int	zin-ner	Number of grid points in z`-direction for the lattice subdomain of current processor excluding the boundary halo.
int	zouter	Number of grid points in z-direction for the lattice subdomain of current processor including the boundary halo.
int	zs	Lowest z-coordinate for lattice subdomain of current processor (excluding boundary halo).

Enumeration Type Documentation

BoundaryType

```
enum BoundaryType
```

Numerical identifiers for constant density/velocity, concentration and temperature boundary points based on direction. All of these are in use for three-dimensional LBE simulations: two-dimensional simulations make use of concave edges and corners pointing to the front (CCTRF, CCDTLF, CCDLF, CCDRF, CETF, CELF, CEDF, CERF).

Table 5.3: Enumerator

PST	Planar surface (bottom) pointing upwards.
PSD	Planar surface (top) pointing downwards.
PSL	Planar surface (right) pointing to the left.
PSR	Planar surface (left) pointing to the right.
PSF	Planar surface (back) pointing to the front.
PSB	Planar surface (front) pointing to the back.
CCTRB	Concave corner (bottom-left-front) pointing upwards, to the right and back.
CCTLB	Concave corner (bottom-right-front) pointing upwards, to the left and back.
CCDLB	Concave corner (top-right-front) pointing downwards, to the left and back.
CCDRB	Concave corner (top-left-front) pointing downwards, to the right and back.
CCTRF	Concave corner (bottom-left-back) pointing upwards, to the right and front.
CCTLF	Concave corner (bottom-right-back) pointing upwards, to the left and ftont.
CCDLF	Concave corner (top-right-back) pointing downwards, to the left and front.
CCDRF	Concave corner (top-left-back) pointing downwards, to the right and front.
CETR	Concave edge (bottom-left) pointing upwards to the right.
CETL	Concave edge (bottom-right) pointing upwards to the left.
CEDL	Concave edge (top-right) pointing downwards to the left.
CEDR	Concave edge (top-left) pointing downwards to the right.
CETF	Concave edge (bottom-back) pointing upwards to the front.
CELF	Concave edge (right-back) pointing to the left and front.
CEDF	Concave edge (top-back) pointing downwards to the front.
CERF	Concave edge (left-back) pointing to the right and front.
CETB	Concave edge (bottom-front) pointing to the top and back.
CELB	Concave edge (right-front) pointing to the left and back.
CEDB	Concave edge (top-front) pointing to the bottom and back.
CERB	Concave edge (left-front) pointing to the right and back.

Variable Documentation

bigend

```
extern int bigend
```

Flag to indicate whether or not the system stores binary numbers in big endian order (1 = big endian, 0 = little endian).

collide

```
extern int collide
```

Parameter for type of collisions and forcing scheme, used to select collision subroutines for simulation.

incompress

```
extern int incompress
```

Flag to indicate whether or not fluids are fully incompressible (1 = fully incompressible, 0 = mildly compressible).

interact

```
extern int interact
```

Parameter for type of interactions between fluids and/or phases, used to select main simulation loop.

lbacentric

```
extern double* lbacentric
```

Acentric factors ω for fluids, used in equations of state applied by Shan-Chen pseudopotential and Swift free-energy interaction models.

lbbacc

```
extern double* lbbacc
```

Concentrations of solutes at back boundary when using constant concentration boundary condition.

lbbacdt

```
extern double lbbacdt
```

Constant rate of change of temperature at back boundary when using temperature boundary condition.

lbbacp

```
extern double* lbbacp
```

Densities of fluids at back boundary when using constant density boundary condition.

lbbact

```
extern double lbbact
```

Temperature at back boundary when using constant temperature boundary condition.

lbbacv

```
extern double lbbacv[3]
```

Velocity for fluids at back boundary when using constant velocity boundary condition.

lbbacvbc

```
extern int lbbacvbc
```

Flag to indicate whether or not to include sinusoidal oscillation in time to velocity at back boundary when using constant velocity boundary condition (0 = off, 1 = on).

lbbacvfq

```
extern double lbbacvfq
```

Angular frequency of fluid velocity sinusoidal oscillations applied at back boundary when using constant velocity boundary condition.

lbbacvoscil

```
extern double lbbacvoscil[3]
```

Amplitude of velocity for fluids at back boundary varying sinusoidally with time when using constant velocity boundary condition.

lbbctyp

```
extern int lbbctyp
```

Type of boundary conditions in use for constant fluid velocities or densities (0 = Zou-He, 1 = Inamuro, 2 = regularised, 10 = simple Zou-He, 11 = kinetic).

lbbdforce

```
extern double* lbbdforce
```

Constant body forces (e.g. gravity) acting on each fluid, applied using forcing terms during collisions.

lbbdforcetyp

```
extern int lbbdforcetyp
```

Flag to indicate whether or not to include sinusoidal oscillation in time to forces acting on all fluids (0 = off, 1 = on).

lbbotc

```
extern double* lbbotc
```

Concentrations of solutes at bottom boundary when using constant concentration boundary condition.

lbbotdt

```
extern double lbbotdt
```

Constant rate of change of temperature at bottom boundary when using temperature boundary condition.

lbbotp

```
extern double* lbbotp
```

Densities of fluids at bottom boundary when using constant density boundary condition.

lbbott

```
extern double lbbott
```

Temperature at bottom boundary when using constant temperature boundary condition.

lbbotv

```
extern double lbbotv[3]
```

Velocity for fluids at bottom boundary when using constant velocity boundary condition.

lbbotvbc

```
extern int lbbotvbc
```

Flag to indicate whether or not to include sinusoidal oscillation in time to velocity at bottom boundary when using constant velocity boundary condition (0 = off, 1 = on).

lbbotvfq

```
extern double lbbotvfq
```

Angular frequency of fluid velocity sinusoidal oscillations applied at bottom boundary when using constant velocity boundary condition.

lbbotvoscil

```
extern double lbbotvoscil[3]
```

Amplitude of velocity for fluids at bottom boundary varying sinusoidally with time when using constant velocity boundary condition.

lbboundnorm

```
extern double* lbboundnorm
```

Normals of surfaces at lattice points for calculating solid-fluid interfacial forces using Lishchuk continuum-based interactions.

lbbousforce

```
extern double* lbbousforce
```

Products of gravitational acceleration and volumetric expansivity $\vec{g}\beta$ for fluids, used to determine temperature-dependent buoyancy forces for Boussinesq approximation.

lbbousth

```
extern double lbbousth
```

Maximum system temperature used in temperature-dependent forces for Boussinesq approximation T_h.

lbboustl

```
extern double lbboustl
```

Minimum system temperature used in temperature-dependent forces for Boussinesq approximation T_l.

lbcalctime

```
extern double lbcalctime
```

Total wall time (in seconds) to run DL_MESO_LBE calculation before writing a restart file (lbout.dump) and shutting down. (If this value is zero, the calculation will continue until all timesteps have been completed.)

lbcritp

```
extern double* lbcritp
```

Critical pressures p_c for fluids, connected to parameters for equations of state.

lbcritt

```
extern double* lbcritt
```

Critical temperatures T_c for fluids, connected to parameters for equations of state.

lbcs

```
extern double lbcs
```

Speed of sound for the lattice, c_s, given in lattice-based units.

lbcssq

```
extern double lbcssq
```

Square of the speed of sound for the lattice, c_s^2, given in lattice-based units.

lbcurstep

```
extern int lbcurstep
```

Timestep number for current point in LBE simulation.

lbdm

```
extern sDomain lbdm
```

Domain information for LBE simulation (e.g. extents of lattice held by processors), using *struct sDomain* structure to store information.

lbdt

```
extern double lbdt
```

Timestep Δt given in consistent 'real world' units, calculated from the kinetic viscosity and speed of sound in 'real world' units and the selected relaxation time for fluid 0. This value is checked to ensure a valid value for the relaxation time/frequency of fluid 0 is selected.

lbdump

```
extern int lbdump
```

Interval (number of timesteps) between overwrites of a simulation restart file (lbout.dump).

lbdx

```
extern double lbdx
```

Grid spacing between lattice points Δx given in consistent ‘real world’ units, calculated from the speed of sound for fluid 0 and the ‘real world’ timestep size.

lbeosa

```
extern double* lbeosa
```

Attraction coefficients a for fluids, used in equations of state applied by Shan-Chen pseudopotential and Swift free-energy interaction models.

lbeosb

```
extern double* lbeosb
```

Finite volume coefficients b for fluids, used in equations of state applied by Shan-Chen pseudopotential and Swift free-energy interaction models.

lbequstep

```
extern int lbequstep
```

Number of timesteps required at start of LBE simulation to allow system to settle before applying boundary conditions and body forces.

lbevaplim

```
extern double lbevaplim
```

Minimum viable fluid density for applying Inamuro and kinetic boundary conditions, and calculating mass fractions and fluid velocities for output files.

lbf

```
extern double* lbf
```

Distribution functions (f_i, g_i, h_i) for all lattice points in subdomain: in order of fluid/solute/temperature field, then lattice link and then lattice point.

lbfeeos

```
extern int lbfeeos
```

Selected equation of state for one or two fluids undergoing Swift free-energy interactions.

lbfemob

```
extern double lbfemob
```

Mobility parameter (multiplier) Γ between two fluids undergoing Swift free-energy interactions.

lbfepot

```
extern int lbfepot
```

Selected chemical potential between two fluids undergoing Swift free-energy interactions (0 = none, 1 = quartic).

lbfeq

```
extern double* lbfeq
```

Local equilibrium distribution functions (f_i^{eq}, g_i^{eq}, h_i^{eq}) for a lattice point, used during system initialisation and collisions.

lbfevx

```
extern int lbfevx[19]
```

Vectors (x-components) for calculating first-order and second-order gradients of density and concentration for Swift free-energy calculations with reduced microcurrents [102] used at fluid lattice points without neighbouring boundaries.

lbfevy

```
extern int lbfevy[19]
```

Vectors (y-components) for calculating first-order and second-order gradients of density and concentration for Swift free-energy calculations with reduced microcurrents [102] used at fluid lattice points without neighbouring boundaries.

lbfevz

```
extern int lbfevz[19]
```

Vectors (z-components) for calculating first-order and second-order gradients of density and concentration for Swift free-energy calculations with reduced microcurrents [102] used at fluid lattice points without neighbouring boundaries.

lbfewet

```
extern double lbfewet[4]
```

Parameters for surface free energy with Swift free-energy interactions: indices 0 and 1 for linear and quadratic terms in density, 2 and 3 for linear and quadratic terms in concentration.

lbfroc

```
extern double* lbfroc
```

Concentrations of solutes at front boundary when using constant concentration boundary condition.

lbfrodt

```
extern double lbfrodt
```

Constant rate of change of temperature at front boundary when using temperature boundary condition.

lbfrop

```
extern double* lbfrop
```

Densities of fluids at front boundary when using constant density boundary condition.

lbfrot

```
extern double lbfrot
```

Temperature at front boundary when using constant temperature boundary condition.

lbfrov

```
extern double lbfrov[3]
```

Velocity for fluids at front boundary when using constant velocity boundary condition.

lbfrovbc

```
extern int lbfrovbc
```

Flag to indicate whether or not to include sinusoidal oscillation in time to velocity at front boundary when using constant velocity boundary condition (0 = off, 1 = on).

lbfrovfq

```
extern double lbfrovfq
```

Angular frequency of fluid velocity sinusoidal oscillations applied at front boundary when using constant velocity boundary condition.

lbfrovoscil

```
extern double lbfrovoscil[3]
```

Amplitude of velocity for fluids at front boundary varying sinusoidally with time when using constant velocity boundary condition.

lbft

```
extern double* lbft
```

Pseudopotentials for Shan-Chen interactions (ordered by fluid), interfacial normals for Lishchuk continuum-based interactions (ordered by fluid pairs) or gradients of density/concentration for Swift free-energy interactions (all three components of first-order gradients and then secord-order gradient for density and concentration).

lbg

```
extern double* lbg
```

Interaction parameters between fluids (including self-interactions) g_{ab}, used for Shan-Chen pseudopotential and Lishchuk continuum-based interactions.

lbgasconst

```
extern double lbgasconst
```

Universal gas constant R used for equations of state applied in Shan-Chen pseudopotential and Swift free-energy interactions.

lbgradord

```
extern int lbgradord
```

Selected accuracy of one-sided gradient approximations used at lattice points next to solid boundaries for Lishchuk continuum-based phase index gradients and interfacial forces and Swift free-energy solid-fluid interactions.

lbgwall

```
extern double* lbgwall
```

Interaction parameters between fluids and solid walls $g_{wall,a}$, used for Shan-Chen pseudopotential and Lishchuk continuum-based interactions.

lbheatforce

```
extern double* lbheatforce
```

Forces acting on fluids due to heat convection as given by Boussinesq approximation.

lbincp

```
extern double* lbincp
```

Constant densities for fully incompressible fluids in LBE simulation (used in local equilibrium distribution functions, and as parameters for 1994 Shan-Chen [119] and Qian [106] pseudopotentials).

lbinic

```
extern double* lbinic
```

Initial default concentrations for solutes at all lattice points in LBE simulation (unless overridden by values in initial state input file lbin.init).

lbinip

```
extern double* lbinip
```

Initial default densities for fluids at all lattice points in LBE simulation (unless overridden by values in initial state input file lbin.init).

lbinit

```
extern double lbinit
```

Initial default temperature at all lattice points in LBE simulation (unless overriden by values in initial state input file lbin.init).

lbiniv

```
extern double lbiniv[3]
```

Initial default velocities for fluids at all lattice points in LBE simulation (unless overridden by values in initial state input file lbin.init).

lbinterforce

```
extern double* lbinterforce
```

Forces acting on fluids due to interfaces between fluids or phases (obtained from Shan-Chen pseudopotential or Lishchuk continuum-based interactions).

lbkappa

```
extern double lbkappa
```

Surface tension parameter κ between phases and/or fluids for Swift free-energy intereactions.

lbkinetic

```
extern double lbkinetic
```

Kinematic viscosity (quotient of dynamic viscosity with density) for fluid 0 given in consistent 'real world' units, used to find actual timestep size.

lblefc

```
extern double* lblefc
```

Concentrations of solutes at left boundary when using constant concentration boundary condition.

lblefdt

```
extern double lblefdt
```

Constant rate of change of temperature at left boundary when using temperature boundary condition.

lblefp

```
extern double* lblefp
```

Densities of fluids at left boundary when using constant density boundary condition.

lbleft

```
extern double lbleft
```

Temperature at left boundary when using constant temperature boundary condition.

lblefv

```
extern double lblefv[3]
```

Velocity for fluids at left boundary when using constant velocity boundary condition.

lblefvbc

```
extern int lblefvbc
```

Flag to indicate whether or not to include sinusoidal oscillation in time to velocity at left boundary when using constant velocity boundary condition (0 = off, 1 = on).

lblefvfq

```
extern double lblefvfq
```

Angular frequency of fluid velocity sinusoidal oscillations applied at left boundary when using constant velocity boundary condition.

lblefvoscil

```
extern double lblefvoscil[3]
```

Amplitude of velocity for fluids at left boundary varying sinusoidally with time when using constant velocity boundary condition.

lbmpiio

```
extern int lbmpiio
```

Flag to indicate if a single output file per simulation snapshot is to be written using MPI-IO (0 = off, 1 = on).

lbmrts

```
extern double lbmrts[8]
```

System-wide all-fluid collision frequencies for multiple relaxation time (MRT) collisions for non-hydrodynamic moments intended to enhance the numerical stability of LBE calculations. (Default values are provided for each lattice scheme but can be overridden by the user.)

lbmrtw

```
extern double lbmrtw[3]
```

System-wide parameters for local equilibrium values of energy-squared and diagonal fourth-order moments used in multiple relaxation time (MRT) collisions. (Default values are provided for D2Q9, D3Q15 and D3Q19 lattice schemes to boost numerical stablity of LBE calculations, which can only be changed by modifying source code: no values are required for the D3Q27 lattice scheme.)

lbneigh

```
extern int* lbneigh
```

Numbers indicating existence and directions of neighbouring boundary lattice sites, with a number's hundreds indicating x-dimension, tens indicating y-dimension and units indicating z-dimension. The digits indicate existence of neighbouring boundary points in particular directions and any fluid points available for calculating gradients of fluid properties.

lbnoise

```
extern double lbnoise
```

Maximum amplitude of random noise in fluid densities (relative to specified values) when setting up LBE simulation.

lbomega

```
extern double* lbomega
```

Relaxation frequencies for all fluids at each lattice point: these will change over time when using non-Newtonian rheological models for fluids.

lbopv

```
extern int* lbopv
```

Lattice links that are conjugates (exact opposites) to current links.

lboscilforce

```
extern double* lboscilforce
```

Amplitudes of oscillating forces acting on fluids throughout lattice that vary sinusoidally with time.

lboscilfreq

```
extern double lboscilfreq
```

Angular frequency of force sinusoidal oscillations applied at top boundary when using constant velocity boundary condition.

lbouter

```
extern unsigned long* lbouter
```

List of lattice points (given as one-dimensional positions) at the outer edge of the processor's subdomain, used to distinguish boundary halo regions in parallel and lattice points with neighbours on opposite sides of the grid in serial.

lboutersize

```
extern int lboutersize
```

Total number of lattice points at the outer edge of the processor's subdomain.

lbphi

```
extern int* lbphi
```

Numbers indicating existence, type and directions for boundary conditions for all lattice points in subdomain.

lbpsi0

```
extern double* lbpsi0
```

Parameters (multipliers) for 1994 Shan-Chen pseudopotential model [119], ψ_0.

lbrcssq

```
extern double lbrcssq
```

Reciprocal of the square of the speed of sound for the lattice, c_s^{-2}, given in lattice-based units.

lbrestart

```
extern int lbrestart
```

Flag to indicate whether or not to read in the state of a previous simulation from a restart file (lbout.dump) and resume that simulation (0 = off, 1 = on).

lbreynolds

```
extern double lbreynolds
```

Reynolds number $Re = \frac{uL}{\nu}$ representative of the LBE simulation, used in Plot3D solution files.

lbrheo

```
extern int* lbrheo
```

Identifiers of rheological models for fluids: 0 = constant kinematic viscosity, 1 = constant dynamic viscosity, 2 = power law, 3 = Bingham plastic, 4 = Herschel-Bulkley, 5 = Casson, 6 = Carreau-Yasuda.

lbrheoa

```
extern double* lbrheoa
```

First parameters for rheological models (ν_0, μ_0 or μ_∞).

lbrheob

```
extern double* lbrheob
```

Second parameters for rheological models (σ_y, μ_0 or $(\mu_0 - \mu_\infty)$).

lbrheoc

```
extern double* lbrheoc
```

Third parameters for rheological models (λ or exponential decay parameter m for models with yield stresses).

lbrheod

```
extern double* lbrheod
```

Fourth parameters for rheological models (λ or exponential decay parameter m for models with yield stresses).

lbrheopower

```
extern double* lbrheopower
```

Indices n for power-law-based rheological models (power law, Herschel-Bulkley, Carreau-Yasuda).

lbrigc

```
extern double* lbrigc
```

Concentrations of solutes at right boundary when using constant concentration boundary condition.

lbrigdt

```
extern double lbrigdt
```

Constant rate of change of temperature at right boundary when using temperature boundary condition.

lbrigp

```
extern double* lbrigp
```

Densities of fluids at right boundary when using constant density boundary condition.

lbrigt

```
extern double lbrigt
```

Temperature at right boundary when using constant temperature boundary condition.

lbrigv

```
extern double lbrigv[3]
```

Velocity for fluids at right boundary when using constant velocity boundary condition.

lbrigvbc

```
extern int lbrigvbc
```

Flag to indicate whether or not to include sinusoidal oscillation in time to velocity at right boundary when using constant velocity boundary condition (0 = off, 1 = on).

lbrigvfq

```
extern double lbrigvfq
```

Angular frequency of fluid velocity sinusoidal oscillations applied at right boundary when using constant velocity boundary condition.

lbrigvoscil

```
extern double lbrigvoscil[3]
```

Amplitude of velocity for fluids at right boundary varying sinusoidally with time when using constant velocity boundary condition.

lbsave

```
extern int lbsave
```

Interval (number of timesteps) between each simulation snapshot written to output file(s).

lbsbctyp

```
extern int lbsbctyp
```

Type of boundary conditions in use for constant solute concentrations: 0 = Zou-He, 1 = Inamuro.

lbscpot

```
extern int* lbscpot
```

Shan-Chen pseudopotential types for fluids to obtain particular equations of state.

lbscquad

```
extern double* lbscquad
```

Weighting factors between fluids for Shan-Chen interfacial forces with quadratic pseudopotential terms β_{ab}.

lbseg

```
extern double* lbseg
```

Post-collisional segregation parameters between fluids β^{ab}, used for Lishchuk continuum-based interactions.

lbsitelength

```
extern int lbsitelength
```

Total number of distribution functions per lattice point, equal to the product of the number of links per point and the sum of numbers of fluids, solutes and temperature fields.

lbsoundv

```
extern double lbsoundv
```

Speed of sound for fluid 0 given in consistent 'real world' units, used to find actual grid spacing and timestep sizes.

lbspair

```
extern int* lbspair
```

List of unlink fluid pairs to calculate interfacial normals for Lishchuk continuum-based interactions, also used to calculate interfacial forces and apply post-collisional segregation.

lbsteer

```
extern int lbsteer
```

Flag to indicate if computational steering is to be applied during DL_MESO_LBE run (currently not in use) (0 = off, 1 = on).

lbsy

```
extern sSystem lbsy
```

System information for LBE simulation (e.g. lattice scheme, system size, numbers of fluids), using *struct sSystem* structure to store information.

lbsysdt

```
extern double lbsysdt
```

Constant rate of change of initial system-wide temperature at all lattice points when using temperature fields.

lbsyst

```
extern double lbsyst
```

System-wide temperature used in equations of state for Shan-Chen pseudiopotential and Swift free-energy interactions.

lbtbctyp

```
extern int lbtbctyp
```

Type of boundary conditions in use for constant temperatures: 0 = Zou-He, 1 = Inamuro.

lbtc

```
extern double* lbtc
```

Relaxation frequencies $\omega_c = \frac{1}{\tau_c}$ for solutes, related to (mass) diffusivities.

lbtf

```
extern double* lbtf
```

Relaxation frequencies $\omega = \frac{1}{\tau}$ for fluids (or symmetric relaxation times for TRT collisions), related to kinematic viscosities. (These are initial values when using non-Newtonian rheological models.)

lbtfbulk

```
extern double* lbtfbulk
```

Bulk relaxation frequencies $\omega_b = \frac{1}{\tau_b}$ for fluids, related to bulk viscosities.

lbtfclb3

```
extern double* lbtfclb3
```

Third-order relaxation frequencies $\omega_3 = \frac{1}{\tau_3}$ for fluids, used as numerical stability parameters for cascaded LBE (CLBE) collisions.

lbtfclb4

```
extern double* lbtfclb4
```

Fourth-order relaxation frequencies $\omega_4 = \frac{1}{\tau_4}$ for fluids, used as numerical stability parameters for cascaded LBE (CLBE) collisions.

lbtmob

```
extern double lbtmob
```

Concentration relaxation frequency $\omega_\phi = \frac{1}{\tau_\phi}$ between two fluid species with Swift free-energy interactions, related to mobility.

lbtopc

```
extern double* lbtopc
```

Concentrations of solutes at top boundary when using constant concentration boundary condition.

lbtopdt

```
extern double lbtopdt
```

Constant rate of change of temperature at top boundary when using temperature boundary condition.

lbtopp

```
extern double* lbtopp
```

Densities of fluids at top boundary when using constant density boundary condition.

lbtopt

```
extern double lbtopt
```

Temperature at top boundary when using constant temperature boundary condition.

lbtopv

```
extern double lbtopv[3]
```

Velocity for fluids at top boundary when using constant velocity boundary condition.

lbtopvbc

```
extern int lbtopvbc
```

Flag to indicate whether or not to include sinusoidal oscillation in time to velocity at top boundary when using constant velocity boundary condition (0 = off, 1 = on).

lbtopvfq

```
extern double lbtopvfq
```

Angular frequency of fluid velocity sinusoidal oscillations applied at top boundary when using constant velocity boundary condition.

lbtopvoscil

```
extern double lbtopvoscil[3]
```

Amplitude of velocity for fluids at top boundary varying sinusoidally with time when using constant velocity boundary condition.

lbtotstep

```
extern int lbtotstep
```

Total number of timesteps required for LBE simulation.

lbtr

```
extern double* lbtr
```

Matrix used in multiple relaxation time (MRT) collisions $\mathbf{T}$ to transform distribution functions to moments prior to applying collisions.

lbtrinv

```
extern double* lbtrinv
```

Matrix used in multiple relaxation time (MRT) collisions $\mathbf{T}^{-1}$ to transform moments to distribution functions after applying collisions.

lbtrtmagic

```
extern double lbtrtmagic
```

'Magic number' for two relaxation time (TRT) symmetric and anti-symmetric relaxation times: $\Lambda_{eo} = \left(\tau^{+} - \frac{1}{2}\right)\left(\tau^{-} - \frac{1}{2}\right)$.

lbtt

```
extern double* lbtt
```

Relaxation frequencies $\omega_t = \frac{1}{\tau_t}$ for temperature fields, related to thermal diffusivities (conduction). (Only one thermal relaxation frequency is currently used.)

lbvwx

```
extern double* lbvwx
```

Products of x-components of link vectors and the corresponding weighting parameter, $w_i e_{i,x}$, used for gradient calculations and post-collisional segregations in Shan-Chen pseudopotential and Lishchuk continuum-based interactions.

lbvwy

```
extern double* lbvwy
```

Products of y-components of link vectors and the corresponding weighting parameter, $w_i e_{i,y}$, used for gradient calculations and post-collisional segregations in Shan-Chen pseudopotential and Lishchuk continuum-based interactions.

lbvwz

```
extern double* lbvwz
```

Products of z-components of link vectors and the corresponding weighting parameter, $w_i e_{i,z}$, used for gradient calculations and post-collisional segregations in Shan-Chen pseudopotential and Lishchuk continuum-based interactions.

lbvx

```
extern int* lbvx
```

Link (speed) vectors for the lattice scheme in use (x-components), $e_{i,x}$.

lbvy

```
extern int* lbvy
```

Link (speed) vectors for the lattice scheme in use (y-components), $e_{i,y}$.

lbvz

```
extern int* lbvz
```

Link (speed) vectors for the lattice scheme in use (z-components), $e_{i,z}$.

lbw

```
extern double* lbw
```

Weighting parameters for each lattice link w_i used to calculate local equilibrium distribution functions (excluding those for Swift free-energy interactions).

lbw0

```
extern double* lbw0
```

Weighting parameters for each lattice link w_i^{00} used for density-dependent terms in local equilibrium distribution functions for Swift free-energy interactions.

lbwdel

```
extern double* lbwdel
```

Weighting parameters for each lattice link δ_i used for Galilean invariance correction terms in local equilibrium distribution functions for Swift free-energy interactions.

lbwet

```
extern int* lbwet
```

Type of solid-fluid wetting to be applied at lattice points close to solid boundaries, for either Shan-Chen pseudopotential (0 = density, 1 = potential, 2 = screened potential) or Swift free-energy interactions (0 = none, 1 = quadratic).

lbwgam

```
extern double* lbwgam
```

Weighting parameters for each lattice link γ_i used for Galilean invariance correction terms in local equilibrium distribution functions for Swift free-energy interactions.

lbwi

```
extern double* lbwi
```

Weighting parameters for each lattice link w_i used for velocity-dependent terms in local equilibrium distribution functions for Swift free-energy interactions.

lbwpt

```
extern double* lbwpt
```

Weighting parameters for each lattice link $w_i^p = w_i^t$ used for bulk pressure and second-order gradient terms for surface tension in local equilibrium distribution functions for Swift free-energy interactions.

lbwxx

```
extern double* lbwxx
```

Weighting parameters for each lattice link w_i^{xx} used for xx-component first-order gradient terms for surface tension in local equilibrium distribution functions for Swift free-energy interactions.

lbwxy

```
extern double* lbwxy
```

Weighting parameters for each lattice link w_i^{xy} used for xy-component first-order gradient terms for surface tension in local equilibrium distribution functions for Swift free-energy interactions.

lbwxz

```
extern double* lbwxz
```

Weighting parameters for each lattice link w_i^{xz} used for xz-component first-order gradient terms for surface tension in local equilibrium distribution functions for Swift free-energy interactions.

lbwyy

```
extern double* lbwyy
```

Weighting parameters for each lattice link w_i^{yy} used for yy-component first-order gradient terms for surface tension in local equilibrium distribution functions for Swift free-energy interactions.

lbwyz

```
extern double* lbwyz
```

Weighting parameters for each lattice link w_i^{yz} used for yz-component first-order gradient terms for surface tension in local equilibrium distribution functions for Swift free-energy interactions.

lbwzz

```
extern double* lbwzz
```

Weighting parameters for each lattice link w_i^{zz} used for zz-component first-order gradient terms for surface tension in local equilibrium distribution functions for Swift free-energy interactions.

nonnewtonian

```
extern int nonnewtonian
```

Flag to indicate whether or not fluid relaxation frequencies (and shear rates) are to be calculated for each timestep, i.e. if a rheological model is to be applied: 0 = no rheology calculations, 1 = calculations of relaxation frequencies for Newtonian fluids without calculating shear rates, 2 = calculations of shear rates and relaxation frequencies for non-Newtonian fluids.

outformat

```
extern int outformat
```

Flag to indicate format for output files with simulation snapshots: 0 = binary XML-based Vtk, 1 = binary Legacy VTK, 2 = binary Plot3D, 3 = ANSI/text XML-based VTK, 4 = ANSI/text Legacy VTK, 5 = ANSI/text Plot3D.

postequil

```
extern int postequil
```

Flag to indicate whether or not equilibration period has passed (1 = yes, 0 = no): boundary conditions other than bounce-back and body forces are only applied after equilibration.

qVersion

```
extern int qVersion
```

Upcoming number for output file(s) of simulation snapshot, used in filename(s) for output file(s).

timetotal

```
extern double timetotal
```

Total wall time (in seconds) spent on LBE calculations during DL_MESO_LBE run: used to determine efficiency measure (MLUPS, millions of lattice updates per second).

5.2 plbe.hpp

Common variables and arrays when running DL_MESO_LBE in parallel.

Required variables and arrays for LBE calculations that are applicable for parallel running of DL_MESO_LBE.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <ctime>
#include <cstdio>
#include <cmath>
#include <iomanip>
#include <string>
#include <cstring>
#include <sstream>
#include <vector>
#include <sys/time.h>
#include <sys/stat.h>
#include <mpi.h>
#include "lbe.hpp"
```

5.2.1 Classes

- *struct sNeighbour*

  Structure for neighbouring processor information.

- *struct sIOGroup*

  Structure for I/O group.

5.2.2 Macros

- `#define` *omp_get_num_thread*

  Setting OpenMP number of threads when compiling DL_MESO_LBE without OpenMP.

- `#define` *omp_get_thread_num*

  Setting OpenMP thread number when compiling DL_MESO_LBE without OpenMP.

5.2.3 Variables

- *sNeighbour lbnb*

  Neighbour information for LBE simulation on current processor.
- *sIOGroup lbIOGroup*

  I/O group information for LBE simulation on current processor.
- `MPI_Comm` *ioRootCommunicator*

  MPI communicator among file writing processors.
- `MPI_File` *output_handle*

  MPI file handle for writing simulation snapshot output files.
- `MPI_File` *dump_handle*

  MPI file handle for writing simulation restart files.
- `MPI_Datatype` *lbmsg2x*

  MPI derived datatype to send/receive distribution functions in x-direction in 2D.
- `MPI_Datatype` *lbmsg2y*

  MPI derived datatype to send/receive distribution functions in y-direction in 2D.
- `MPI_Datatype` *lbmsg3x*

  MPI derived datatype to send/receive distribution functions in x-direction in 3D.
- `MPI_Datatype` *lbmsg3y*

  MPI derived datatype to send/receive distribution functions in y-direction in 3D.
- `MPI_Datatype` *lbmsg3z*

  MPI derived datatype to send/receive distribution functions in z-direction in 3D.
- `MPI_Datatype` *lbbmsg2x*

  MPI derived datatype to send/receive boundary conditions (phase fields) in x-direction in 2D.
- `MPI_Datatype` *lbbmsg2y*

  MPI derived datatype to send/receive boundary conditions (phase fields) in y-direction in 2D.
- `MPI_Datatype` *lbbmsg3x*

  MPI derived datatype to send/receive boundary conditions (phase fields) in x-direction in 3D.
- `MPI_Datatype` *lbbmsg3y*

  MPI derived datatype to send/receive boundary conditions (phase fields) in y-direction in 3D.
- `MPI_Datatype` *lbbmsg3z*

  MPI derived datatype to send/receive boundary conditions (phase fields) in z-direction in 3D.
- `MPI_Datatype` *lbfmsg2x*

  MPI derived datatype to send/receive interfacial forces in x-direction in 2D.
- `MPI_Datatype` *lbfmsg2y*

  MPI derived datatype to send/receive interfacial forces in y-direction in 2D.
- `MPI_Datatype` *lbfmsg3x*

  MPI derived datatype to send/receive interfacial forces in x-direction in 3D.
- `MPI_Datatype` *lbfmsg3y*

  MPI derived datatype to send/receive interfacial forces in y-direction in 3D.

- `MPI_Datatype` *lbfmsg3z*

  MPI derived datatype to send/receive interfacial forces in z-direction in 3D.

- `MPI_Datatype` *lbimsg2x*

  MPI derived datatype to send/receive interfacial normals or density/concentration gradients in x-direction in 2D.

- `MPI_Datatype` *lbimsg2y*

  MPI derived datatype to send/receive interfacial normals or density/concentration gradients in y-direction in 2D.

- `MPI_Datatype` *lbimsg3x*

  MPI derived datatype to send/receive interfacial normals or density/concentration gradients in x-direction in 3D.

- `MPI_Datatype` *lbimsg3y*

  MPI derived datatype to send/receive interfacial normals or density/concentration gradients in y-direction in 3D.

- `MPI_Datatype` *lbimsg3z*

  MPI derived datatype to send/receive interfacial normals or density/concentration gradients in z-direction in 3D.

- `MPI_Datatype` *lbnmsg2x*

  MPI derived datatype to send/receive surface normals in x-direction in 2D.

- `MPI_Datatype` *lbnmsg2y*

  MPI derived datatype to send/receive surface normals in y-direction in 2D.

- `MPI_Datatype` *lbnmsg3x*

  MPI derived datatype to send/receive surface normals in x-direction in 3D.

- `MPI_Datatype` *lbnmsg3y*

  MPI derived datatype to send/receive surface normals in y-direction in 3D.

- `MPI_Datatype` *lbnmsg3z*

  MPI derived datatype to send/receive surface normals in z-direction in 3D.

5.2.4 Class Documentation

struct sNeighbour

Structure for information about communications between processors (e.g. numbers for neighbouring processors, locations in arrays for sending and receiving data to/from neighbours).

Table 5.4: Class Members

unsigned long int	brpos	Starting positions in boundary condition (lbphi) and neighbouring boundary information (lbneigh) arrays for receiving boundary information from neighbouring processors as boundary halos.
unsigned long int	bspos	Starting positions in boundary condition (lbphi) and neighbouring boundary information (lbneigh) arrays for sending boundary information to neighbouring processors.
unsigned long int	frpos	Starting positions in interfacial forces array (lbinterforce) for receiving interfacial forces from neighbouring processors as boundary halos.
unsigned long int	fspos	Starting positions in interfacial forces array (lbinterforce) for sending interfacial forces to neighbouring processors.
unsigned long int	irpos	Starting positions in array for either Lishchuk interfacial normals or Swift free-energy gradients in density and concentration (lbft) for receiving normals or gradients from neighbouring processors as boundary halos.
unsigned long int	ispos	Starting positions in array for either Lishchuk interfacial normals or Swift free-energy gradients in density and concentration (lbft) for sending normals or gradients to neighbouring processors.
unsigned long int	nrpos	Starting positions in surface normals (lbboundnorm) array for receiving surface normals from neighbouring processors as boundary halos.
unsigned long int	nspos	Starting positions in surface normals (lbboundnorm) array for sending surface normals to neighbouring processors.
int	rank	Identifying neighbouring processors by numbers (ranks).
unsigned long int	rpos	Starting positions in distribution function array (lbf) for receiving distribution functions from neighbouring processors as boundary halos.
unsigned long int	spos	Starting positions in distribution function array (lbf) for sending distribution functions to neighbouring processors.

struct sIOGroup

Structure for information about I/O group for writing output files (e.g. processors in group, identity of root processor for gathering and writing data to files, extent of lattice covered by group).

Table 5.5: Class Members

MPI_Comm	cartCommunicator	MPI communicator for entire Cartesian grid of processors in simulation, used to generate I/O communicator by specifying combinations in Cartesian directions (subgrid).
int	cartCoords[3]	Coordinates of current processor within entire grid of processors established to divide up lattice as equally as possible. These coordinates are determined by creating a Cartesian communicator and are checked against those predicted in simulation domain setup.
int	cartEnd[3]	Ending coordinates of the I/O group in terms of numbers of processors (top-right-front corner).
int	cartStart[3]	Starting coordinates of the I/O group in terms of numbers of processors (bottom-left-back corner).
int	gridEndGlobal[3]	Ending coordinates of the section of lattice covered by I/O group (top-right-front corner).
int	gridStartGlobal[3]	Starting coordinates of the section of lattice covered by I/O group (bottom-left-back corner).
int	groupId	Number (identifier) for the current I/O group: this is used in output filenames when more than one file is written per snapshot (i.e. when MPI-IO is not in use).
MPI_Comm	ioCommunicator	MPI communicator for all processors in the I/O group, used for gathering data to be written to output files.
int	rank	Number (rank) of current processor in I/O group: used to identify processor within group. (This value might not be the same as the rank (processor number) for the entire simulation domain.)
int	rootRank	Number (rank) of root processor for the I/O group: data for the group is gathered onto this processor, which then writes it to the output file.
int	rootSize	Total number of root processors in all I/O groups, i.e. the total number of processors involved in writing to output files.
int	size	Total number of processors in the I/O group.
int *	sortpoint	List of one-dimensional grid locations to put each data value in the required sorted order for the output file (sorting by x-coordinate as the fastest changing coordinate, followed by y-coordinate and then z-coordinate) prior to writing the data to the file.
int	subgrid[3]	Flags to indicate in which directions to combine lattice subdomains when forming I/O groups, i.e. creating I/O groups by collecting together processors in each Cartesian direction.

Macro Definition Documentation

omp_get_num_thread

```
#define omp_get_num_thread( )   1
```

Sets the number of OpenMP threads to 1 for the function omp_get_num_thread when DL_MESO_LBE is not compiled with OpenMP.

omp_get_thread_num

```
#define omp_get_thread_num( )    0`
```

Sets the OpenMP thread number to 0 for the function omp_get_thread_num when DL_MESO_LBE is not compiled with OpenMP.

Variable Documentation

dump_handle

```
MPI_File dump_handle
```

MPI file handle to identify simulation restart files (lbout.dump) when writing data using MPI-IO.

ioRootCommunicator

```
MPI_Comm ioRootCommunicator
```

MPI communicator among root processors of I/O groups to coordinate writing to output files using MPI-IO.

lbbmsg2x

```
MPI_Datatype lbbmsg2x
```

MPI derived datatype to send and receive boundary conditions (phase fields) in x-directions for a two-dimensional simulation.

lbbmsg2y

```
MPI_Datatype lbbmsg2y
```

MPI derived datatype to send and receive boundary conditions (phase fields) in y-directions for a two-dimensional simulation.

lbbmsg3x

```
MPI_Datatype lbbmsg3x
```

MPI derived datatype to send and receive boundary conditions (phase fields) in x-directions for a three-dimensional simulation.

lbbmsg3y

```
MPI_Datatype lbbmsg3y
```

MPI derived datatype to send and receive boundary conditions (phase fields) in y-directions for a three-dimensional simulation.

lbbmsg3z

```
MPI_Datatype lbbmsg3z
```

MPI derived datatype to send and receive boundary conditions (phase fields) in z-directions for a three-dimensional simulation.

lbfmsg2x

```
MPI_Datatype lbfmsg2x
```

MPI derived datatype to send and receive interfacial forces in x-directions for a two-dimensional simulation.

lbfmsg2y

```
MPI_Datatype lbfmsg2y
```

MPI derived datatype to send and receive interfacial forces in y-directions for a two-dimensional simulation.

lbfmsg3x

```
MPI_Datatype lbfmsg3x
```

MPI derived datatype to send and receive interfacial forces in x-directions for a three-dimensional simulation.

lbfmsg3y

```
MPI_Datatype lbfmsg3y
```

MPI derived datatype to send and receive interfacial forces in y-directions for a three-dimensional simulation.

lbfmsg3z

```
MPI_Datatype lbfmsg3z
```

MPI derived datatype to send and receive interfacial forces in z-directions for a three-dimensional simulation.

lbimsg2x

```
MPI_Datatype lbimsg2x
```

MPI derived datatype to send and receive interfacial normals or density/concentration gradients in x-directions for a two-dimensional simulation.

lbimsg2y

```
MPI_Datatype lbimsg2y
```

MPI derived datatype to send and receive interfacial normals or density/concentration gradients in y-directions for a two-dimensional simulation.

lbimsg3x

```
MPI_Datatype lbimsg3x
```

MPI derived datatype to send and receive interfacial normals or density/concentration gradients in x-directions for a three-dimensional simulation.

lbimsg3y

```
MPI_Datatype lbimsg3y
```

MPI derived datatype to send and receive interfacial normals or density/concentration gradients in y-directions for a three-dimensional simulation.

lbimsg3z

```
MPI_Datatype lbimsg3z
```

MPI derived datatype to send and receive interfacial normals or density/concentration gradients in z-directions for a three-dimensional simulation.

lbIOGroup

```
extern sIOGroup lbIOGroup
```

Information about I/O group for LBE simulation on current processor (e.g. MPI communicators, processor and lattice extents covered by group).

lbmsg2x

```
MPI_Datatype lbmsg2x
```

MPI derived datatype to send and receive distribution functions in x-directions for a two-dimensional simulation.

lbmsg2y

```
MPI_Datatype lbmsg2y
```

MPI derived datatype to send and receive distribution functions in y-directions for a two-dimensional simulation.

lbmsg3x

```
MPI_Datatype lbmsg3x
```

MPI derived datatype to send and receive distribution functions in x-directions for a three-dimensional simulation.

lbmsg3y

```
MPI_Datatype lbmsg3y
```

MPI derived datatype to send and receive distribution functions in y-directions for a three-dimensional simulation.

lbmsg3z

```
MPI_Datatype lbmsg3z
```

MPI derived datatype to send and receive distribution functions in z-directions for a three-dimensional simulation.

lbnb

```
extern sNeighbour lbnb[6]
```

Neighbour information for LBE simulation on current processor (e.g. neighbouring processor ranks, locations for sending/receiving data).

lbnmsg2x

```
MPI_Datatype lbnmsg2x
```

MPI derived datatype to send and receive surface normals in x-directions for a two-dimensional simulation.

lbnmsg2y

```
MPI_Datatype lbnmsg2y
```

MPI derived datatype to send and receive surface normals in y-directions for a two-dimensional simulation.

lbnmsg3x

```
MPI_Datatype lbnmsg3x
```

MPI derived datatype to send and receive surface normals in x-directions for a three-dimensional simulation.

lbnmsg3y

```
MPI_Datatype lbnmsg3y
```

MPI derived datatype to send and receive surface normals in y-directions for a three-dimensional simulation.

lbnmsg3z

```
MPI_Datatype lbnmsg3z
```

MPI derived datatype to send and receive surface normals in z-directions for a three-dimensional simulation.

output_handle

```
MPI_File output_handle
```

MPI file handle to identify simulation snapshot output files (in XML-based VTK, Legacy VTK or Plot3D formats) when writing data using MPI-IO.

5.3 slbe.hpp

Common variables and arrays when running DL_MESO_LBE in serial.

Required variables and arrays for LBE calculations that are applicable for serial running of DL_MESO_LBE.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <ctime>
#include <cstdio>
#include <cmath>
#include <iomanip>
#include <string>
#include <cstring>
#include <sstream>
#include <vector>
#include <sys/time.h>
#include <sys/stat.h>
#include "lbe.hpp"
#include "lbpBASIC.hpp"
```

(continues on next page)

(continued from previous page)

```
#include "lbpGET.hpp"
#include "lbpIOAGGSER.hpp"
#include "lbpMODEL.hpp"
#include "lbpIO.hpp"
#include "lbpIOPlot3D.hpp"
#include "lbpIOLegacyVTK.hpp"
#include "lbpIOVTK.hpp"
#include "lbpFORCE.hpp"
#include "lbpRHEOLOGY.hpp"
#include "lbpBGK.hpp"
#include "lbpTRT.hpp"
#include "lbpMRT.hpp"
#include "lbpCLBE.hpp"
#include "lbpSUB.hpp"
#include "lbpBOUND.hpp"
#include "lbpBOUNDZouHe.hpp"
#include "lbpBOUNDInamuro.hpp"
#include "lbpBOUNDRegular.hpp"
#include "lbpBOUNDKinetic.hpp"
#include "lbpUSER.hpp"
#include "lbpRUNSER.hpp"
#include "lbpBASIC.cpp"
#include "lbpGET.cpp"
#include "lbpIOAGGSER.cpp"
#include "lbpMODEL.cpp"
#include "lbpIO.cpp"
#include "lbpIOPlot3D.cpp"
#include "lbpIOLegacyVTK.cpp"
#include "lbpIOVTK.cpp"
#include "lbpFORCE.cpp"
#include "lbpRHEOLOGY.cpp"
#include "lbpBGK.cpp"
#include "lbpTRT.cpp"
#include "lbpMRT.cpp"
#include "lbpCLBE.cpp"
#include "lbpSUB.cpp"
#include "lbpBOUND.cpp"
#include "lbpBOUNDZouHe.cpp"
#include "lbpBOUNDInamuro.cpp"
#include "lbpBOUNDRegular.cpp"
#include "lbpBOUNDKinetic.cpp"
#include "lbpUSER.cpp"
#include "lbpRUNSER.cpp"
```

5.3.1 Classes

- *struct sIOGroup*

  Structure for I/O group. (Common to *plbe.hpp*.)

5.3.2 Macros

- `#define` *omp_get_num_thread*

  Setting OpenMP number of threads when compiling DL_MESO_LBE without OpenMP.

- `#define` *omp_get_thread_num*

  Setting OpenMP thread number when compiling DL_MESO_LBE without OpenMP.

5.3.3 Variables

- *sIOGroup lbIOGroup*

  I/O group information for LBE simulation on current processor.

Variable Documentation

lbIOGroup

lbIOGroup slbe.hpp slbe.hpp lbIOGroup `extern sIOGroup lbIOGroup`

Information about I/O group for LBE simulation on current processor (e.g. MPI communicators, processor and lattice extents covered by group).

5.4 plbe.cpp

Main DL_MESO_LBE program for parallel running. (Header file available in plbe.hpp.)

Main DL_MESO_LBE program to run LBE simulations in parallel using Message Passing Interface (MPI) for processor-to-processor communications and optionally using OpenMP multithreading.

5.4.1 Functions

- int *main()*

  Main program for running DL_MESO_LBE in parallel.

5.4.2 Function Documentation

main()

```
int main (int argc, char * argv[])
```

The main content of this source file is to read input files, set up the simulation, select the main simulation loop based on the selected type of interactions between fluids or phases, and close the simulation down after completing all specified timesteps or running out of calculation time.

Parameters

in	argc	Number of command-line arguments included in command to launch DL_MESO_LBE
in	argv	Character array of command-line arguments

5.5 slbe.cpp

Main DL_MESO_LBE program for serial running. (Header file available in slbe.hpp.)

Main DL_MESO_LBE program to run LBE simulations in serial, optionally using OpenMP multithreading.

5.5.1 Functions

- int *main()*

  Main program for running DL_MESO_LBE in serial.

5.5.2 Function Documentation

main()

```
int main (int argc, char * argv[])
```

The main content of this source file is to read input files, set up the simulation, select the main simulation loop based on the selected type of interactions between fluids or phases, and close the simulation down after completing all specified timesteps or running out of calculation time.

Parameters

in	argc	Number of command-line arguments included in command to launch DL_MESO_LBE
in	argv	Character array of command-line arguments

5.6 lbpRUNPAR.cpp and lbpRUNSER.cpp

Modules with main simulation loops for parallel and serial running. (Header files available as lbpRUNPAR.hpp and lbpRUNSER.hpp.)

Subroutines with main simulation loops for LBE simulations in parallel (lbpRUNPAR.cpp) and serial (lbpRUN-PAR.cpp) based on interaction types.

5.6.1 Functions

The functions shown here are those included in lbpRUNPAR.cpp for parallel LBE simulations: similar ones are available in lbpRUNSER.cpp for serial LBE simulations with the initial `f` changed to `fs`, e.g. `fsNoInteract` in place of `fNoInteract`.

- `int` *fNoInteract()*

  Simulation loop for LBE simulations without mesoscopic interactions.

- `int` *fShanChen()*

  Simulation loop for LBE simulations with Shan-Chen pseudopotential interactions.

- `int` *fShanChenQuadratic()*

  Simulation loop for LBE simulations with quadratic Shan-Chen pseudopotential interactions.

- `int` *fLishchuk()*

  Simulation loop for LBE simulations with Lishchuk continuum-based interactions.

- `int` *fLishchukSpencer()*

  Simulation loop for LBE simulations with Lishchuk-Spencer continuum-based interactions.
- `int` *fLishchukSpencerTensor()*

  Simulation loop for LBE simulations with Lishchuk 'Spencer tensor' continuum-based interactions.
- `int` *fLishchukLocal()*

  Simulation loop for LBE simulations with local Lishchuk continuum-based interactions.
- `int` *fSwift()*

  Simulation loop for LBE simulations with Swift free-energy interactions.

5.6.2 Function Documentation

Note that the first call for each function refers to the parallel version of DL_MESO_LBE (as contained in lbpRUNPAR.cpp), while the second refers to the serial version (as contained in lbpRUNSER.cpp). The major differences between the two functions are additional calls for core-to-core communications in the parallel version and alternative interaction calculation routines to include grid points at the system edges for the serial version.

fLishchuk()

```
int fLishchuk ()
int fsLishchuk ()
```

Main simulation loop for a LBE simulation with standard Lishchuk continuum-based interactions between fluids (non-local calculation of interfacial normals, calculation of interfacial forces with interfacial curvatures). By default, the simulation will run for the specified number of timesteps, but if a calculation time is also selected and the running time exceeds this value at the end of an earlier timestep, the simulation will be terminated safely.

fLishchukLocal()

```
int fLishchukLocal ()
int fsLishchukLocal ()
```

Main simulation loop for a LBE simulation with fully local Lishchuk continuum-based interactions between fluids (local calculation of interfacial normals, application of direct interaction forcing term in collisions). By default, the simulation will run for the specified number of timesteps, but if a calculation time is also selected and the running time exceeds this value at the end of an earlier timestep, the simulation will be terminated safely.

fLishchukSpencer()

```
int fLishchukSpencer ()
int fsLishchukSpencer ()
```

Main simulation loop for a parallel LBE simulation with Lishchuk-Spencer continuum-based interactions between fluids (non-local calculation of interfacial normals, calculation of interfacial forces without interfacial curvatures). By default, the simulation will run for the specified number of timesteps, but if a calculation time is also selected and the running time exceeds this value at the end of an earlier timestep, the simulation will be terminated safely.

fLishchukSpencerTensor()

```
int fLishchukSpencerTensor ()
int fsLishchukSpencerTensor ()
```

Main simulation loop for a LBE simulation with Lishchuk ‘Spencer tensor’ continuum-based interactions between fluids (non-local calculation of interfacial normals, application of direct interaction forcing term in collisions). By default, the simulation will run for the specified number of timesteps, but if a calculation time is also selected and the running time exceeds this value at the end of an earlier timestep, the simulation will be terminated safely.

fNoInteract()

```
int fNoInteract ()
int fsNoInteract ()
```

Main simulation loop for a LBE simulation without mesoscopic interactions between fluids or phases. By default, the simulation will run for the specified number of timesteps, but if a calculation time is also selected and the running time exceeds this value at the end of an earlier timestep, the simulation will be terminated safely.

fShanChen()

```
int fShanChen ()
int fsShanChen ()
```

Main simulation loop for a LBE simulation with standard Shan-Chen pseudopotential interactions between fluids or phases. By default, the simulation will run for the specified number of timesteps, but if a calculation time is also selected and the running time exceeds this value at the end of an earlier timestep, the simulation will be terminated safely.

fShanChenQuadratic()

```
int fShanChenQuadratic ()
int fsShanChenQuadratic ()
```

Main simulation loop for a parallel LBE simulation with Shan-Chen pseudopotential interactions between fluids or phases with quadratic pseudopotential terms. By default, the simulation will run for the specified number of timesteps, but if a calculation time is also selected and the running time exceeds this value at the end of an earlier timestep, the simulation will be terminated safely.

fSwift()

```
int fSwift ()
int fsSwift ()
```

Main simulation loop for a LBE simulation with Swift free-energy interactions between fluids or phases. By default, the simulation will run for the specified number of timesteps, but if a calculation time is also selected and the running time exceeds this value at the end of an earlier timestep, the simulation will be terminated safely.

5.7 plbecustom.cpp, slbecustom.cpp and slbecombine.cpp

Customisable DL_MESO_LBE programs for parallel running (plbecustom.cpp), serial running (slbecustom.cpp) and serial running with boundary halos (slbecombine.cpp). The standard headers for parallel and serial running, *plbe.hpp* and *slbe.hpp*, are used for these programs.

Customisable versions of DL_MESO_LBE program to run LBE simulations in parallel using Message Passing Interface (MPI) for processor-to-processor communications (plbecustom.cpp), in serial (slbecustom.cpp) and in serial with boundary halos (slbecombine.cpp), all optionally using OpenMP multithreading.

5.7.1 Functions

- `int` *main()*

  Customisable program for running DL_MESO_LBE.

5.7.2 Function Documentation

main()

```
int main (int argc, char ** argv)
```

The main content of these source files is to set up MPI (if running in parallel), read input files, set up the simulation (including I/O groups of processors to write output files), run through the main simulation loop, and close the simulation down after completing all specified timesteps. These versions of the code are intended for advanced DL_MESO users to select which subroutines are used for force calculations, collisions, output file writing and communications: these options are effectively hard-coded in and cannot be changed in input files. (This approach could be used for running LBE simulations with user-created subroutines.) The source file slbecombine.cpp makes use of boundary halos for serial running (unlike the standard serial code *slbe.cpp*) and includes calls to subroutines for filling grid points in the boundary halo with values for distribution functions, interaction forces etc.

Parameters

in	argc	Number of command-line arguments included in command to launch DL_MESO_LBE
in	argv	Character array of command-line arguments

5.8 lbpBASIC.cpp

5.8.1 Summary

Module with general-purpose functions and subroutines required for LBE simulations, such as sorting numbers, byte-swapping and random number generator. (Header file available as lbpBASIC.hpp.)

5.8.2 Functions

- `template <class T> T` *fCppAbs()*

  Returns absolute value of input value.

- `template <class T> T` *fCppSign()*

  Returns sign of input value.

- `template <class T> T` *fReciprocal()*

  Returns reciprocal while avoiding divisions by zero.

- `template <class T> T` *fEvapLimit()*

  Applies evaporation limit for input value.
- `template <class T> void` *fSwapPair()*

  Swaps a pair of numbers.
- `template <typename T> T` *fStringToNumber()*

  Parses a string and returns numbers contained in it.
- `template <typename T> T` *fCppMax()*

  Finds maximum of a pair of numbers.
- `template <typename T> T` *fCppMin()*

  Finds minimum of a pair of numbers.
- `int` *fGetNumberOrdered()*

  Rearranges two or three integers in descending numerical order.
- `int` *fGetNumberOrderFixed()*

  Rearranges two or three integers in the same numerical order as another set of two or three integers.
- `int` *fBestGrouping()*

  Distributes grid points among processors as evenly as possible.
- `int` *fCppMod()*

  Calculates modulo of a value within a given range.
- `long` *fCppMod()*

  Calculates modulo of a value within a given range.
- `int` *fPrintLine()*

  Prints a line of '-' characters.
- `int` *fPrintDoubleLine()*

  Prints a line of '=' characters.
- `double` *fRandom()*

  Generates a random number between -1 and +1.
- `int` *fBigEndian()*

  Detects endianness of machine running DL_MESO_LBE.
- `void` *fByteSwap()*

  Swaps the byte order of a given value or series of values.
- `double` *fCheckTimeSerial()*

  Outputs time in seconds.
- `string` *fReadString()*

  Outputs a given 'word' in an input string.

5.8.3 Function Documentation

fBestGrouping()

```
int fBestGrouping (int totalgrid,
                   int totalgroup,
                   int & indigrid,
                   int & critigroup)
```

Based on the total number of grid points in a given direction and the total number of processors in the same direction, calculate the numbers of grid points per processor to give as even a distribution as possible. This is achieved by calculating a larger number of grid points and the number of processors to apply this to: the other processors will use the same number less 1.

Parameters

in	total-grid	Total number of grid points in given direction
in	total-group	Total number of processors over which to split the grid points
out	indigrid	Larger number of grid points per processor
out	criti-group	Number of processors to apply larger number of grid points (beyond which, the number of grid points is indigrid-1)

fBigEndian()

```
int fBigEndian ()
```

Check to determine the endianness of the computer running DL_MESO_LBE: returns 1 for big endian, 0 for little endian.

fByteSwap()

```
void fByteSwap (void * data,
                int len,
                int count)
```

Swaps the byte order of a given array of values to convert between endian types. This subroutine is mainly required when writing binary files where a specific endianness is required (e.g. binary VTK files are required in big endian).

Parameters

in,out	data	Values in an array (of any type) for swapping endianness
in	len	Length of a single value in input array in bytes
in	count	Number of values in array

fCheckTimeSerial()

```
double fCheckTimeSerial ()
```

Checks the time since the first call of the function obtained from system clock. This function is used to time DL_MESO_LBE simulations run in serial: there is an alternative function to do the same for parallel calculations (fCheckTimeMPI).

fCppAbs()

```
template <class T>
T fCppAbs (T a)
```

Finds and returns the absolute value of an inputted number by removing its negative sign if less than zero.

Parameters

in	a	Number whose absolute value is to be found.

fCppMax()

```
template <typename T>
T fCppMax (T & a, T & b)
```

Finds the larger of two inputted numbers and returns the larger number's value.

Parameters

in	a	First number to compare.
in	b	Second number to compare.

fCppMin()

```
template <typename T>
T fCppMin (T & a, T & b)
```

Finds the smaller of two inputted numbers and returns the smaller number's value.

Parameters

in	a	First number to compare.
in	b	Second number to compare.

fCppMod()

```
int fCppMod (int a, int b)
long fCppMod (long a, long b)
```

Ensures a given value (a) is within a given range (0 to b-1), so the value immediately beyond the maximum value equals the minimum, and vice versa, i.e. the output equals $a - b$ when $a \geq b$ or $a + b$ when $a < 0$. This function is useful for applying periodic boundary conditions.

Parameters

in,out	a	Value to find modulo of (i.e. output is in range 0 to b-1)
in	b	Range within which value should fit

fCppSign()

```
template <class T>
T fCppSign (T a)
```

Finds and returns the sign of an inputted number (giving +1 for a positive number, -1 for a negative number, or 0 for zero.)

Parameters

in	a	Number whose sign is to be found.

fEvapLimit()

```
template <class T>
T fEvapLimit (T a)
```

Returns an input number if it is larger than the small number set as the evaporation limit: if the number is smaller, zero is returned.

Parameters

in	a	Number to assess compared with evaporation limit.

fGetNumberOrdered()

```
int fGetNumberOrdered (int & iox, int & ioy)
int fGetNumberOrdered (int & iox, int & ioy, int & ioz)
```

Rearranges two or three integers to put them into descending numerical order, i.e. largest to smallest.

Parameters

in,out	&iox	First integer
in,out	&ioy	Second integer
in,out	&ioz	Third integer

fGetNumberOrderFixed()

```
int fGetNumberOrderFixed (int & iox, int & ioy, int & ioz, int ix, int iy, int iz)
int fGetNumberOrderFixed (int & iox, int & ioy, int ix, int iy)
```

Rearranges two or three integers to put them into the same numerical order as another set of two or three integers.

Parameters

in,out	&iox	First integer to sort
in,out	&ioy	Second integer to sort
in,out	&ioz	Third integer to sort
in	ix	First integer as basis of sorting
in	iy	Second integer as basis of sorting
in	iz	Third integer as basis of sorting

fPrintDoubleLine()

```
int fPrintDoubleLine ( )
```

Prints a line of 76 '=' characters to standard output as part of a DL_MESO_LBE run.

fPrintLine()

```
int fPrintLine ( )
```

Prints a line of 76 '-' characters to standard output as part of a DL_MESO_LBE run.

fRandom()

```
double fRandom ( )
```

Applies a single linear congruential random number generator:

$$u_n = (au_{n-1} + c) \pmod m$$

to generate a random number between -1 and +1 for initailising LBE simulations with random noise in fluid densities. This random number generator applies a seed based on the processor number (rank) the first time this function is called, so different results will be obtained when using different numbers of processors.

fReadString()

```
string fReadString (string line, int i)
```

Reads a given string delimited by spaces and outputs the 'i'-th word from that string: this function is used when reading input files.

Parameters

in	line	String to be read
in	i	Number of word in input string to output

fReciprocal()

```
template <class T>
T fReciprocal (T a)
```

Finds and returns the reciprocal of an inputted number if the number is not equal to zero: if the number is zero, the function returns zero to avoid divisions by zero.

Parameters

in	a	Number whose reciprocal is to be found.

fStringToNumber()

```
template <typename T>
T fStringToNumber (const string & text)
```

Takes a string as input and returns any numbers found inside: if no number can be found, this function returns a value of zero.

Parameters

in	text	String to parse and find numbers.

fSwapPair()

```
template <class T>
void fSwapPair (T & a, T & b)
```

Takes a pair of numbers and swaps their values, each returning the other's original value.

Parameters

in	a	First number to swap.
in	b	Second number to swap.

5.9 lbpGET.cpp

Module with routines to calculate fluid, solute and temperature properties at lattice points. (Header file available as lbpGET.hpp.)

Functions and subroutines to calculate memory locations for grid points, find fluid velocities, densities and mass fractions, solute concentrations and temperature, and overall fluid masses and momentum for subdomain. Many of these are used during LBE calculations to e.g. determine local equilibrium distribution functions, while others are used in output files and to report on the progress of a LBE simulation.

5.9.1 Functions

- `long` *fGetNodePosi()*

  Calculates the position of a grid point in a one-dimensional array from its two- or three-dimensional Cartesian coordinates.

- `int` *fGetCoord()*

  Calculates the two- or three-dimensional Cartesian coordinates of a grid point from its one-dimensional array position.

  Calculates the two-dimensional Cartesian coordinates of a grid point from its one-dimensional array position.

- `double` *fGetOneMassSite()*

  Calculates the density of a single fluid at a grid point using distribution functions.

- `int` *fGetAllMassSite()*

  Calculates the densities of all fluids at a grid point using distribution functions.

- `double` *fGetTotMassSite()*

  Calculates the total mass density of all fluids at a grid point using distribution functions.

- `int` *fGetAllConcSite()*

  Calculates the concentrations of all solutes at a grid point using distribution functions.

- `double` *fGetOneMassDomain()*

  Calculates the total mass of a specific fluid in the simulation subdomain.

- `double` *fGetOneMassSwiftDomain()*

  Calculates the total mass of a specific fluid in the simulation subdomain when using Swift free-energy interactions.

- `double` *fGetTotMassDomain()*

  Calculates the total mass of all fluids in the simulation subdomain.

- `double` *fGetFracSite()*

  Calculates the mass fraction of a single fluid at a grid point using distribution functions.

- `double` *fGetFracSwiftSite()*

  Calculates the mass fraction of a single fluid at a grid point using distribution functions when using two-fluid Swift free-energy interactions.

- `int` *fGetOneSpeedSite()*

  Calculates the macroscopic speed of a specific compressible fluid at a grid point using distribution functions.

- `int` *fGetOneSpeedIncomSite()*

  Calculates the macroscopic speed of a specific incompressible fluid at a grid point using distribution functions.

- `int` *fGetOneMomentSite()*

  Calculates the macroscopic momentum of a specific fluid at a grid point using distribution functions.

- `int` *fGetTotMomentSite()*

  Calculates the macroscopic momentum of all fluids at a grid point using distribution functions.

- `int` *fGetTotMomentDomain()*

  Calculates the total momentum of all fluids in the simulation subdomain.

- `int` *fGetTotMomentSwiftDomain()*

  Calculates the total momentum of all fluids in the simulation subdomain when using Swift free-energy interactions.

- `int` *fGetSpeedSite()*

  Calculates the macroscopic velocity of all compressible fluids at a grid point using distribution functions.

- `int` *fGetSpeedAllMassSite()*

  Calculates the macroscopic velocity and densities of all compressible fluids at a grid point using distribution functions.

- `int` *fGetSpeedIncomSite()*

  Calculates the macroscopic velocity of all incompressible fluids at a grid point using distribution functions.

- `int` *fGetSpeedIncomAllMassSite()*

  Calculates the macroscopic velocity and variable densities of all incompressible fluids at a grid point using distribution functions.

- `int` *fGetSpeedShanChenSite()*

  Calculates the macroscopic velocity of all compressible fluids at a grid point using distribution functions when using Shan-Chen interactions.

- `int` *fGetSpeedShanChenAllMassSite()*

  Calculates the macroscopic velocity and densities of all compressible fluids at a grid point using distribution functions when using Shan-Chen interactions.

- `int` *fGetSpeedShanChenIncomSite()*

  Calculates the macroscopic velocity of all incompressible fluids at a grid point using distribution functions when using Shan-Chen interactions.

- `int` *fGetSpeedShanChenIncomAllMassSite()*

  Calculates the macroscopic velocity and variable densities of all incompressible fluids at a grid point using distribution functions when using Shan-Chen interactions.

- `float` *fGetOneDirecSpeedSite()*

  Calculates component of compressible fluid velocity at specified grid point.

- `float` *fGetOneDirecSpeedIncomSite()*

  Calculates component of incompressible fluid velocity at specified grid point.

- `float` *fGetOneDirecSpeedSwiftSite()*

  Calculates component of compressible fluid velocity at specified grid point when using Swift free-energy interactions.

- `double` *fGetOneConcSite()*

  Calculates the concentration of a single solute at a grid point using distribution functions.

- `double` *fGetTemperatureSite()*

  Calculates the temperature at a grid point using distribution functions.

5.9.2 Function Documentation

fGetAllConcSite()

```
int fGetAllConcSite (double * rho, double * startpos)
```

Returns the concentrations of all solutes at a given lattice point by summing up solute distribution functions, i.e.

$$c^a = \sum_i g_i^a$$

Since the distribution functions for each grid point are sorted by fluid, solutes and temperature field and then by lattice link, this subroutine will only give the correct densities if the starting position for the pointer is g_0 for solute 0.

Parameters

out	rho	Solute concentrations at given lattice point
in	startpos	Pointer for distribution function of solute 0 in link 0.

fGetAllMassSite()

```
int fGetAllMassSite (double * rho, double * startpos)
int fGetAllMassSite (double * rho, int xpos, int ypos, int zpos)
int fGetAllMassSite (double * rho, long tpos)
```

Returns the densities of all fluids at a given lattice point by summing up distribution functions, i.e.

$$\rho^a = \sum_i f_i^a$$

Three different interfaces for the function are available with different inputs to specify the lattice point and/or the starting distribution function. Since the distribution functions for each grid point are sorted by fluid (plus solutes and temperature field) and then by lattice link, this subroutine will only give the correct densities if the starting position for the pointer is f_0 for fluid 0.

Parameters

out	rho	Fluid densities at given lattice point
in	startpos	Pointer for distribution function of fluid 0 in link 0.
in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)
in	tpos	Position of lattice site in one-dimensional form

fGetCoord()

```
int fGetCoord (long tpos, int & xpos, int & ypos)
int fGetCoord (long tpos, int & xpos, int & ypos, int & zpos)
```

Depending on which interface is used, returns the two- or three-dimensional Cartesian coordinates of a lattice point specified by a one-dimensional array position.

Parameters

in	tpos	Position of lattice site in one-dimensional form
out	xpos	Coordinate of lattice point (x-component)
out	ypos	Coordinate of lattice point (y-component)
out	zpos	Coordinate of lattice point (z-component)

fGetFracSite()

```
double fGetFracSite (int fpos, double * startpos)
double fGetFracSite (int fpos, int xpos, int ypos, int zpos)
double fGetFracSite (int fpos, long tpos)
```

Returns the mass fraction of a single fluid at a given lattice point by summing up distribution functions, i.e.

$$\frac{\rho^a}{\rho} = \frac{\sum_i f_i^a}{\sum_{i,a} f_i^a}$$

Three different interfaces for the function are available with different inputs to specify the lattice point and/or the starting distribution function. Since the distribution functions for each grid point are sorted by fluid (plus solutes and temperature field) and then by lattice link, this function will only give the correct mass fraction if the starting position for the pointer is f_0 for fluid 0. If the total mass of all fluids is less than the evaporation limit (a very small number, set to 10^{-8} by default), the returned mass fraction will be zero.

Parameters

in	fpos	Number of fluid whose mass fraction is to be determined at given lattice point
in	startpos	Pointer for distribution function of fluid 0 in link 0
in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)
in	tpos	Position of lattice site in one-dimensional form

fGetFracSwiftSite()

```
double fGetFracSwiftSite (int fpos, double * startpos)                double
fGetFracSwiftSite (int fpos, int xpos, int ypos, int zpos)           double
fGetFracSwiftSite (int fpos, long tpos)
```

Returns the mass fraction of a single fluid at a given lattice point when using two-fluid Swift free-energy interactions. This function uses fluid concentrations to determine the mass fractions, i.e.

$$\frac{\rho^{0,1}}{\rho} = \frac{1}{2}(1 \pm \phi)$$

Three different interfaces for the function are available with different inputs to specify the lattice point and/or the starting distribution function. Since the distribution functions for each grid point are sorted by fluid density and concentration (plus solutes and temperature field) and then by lattice link, this function will only give the correct mass fraction if the starting position for the pointer is f_0 for fluid densities.

Parameters

in	fpos	Number of fluid whose mass fraction is to be determined at given lattice point
in	startpos	Pointer for distribution function of fluid density in link 0
in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)
in	tpos	Position of lattice site in one-dimensional form

fGetNodePosi()

```
inline long fGetNodePosi (int xpos, int ypos)
inline long fGetNodePosi (int xpos, int ypos, int zpos)
```

Returns a one-dimensional position in arrays for e.g. distribution functions based on the Cartesian coordinates for a two- or three-dimensional simulation: this value follows the standard data structure for C++ (i.e. row-major) with the z-component as the fastest changing coordinate, followed by the y-component and then the x-component.

Parameters

in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)

fGetOneConcSite()

```
double fGetOneConcSite (int cpos, int xpos, int ypos, int zpos)
double fGetOneConcSite (int cpos, long tpos)
```

Returns the concentration of a single solute at a given lattice point by summing up solute distribution functions, i.e.

$$c^a = \sum_i g_i^a$$

The solute and three-dimensional Cartesian coordinates or one-dimensional grid position are inputs for this function, which point to the required starting distribution function for the given solute (g_0).

Parameters

in	cpos	Number of solute whose concentration is to be determined at given lattice point
in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)
in	tpos	Position of lattice site in one-dimensional form

fGetOneDirecSpeedIncomSite()

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

Returns the specified component of velocity for all incompressible fluids at a given lattice point by summing moments of distribution functions, i.e.

$$u_\alpha = \frac{sum_{i,a} f_i^a e_{i,\alpha}}{\sum_a \rho_0^a}.$$

where ρ_0^a is the constant density for fluid a. The result is output as a single-precision float (real) number: this function is intended for obtaining fluid velocities to write to output files. Three different interfaces for the function are available with different inputs to specify the lattice point and/or the starting distribution function. Since the distribution functions for each grid point are sorted by fluid (plus solutes and temperature field) and then by lattice link, this function will only give the correct velocity and densities if the starting position for the pointer is f_0 for fluid 0.

Parameters

in	dire	Component of velocity to output (0 = x, 1 = y, 2 = z)
in	startpos	Pointer for distribution function of fluid 0 in link 0.
in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)
in	tpos	Position of lattice site in one-dimensional form

fGetOneDirecSpeedSite()

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

Returns the specified component of velocity for all compressible fluids at a given lattice point by summing moments of distribution functions, i.e.

$$u_\alpha = \frac{sum_{i,a} f_i^a e_{i,\alpha}}{\sum_{i,a} f_i^a}.$$

The result is output as a single-precision float (real) number: this function is intended for obtaining fluid velocities to write to output files. Three different interfaces for the function are available with different inputs to specify the lattice point and/or the starting distribution function. Since the distribution functions for each grid point are sorted by fluid (plus solutes and temperature field) and then by lattice link, this function will only give the correct velocity and densities if the starting position for the pointer is f_0 for fluid 0.

Parameters

in	dire	Component of velocity to output (0 = x, 1 = y, 2 = z)
in	startpos	Pointer for distribution function of fluid 0 in link 0.
in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)
in	tpos	Position of lattice site in one-dimensional form

fGetOneDirecSpeedSwiftSite()

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

Returns the specified component of velocity for all compressible fluids at a given lattice point when using Swift free-energy interactions by summing moments of density distribution functions, i.e.

$$u_\alpha = \frac{sum_i f_i e_{i,\alpha}}{\sum_i f_i}.$$

The result is output as a single-precision float (real) number: this function is intended for obtaining fluid velocities to write to output files. Three different interfaces for the function are available with different inputs to specify the lattice point and/or the starting distribution function. Since the distribution functions for each grid point are sorted by fluid (plus solutes and temperature field) and then by lattice link, this function will only give the correct velocity and densities if the starting position for the pointer is f_0 for the density distribution functions.

Parameters

in	dire	Component of velocity to output (0 = x, 1 = y, 2 = z)
in	startpos	Pointer for distribution function of fluid 0 in link 0.
in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)
in	tpos	Position of lattice site in one-dimensional form

fGetOneMassDomain()

```
double fGetOneMassDomain (int fpos)
```

Returns the total mass of a specified fluid in the current processor's simulation subdomain (i.e. its section of the lattice). This function excludes all boundary lattice sites, including those indicating boundary halos for communications between processors (which avoids double counting if summed up later).

Parameters

in	fpos	Number of fluid whose total mass is to be found

fGetOneMassSite()

```
double fGetOneMassSite (double * startpos)
double fGetOneMassSite (int fpos, int xpos, int ypos, int zpos)
double fGetOneMassSite (int fpos, long tpos)
```

Returns the density of a single fluid at a given lattice point by summing up distribution functions, i.e.

$$\rho^a = \sum_i f_i^a$$

Three different interfaces for the function are available with different inputs to specify the fluid, lattice point and/or the starting distribution function. For each grid point, the distribution functions are sorted by fluid (plus solutes and temperature field) and then by lattice link, so the starting position for the pointer will be f_0 for the given fluid.

Parameters

in	startpos	Pointer for distribution function of specified fluid in link 0.
in	fpos	Number of fluid whose density is to be found
in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)
in	tpos	Position of lattice site in one-dimensional form

fGetOneMassSwiftDomain()

```
double fGetOneMassSwiftDomain (int fpos)
```

Returns the total mass of a specified fluid in the current processor's simulation subdomain (i.e. its section of the lattice) when using Swift free-energy interactions. This function returns the actual mass for the specified fluid from the total masses and the fluid concentrations ϕ at each grid point, i.e.

$$\rho^{0,1} = \frac{1}{2}\rho \left(1 \pm \phi\right)$$

This function excludes all boundary lattice sites, including those indicating boundary halos for communications between processors (which avoids double counting if summed up later).

Parameters

in	fpos	Number of fluid whose total mass in the subdomain is to be found

fGetOneMomentSite()

```
int fGetOneMomentSite (double * speed, double * startpos)
int fGetOneMomentSite (double * speed, int fpos, int xpos, int ypos, int zpos)
int fGetOneMomentSite (double * speed, int fpos, long tpos)
```

Returns the momentum (product of density and velocity) of a single fluid at a given lattice point by summing up distribution functions, i.e.

$$\vec{p}^a = \sum_i f_i^a \hat{e}_i$$

This subroutine will work for both mildly compressible and fully incompressible fluids. Three different interfaces for the function are available with different inputs to specify the fluid, lattice point and/or the starting distribution function. For each grid point, the distribution functions are sorted by fluid (plus solutes and temperature field) and then by lattice link, so the starting position for the pointer will be f_0 for the given fluid.

Parameters

out	speed	Momentum of specified fluid at given lattice point
in	startpos	Pointer for distribution function of specified fluid in link 0
in	fpos	Number of fluid whose momentum is to be determined at given lattice point
in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)
in	tpos	Position of lattice site in one-dimensional form

fGetOneSpeedIncomSite()

```
int fGetOneSpeedIncomSite (double * speed, double * startpos, double rho0)
```

Returns the velocity of a single incompressible fluid at a given lattice point by summing up distribution functions, i.e.

$$\vec{u}^a = \frac{\sum_i f_i^a \hat{e}_i}{\rho_0^a}$$

where ρ_0^a is the constant density for fluid a. Three different interfaces for the function are available with different inputs to specify the fluid (or its constant density), lattice point and/or the starting distribution function. For each grid point, the distribution functions are sorted by fluid (plus solutes and temperature field) and then by lattice link, so the starting position for the pointer will be f_0 for the given fluid.

Parameters

out	speed	Velocity of specified fluid at given lattice point
in	startpos	Pointer for distribution function of specified fluid in link 0
in	rho0	Constant density for specified fluid
in	fpos	Number of fluid whose velocity is to be determined at given lattice point
in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)
in	tpos	Position of lattice site in one-dimensional form

fGetOneSpeedSite()

```
int fGetOneSpeedSite (double * speed, double * startpos)
int fGetOneSpeedSite (double * speed, int fpos, int xpos, int ypos, int zpos)
int fGetOneSpeedSite (double * speed, int fpos, long tpos)
```

Returns the velocity of a single compressible fluid at a given lattice point by summing up distribution functions, i.e.

$$\vec{u}^a = \frac{\sum_i f_i^a \hat{e}_i}{\sum_i f_i^a}$$

Three different interfaces for the function are available with different inputs to specify the fluid, lattice point and/or the starting distribution function. For each grid point, the distribution functions are sorted by fluid (plus solutes and temperature field) and then by lattice link, so the starting position for the pointer will be f_0 for the given fluid.

Parameters

out	speed	Velocity of specified fluid at given lattice point
in	startpos	Pointer for distribution function of specified fluid in link 0
in	fpos	Number of fluid whose velocity is to be determined at given lattice point
in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)
in	tpos	Position of lattice site in one-dimensional form

fGetSpeedAllMassSite()

```
int fGetSpeedAllMassSite (double * speed,
                          double * rho,
                          double * startpos)
```

Returns the velocity and densities of all compressible fluids at a given lattice point by summing up distribution functions, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}$$

and

$$\rho^a = \sum_i f_i^a.$$

Since the distribution functions for each grid point are sorted by fluid (plus solutes and temperature field) and then by lattice link, this function will only give the correct velocity and densities if the starting position for the pointer is f_0 for fluid 0.

Parameters

out	speed	Velocity of all fluids at given lattice point
out	rho	Fluid densities at given lattice point
in	startpos	Pointer for distribution function of fluid 0 in link 0.

fGetSpeedIncomAllMassSite()

```
int fGetSpeedIncomAllMassSite (double * speed,
                               double * rho,
                               double * startpos)
```

Returns the velocity and variable densities of all incompressible fluids at a given lattice point by summing up distribution functions, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_a \rho_0^a}$$

and

$$\rho^a = \sum_i f_i^a.$$

where ρ_0^a is the constant density for fluid a. Since the distribution functions for each grid point are sorted by fluid (plus solutes and temperature field) and then by lattice link, this function will only give the correct velocity and densities if the starting position for the pointer is f_0 for fluid 0.

Parameters

out	speed	Velocity of all fluids at given lattice point
out	rho	Variable fluid densities at given lattice point
in	startpos	Pointer for distribution function of fluid 0 in link 0.

fGetSpeedIncomSite()

```
int fGetSpeedIncomSite (double * speed, double * startpos)
int fGetSpeedIncomSite (double * speed, int xpos, int ypos, int zpos)
int fGetSpeedIncomSite (double * speed, long tpos)
```

Returns the velocity of all incompressible fluids at a given lattice point by summing up distribution functions, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_a \rho_0^a}$$

where ρ_0^a is the constant density for fluid a. Three different interfaces for the function are available with different inputs to specify the lattice point and/or the starting distribution function. Since the distribution functions for each grid point are sorted by fluid (plus solutes and temperature field) and then by lattice link, this function will only give the correct velocity if the starting position for the pointer is f_0 for fluid 0.

Parameters

out	speed	Velocity of all fluids at given lattice point
in	startpos	Pointer for distribution function of fluid 0 in link 0.
in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)
in	tpos	Position of lattice site in one-dimensional form

fGetSpeedShanChenAllMassSite()

```
int fGetSpeedShanChenAllMassSite (double * speed,
                                  double * rho,
                                  double * startpos,
                                  double * omega)
```

Returns the velocity and densities of all compressible fluids at a given lattice point when using Shan-Chen pseudopotential interactions by summing up distribution functions (weighting with relaxation frequencies for the velocity [91]), i.e.

$$\vec{u} = \frac{\sum_{i,a} \omega^a f_i^a \hat{e}_i}{\sum_{i,a} \omega^a f_i^a}$$

and

$$\rho^a = \sum_i f_i^a.$$

where :math:` omega^a = tau_{f,a}^{-1}` is the relaxation frequency of fluid a. Since the distribution functions for each grid point are sorted by fluid (plus solutes and temperature field) and then by lattice link, this function will only give the correct velocity and densities if the starting position for the pointer is f_0 for fluid 0.

Parameters

out	speed	Velocity of all fluids at given lattice point
out	rho	Fluid densities at given lattice point
in	startpos	Pointer for distribution function of fluid 0 in link 0
in	omega	Relaxation frequencies for all fluids at given lattice point

fGetSpeedShanChenIncomAllMassSite()

```
int fGetSpeedShanChenIncomAllMassSite (double * speed,
                                       double * rho,
                                       double * startpos,
                                       double * omega)
```

Returns the velocity and variable densities of all icompressible fluids at a given lattice point when using Shan-Chen pseudopotential interactions by summing up distribution functions (weighting with relaxation frequencies for the velocity [91]), i.e.

$$\vec{u} = \frac{\sum_{i,a} \omega^a f_i^a \hat{e}_i}{\sum_{a} \omega^a \rho_0^a}$$

and

$$\rho^a = \sum_i f_i^a.$$

where :math:` omega^a = tau_{f,a}^{-1}` is the relaxation frequency of fluid a and ρ_0^a is the constant density for the same fluid. Since the distribution functions for each grid point are sorted by fluid (plus solutes and temperature field) and then by lattice link, this function will only give the correct velocity and densities if the starting position for the pointer is f_0 for fluid 0.

Parameters

out	speed	Velocity of all fluids at given lattice point
out	rho	Fluid densities at given lattice point
in	startpos	Pointer for distribution function of fluid 0 in link 0.
in	omega	Relaxation frequencies for all fluids at given lattice point

fGetSpeedShanChenIncomSite()

```
int fGetSpeedShanChenIncomSite (double * speed, double * startpos, double * omega)
int fGetSpeedShanChenIncomSite (double * speed, int xpos, int ypos, int zpos)
int fGetSpeedShanChenIncomSite (double * speed, long tpos)
```

Returns the velocity of all incompressible fluids at a given lattice point when using Shan-Chen pseudopotential interactions by summing up distribution functions weighted by relaxation frequencies [91], i.e.

$$\vec{u} = \frac{\sum_{i,a} \omega^a f_i^a \hat{e}_i}{\sum_a \omega^a \rho_0^a}$$

where $\omega^a = \tau_{f,a}^{-1}$ is the relaxation frequency of fluid a and ρ_0^a is the constant density for the same fluid. Three different interfaces for the function are available with different inputs to specify the lattice point and/or the starting distribution function (and the relaxation frequencies if using the latter). Since the distribution functions for each grid point are sorted by fluid (plus solutes and temperature field) and then by lattice link, this function will only give the correct velocity if the starting position for the pointer is f_0 for fluid 0.

Parameters

out	speed	Velocity of all fluids at given lattice point
in	startpos	Pointer for distribution function of fluid 0 in link 0
in	omega	Relaxation frequencies for all fluids at given lattice point
in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)
in	tpos	Position of lattice site in one-dimensional form

fGetSpeedShanChenSite()

```
int fGetSpeedShanChenSite (double * speed, double * startpos, double * omega)
int fGetSpeedShanChenSite (double * speed, int xpos, int ypos, int zpos)
int fGetSpeedShanChenSite (double * speed, long tpos)
```

Returns the velocity of all compressible fluids at a given lattice point when using Shan-Chen pseudopotential interactions by summing up distribution functions weighted by relaxation frequencies [91], i.e.

$$\vec{u} = \frac{\sum_{i,a} \omega^a f_i^a \hat{e}_i}{\sum_{i,a} \omega^a f_i^a}$$

where $\omega^a = \tau_{f,a}^{-1}$ is the relaxation frequency of fluid a. Three different interfaces for the function are available with different inputs to specify the lattice point and/or the starting distribution function (and the relaxation frequencies if using the latter). Since the distribution functions for each grid point are sorted by fluid (plus solutes and temperature field) and then by lattice link, this function will only give the correct velocity if the starting position for the pointer is f_0 for fluid 0.

Parameters

out	speed	Velocity of all fluids at given lattice point
in	startpos	Pointer for distribution function of fluid 0 in link 0
in	omega	Relaxation frequencies for all fluids at given lattice point
in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)
in	tpos	Position of lattice site in one-dimensional form

fGetSpeedSite()

```
int fGetSpeedSite (double * speed, double * startpos)
int fGetSpeedSite (double * speed, int xpos, int ypos, int zpos)
int fGetSpeedSite (double * speed, long tpos)
```

Returns the velocity of all compressible fluids at a given lattice point by summing up distribution functions, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}$$

Three different interfaces for the function are available with different inputs to specify the lattice point and/or the starting distribution function. Since the distribution functions for each grid point are sorted by fluid (plus solutes and temperature field) and then by lattice link, this function will only give the correct velocity if the starting position for the pointer is f_0 for fluid 0.

Parameters

out	speed	Velocity of all fluids at given lattice point
in	startpos	Pointer for distribution function of fluid 0 in link 0.
in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)
in	tpos	Position of lattice site in one-dimensional form

fGetTemperatureSite()

```
double fGetTemperatureSite (long tpos)
double fGetTemperatureSite (long xpos, long ypos, long zpos)
```

Returns the temperature at a given lattice point by summing up temperature distribution functions, i.e.

$$T = \sum_i h_i$$

The one-dimensional grid point or three-dimensional Cartesian coordinates are inputs for this function, which point to the required starting distribution function for the temperature field (h_0).

Parameters

in	tpos	Position of lattice site in one-dimensional form
in	xpos	Coordinate of lattice point (x-component)
in	ypos	Coordinate of lattice point (y-component)
in	zpos	Coordinate of lattice point (z-component)

fGetTotMassDomain()

```
double fGetTotMassDomain ()
```

Returns the total mass of all fluids in the current processor's simulation subdomain (i.e. its section of the lattice). This function excludes all boundary lattice sites, including those indicating boundary halos for communications between processors (which avoids double counting if summed up later).

fGetTotMassSite()

```
double fGetTotMassSite (double * startpos)
double fGetTotMassSite (long tpos)
```

Returns the total density of all fluids at a given lattice point by summing up distribution functions, i.e.

$$\rho = \sum_{i,a} f_i^a$$

Either the distribution function pointer or the one-dimensional grid point is the input for this function. Since the distribution functions for each grid point are sorted by fluid (plus solutes and temperature field) and then by lattice link, this function will only give the correct total density if the starting position for the pointer is f_0 for fluid 0.

Parameters

in	startpos	Pointer for distribution function of fluid 0 in link 0.
in	tpos	Position of lattice site in one-dimensional form

fGetTotMomentDomain()

```
int fGetTotMomentDomain (double * momentum)
```

Returns the total momentum of all fluids in the current processor's simulation subdomain (i.e. its section of the lattice). This function excludes all boundary lattice sites, including those indicating boundary halos for communications between processors (which avoids double counting if summed up later).

Parameters

momentum	Total momentum of fluids at all fluid sites in simulation subdomain

fGetTotMomentSite()

```
int fGetTotMomentSite (double * momentum, double * startpos)
```

Returns the momentum (product of density and velocity) of all fluids at a given lattice point by summing up distribution functions, i.e.

$$\vec{p} = \sum_{i,a} f_i^a \hat{e}_i$$

This subroutine will work for both mildly compressible and fully incompressible fluids. Since the distribution functions for each grid point are sorted by fluid (plus solutes and temperature field) and then by lattice link, this function will only give the correct momentum if the starting position for the pointer is f_0 for fluid 0.

Parameters

out	momentum	Momentum of all fluids at given lattice point
in	startpos	Pointer for distribution function of specified fluid in link 0.

fGetTotMomentSwiftDomain()

```
int fGetTotMomentSwiftDomain (double * momentum)
```

Returns the total momentum of all fluids in the current processor's simulation subdomain (i.e. its section of the lattice) when using Swift free-energy interactions. This function only uses sums moments of the distribution functions for fluid density - concentration distribution functions used in two-fluid models do not contribute to this property - and excludes all boundary lattice sites, including those indicating boundary halos for communications between processors (which avoids double counting if summed up later).

5.10 lbpMODEL.cpp

Module to set up lattice-based weighting functions, link vectors and multiple relaxation time (MRT) transformation matrices. (Header file available as lbpMODEL.hpp.)

5.10.1 Functions

- `int` *D2Q9()*

  Sets up lattice-based parameters, vectors and transformation matrices for D2Q9 lattice scheme.
- `int` *D3Q15()*

  Sets up lattice-based parameters, vectors and transformation matrices for D3Q15 lattice scheme.
- `int` *D3Q19()*

  Sets up lattice-based parameters, vectors and transformation matrices for D3Q19 lattice scheme.
- `int` *D3Q27()*

  Sets up lattice-based parameters, vectors and transformation matrices for D3Q27 lattice scheme.

5.10.2 Detailed Description

Subroutines to set up speeds of sound c_s, weighting functions for local equilibrium distribution functions w_i (including those for Swift free-energy interactions if applicable), vectors for links between lattice points $\hat{e}_i$, products of weighting functions and link vectors for gradient stencils used in Shan-Chen pseudopotential and Lishchuk continuum-based interactions, first-order and second-order gradient stencils for Swift free-energy interactions (if applicable), conjugate link identifiers, and the multiple relaxation time (MRT) transformation matrix $\mathbf{T}$ and its inverse $\mathbf{T}^{-1}$. (Transformation matrices for cascaded LBE collisions are not set up in these routines as these are dependent on fluid velocities at each lattice point.)

5.10.3 Function Documentation

D2Q9()

```
int D2Q9 ()
```

Sets values for speeds of sound, local equilbrium distribution function weighting parameters, link vectors, gradient stencils, conjugate links and MRT transformation matrices [73] for the D2Q9 (two-dimensional, nine lattice vectors) lattice scheme. If using Swift free-energy interactions, these will include the weighting parameters for the free-energy local equilibrium distribution functions and microcurrent-reducing stencils for first-order and second-order gradients [102].

D3Q15()

```
int D3Q15 ()
```

Sets values for speeds of sound, local equilbrium distribution function weighting parameters, link vectors, gradient stencils, conjugate links and MRT transformation matrices [159] for the D3Q15 (three-dimensional, fifteen lattice vectors) lattice scheme. If using Swift free-energy interactions, these will include the weighting parameters for the free-energy local equilibrium distribution functions and microcurrent-reducing stencils for first-order and second-order gradients [102].

D3Q19()

```
int D3Q19 ()
```

Sets values for speeds of sound, local equilbrium distribution function weighting parameters, link vectors, gradient stencils, conjugate links and MRT transformation matrices [159] for the D3Q19 (three-dimensional, nineteen lattice vectors) lattice scheme. If using Swift free-energy interactions, these will include the weighting parameters for the free-energy local equilibrium distribution functions and microcurrent-reducing stencils for first-order and second-order gradients [102].

D3Q27()

```
int D3Q27 ()
```

Sets values for speeds of sound, local equilbrium distribution function weighting parameters, link vectors, gradient stencils, conjugate links and MRT transformation matrices [134] for the D3Q27 (three-dimensional, twenty-seven lattice vectors) lattice scheme. No Swift free-energy interactions are currently available for this lattice scheme.

5.11 lbpSUB.cpp

Module with important subroutines and functions for LBE calculations. (Header file available as lbpSUB.hpp.)

Subroutines and functions required to carry out key parts of LBE simulations, e.g. propagation, calculate local equilibrium distribution functions, allocate and deallocate arrays.

5.11.1 Functions

- `void` *fWeakMemory()*

  Prints error message due to lack of memory for array allocation.
- `int` *fMemoryAllocation()*

  Allocates memory for LBE calculations.
- `int` *fFreeMemory()*

  Frees allocated memory for LBE calculations.
- `int` *fSetSerialDomain()*

  Determines the domain parameters for the serial LBE calculation.
- `int` *fSetSerialDomainBuffer()*

  Determines the domain parameters for the serial LBE calculation with a boundary halo.
- `int` *fStartDLMESO()*

  Announces start of DL_MESO_LBE calculation.

- `int` *fFinishDLMESO()*

  Announces end of DL_MESO_LBE calculation.

- `int` *fsPrintDomainInfo()*

  Prints information about numbers of threads to standard output.

- `int` *fGetModel()*

  Initialises lattice-related arrays based on lattice scheme in use.

- `int` *fMarkBoundArea3D()*

  Assigns boundary conditions for boundary halos of three-dimensional systems.

- `int` *fMarkBoundArea2D()*

  Assigns boundary conditions for boundary halos of two-dimensional systems.

- `int` *fMarkBoundArea()*

  Assigns boundary conditions for lattice points making up boundary halos.

- `int` *fGetEquilibriumF()*

  Calculates local equilibrium distribution functions for a mildly compressible fluid.

- `int` *fGetEquilibriumFIncom()*

  Calculates local equilibrium distribution functions for a fully incompressible fluid.

- `int` *fGetEquilibriumFSwiftOneFluid()*

  Calculates local equilibrium distribution functions for one mildly compressible fluid with Swift free-energy interactions.

- `int` *fGetEquilibriumFSwiftTwoFluid()*

  Calculates local equilibrium distribution functions for two mildly compressible fluids with Swift free-energy interactions.

- `int` *fGetEquilibriumFCLBED2Q9()*

  Calculates local equilibrium distribution functions for a mildly compressible fluid undergoing CLBE collisions on a D2Q9 lattice.

- `int` *fGetEquilibriumFCLBED3Q19()*

  Calculates local equilibrium distribution functions for a mildly compressible fluid undergoing CLBE collisions on a D3Q19 lattice.

- `int` *fGetEquilibriumFCLBED3Q27()*

  Calculates local equilibrium distribution functions for a mildly compressible fluid undergoing CLBE collisions on a D3Q27 lattice.

- `double` *fGetBulkPressureSwift()*

  Calculates buik pressure for Swift free-energy interactions.

- `double` *fGetPotentialSwift()*

  Calculates potential for two-fluid Swift free-energy interactions.

- `double` *fGetLambdaSwift()*

  Calculates Galilean invariance correction factor for fluids undergoing Swift free-energy interactions.

- `int` *fGetEquilibriumC()*

  Calculates local equilibrium distribution functions for a solute.

- `int` *fGetEquilibriumT()*

  Calculates local equilibrium distribution functions for temperature.

- `int` *fInitializeSystem()*

  Initialises distribution functions for LBE calculation.
- `int` *fPropagationTwoLattice()*

  Carries out propagation stage using a two-lattice algorithm.
- `int` *fPropagationSwap()*

  Carries out propagation stage using a swap-based algorithm.
- `int` *fPropagationCombinedSwap()*

  Carries out propagation stage using a combined swap-based algorithm.

5.11.2 Function Documentation

fFinishDLMESO()

```
int fFinishDLMESO ()
```

Prints messages at the end of a DL_MESO_LBE calculation indicating the elapsed calculation time, the efficiency measure (Millions of Lattice Updates Per Second), the finishing time and possible citations for publishing results to standard output.

fFreeMemory()

```
int fFreeMemory ()
```

Deallocates the arrays previously used for a Lattice Boltzmann Equation (LBE) calculation when DL_MESO_LBE closes down.

fGetBulkPressureSwift()

```
double fGetBulkPressureSwift (double rho,
                              double phi,
                              double T)
```

Calculates and returns the bulk pressure at a given lattice point for Swift free-energy interactions, based on the selected equation of state:

- Ideal lattice gas:

$$P_0 = \rho c_s^2$$

- Shan-Chen 1993 model [118]:

$$P_0 = \rho c_s^2 + \frac{1}{2} c_s^2 g \rho_0^2 \left(1 - e^{-\frac{\rho}{\rho_0}}\right)$$

- Shan-Chen 1994 model [119]:

$$P_0 = \rho c_s^2 + \frac{1}{2} c_s^2 g \psi_0^2 e^{-\frac{2\rho_0}{\rho}}$$

- Qian model [106]:

$$P_0 = \rho c_s^2 + \frac{c_s^2 g \rho_0^2 \rho^2}{(\rho_0 + \rho)^2}$$

- Density model:

$$P_0 = \rho c_s^2 + \frac{1}{2} c_s^2 g \rho^2$$

- Ideal gas:

$$P_0 = \rho RT$$

- van der Waals:

$$P_0 = \frac{\rho RT}{1 - b\rho} - a\rho^2$$

- Carnahan-Starling-van der Waals [16]:

$$P_0 = \rho RT \left(\frac{1 + \phi + \phi^2 - \phi^3}{(1-\phi)^3} \right) - a\rho^2$$

- Redlich-Kwong [110]:

$$P_0 = \frac{\rho RT}{1 - b\rho} - \frac{a\rho^2}{\sqrt{T}\,(1 + b\rho)}$$

- Soave-Redlich-Kwong [128]:

$$P_0 = \frac{\rho RT}{1 - b\rho} - \frac{a\alpha\,(T_r, \omega)\,\rho^2}{1 + b\rho}$$

- Peng-Robinson [99]:

$$P_0 = \frac{\rho RT}{1 - b\rho} - \frac{a\alpha\,(T_r, \omega)\,\rho^2}{1 + 2b\rho - b^2\rho^2}$$

- Carnahan-Starling-Redlich-Kwong [16]:

$$P_0 = \rho RT \left(\frac{1 + \phi + \phi^2 - \phi^3}{(1-\phi)^3} \right) - \frac{a\rho^2}{\sqrt{T}\,(1 + b\rho)}$$

where R is the universal gas constant, a and b are species-dependent coefficients, α is a function dependent on the ratio of temperature to critical temperature $T_r = T/T_c$ and acentric factor ω, and $\phi = \frac{b\rho}{4}$ for Carnahan-Starling equations of state. The temperatures used in some equations of state can either be specified system-wide or at each lattice point if heat effects are coupled to fluid flows with an additional lattice.

If two fluids are being simulated, the following additional contribution due to mixing:

$$P_{mix} = a \left(-\frac{1}{2}\phi^2 + \frac{3}{4}\phi^4 \right)$$

is added to the bulk pressure, where a is the parameter used for the potential between the two fluids.

Parameters

in	rho	Fluid density at lattice site ρ
in	phi	Fluid concentration at lattice site (only used for two-fluid interactions) ϕ
in	T	Temperature at lattice site

fGetEquilibriumC()

```
int fGetEquilibriumC (double * feq,
                      double * v,
                      double rho)
```

Calculates the local equilibrium distribution functions for a solute [62]:

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

as required for collisions and system initialisation. This expression is only suitable for square lattices (i.e. those currently implemented in DL_MESO_LBE).

Parameters

out	feq	Local equilibrium distribution functions for solute at given lattice site
in	v	Fluid velocity at lattice site $\vec{u}$
in	rho	Solute concentration at lattice site C

fGetEquilibriumF()

```
int fGetEquilibriumF (double * feq,
                      double * v,
                      double rho)
```

Calculates the local equilibrium distribution functions for a mildly compressible fluid:

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

as required for collisions and system initialisation. This expression is only suitable for square lattices (i.e. those currently implemented in DL_MESO_LBE).

Parameters

out	feq	Local equilibrium distribution functions for given lattice site
in	v	Fluid velocity at lattice site $\vec{u}$
in	rho	Fluid density at lattice site ρ

fGetEquilibriumFCLBED2Q9()

```
int fGetEquilibriumFCLBED2Q9 (double * feq,
                              double * v,
                              double rho)
```

Calculates the local equilibrium distribution functions for a mildly compressible fluid undergoing cascaded LBE (CLBE) collisions on a two-dimenensional D2Q9 lattice. The distribution functions are obtained by an inverse transformation of the local equilibrium central moments:

$$\vec{f}^{eq} = \mathbf{T}^{-1} \mathbf{N}^{-1} \vec{\tilde{M}}^{eq}.$$

These local equilibrium distribution functions are an approximation of the Maxwell-Boltzmann (general) local equilibrium distribution function, which was used to derive the local equilibrium central moments $\vec{\tilde{M}}^{eq}$. Since the CLBE collisions are carried out using central moments, this subroutine is mainly used to initialise simulations using these collisions.

Parameters

out	feq	Local equilibrium distribution functions for given lattice site
in	v	Fluid velocity at lattice site $\vec{u}$
in	rho	Fluid density at lattice site ρ

fGetEquilibriumFCLBED3Q19()

```
int fGetEquilibriumFCLBED3Q19 (double * feq,
                               double * v,
                               double rho)
```

Calculates the local equilibrium distribution functions for a mildly compressible fluid undergoing cascaded LBE (CLBE) collisions on a three-dimenensional D3Q19 lattice. The distribution functions are obtained by an inverse transformation of the local equilibrium central moments:

$$\vec{f}^{eq} = \mathbf{T}^{-1}\mathbf{N}^{-1}\vec{\tilde{M}}^{eq}.$$

These local equilibrium distribution functions are an approximation of the Maxwell-Boltzmann (general) local equilibrium distribution function, which was used to derive the local equilibrium central moments $\vec{\tilde{M}}^{eq}$. Since the CLBE collisions are carried out using central moments, this subroutine is mainly used to initialise simulations using these collisions.

Parameters

out	feq	Local equilibrium distribution functions for given lattice site
in	v	Fluid velocity at lattice site $\vec{u}$
in	rho	Fluid density at lattice site ρ

fGetEquilibriumFCLBED3Q27()

```
int fGetEquilibriumFCLBED3Q27 (double * feq,
                               double * v,
                               double rho)
```

Calculates the local equilibrium distribution functions for a mildly compressible fluid undergoing cascaded LBE (CLBE) collisions on a three-dimenensional D3Q27 lattice. The distribution functions are obtained by an inverse transformation of the local equilibrium central moments:

$$\vec{f}^{eq} = \mathbf{T}^{-1}\mathbf{N}^{-1}\vec{\tilde{M}}^{eq}.$$

These local equilibrium distribution functions are an approximation of the Maxwell-Boltzmann (general) local equilibrium distribution function, which was used to derive the local equilibrium central moments $\vec{\tilde{M}}^{eq}$. Since the CLBE collisions are carried out using central moments, this subroutine is mainly used to initialise simulations using these collisions.

Parameters

out	feq	Local equilibrium distribution functions for given lattice site
in	v	Fluid velocity at lattice site $\vec{u}$
in	rho	Fluid density at lattice site ρ

fGetEquilibriumFIncom()

```
int fGetEquilibriumFIncom (double * feq,
                           double * v,
                           double rho,
                           double rho0)
```

Calculates the local equilibrium distribution functions for a fully incompressible fluid [53]:

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

as required for collisions and system initialisation, which uses a constant density ρ_0 and a variable density ρ as an analogue for pressure. This expression is only suitable for square lattices (i.e. those currently implemented in DL_MESO_LBE).

Parameters

out	feq	Local equilibrium distribution functions for given lattice site
in	v	Fluid velocity at lattice site $\vec{u}$
in	rho	Variable fluid density at lattice site ρ
in	rho0	Constant fluid density at lattice site ρ_0

fGetEquilibriumFSwiftOneFluid()

```
int fGetEquilibriumFSwiftOneFluid (double * feq,
                                   double * v,
                                   double rho,
                                   double p0,
                                   double lambda,
                                   double * delta)
```

Calculates the local equilibrium distribution functions for one mildly compressible fluid with Swift free-energy interactions [136][102]:

$$f_i^{eq} = w_i^{00}\rho + w_i \left[\rho \left\{ \left(\hat{e}_i \cdot \vec{u}\right) + \frac{3}{2}\left(\hat{e}_i \cdot \vec{u}\right)^2 - \frac{1}{2}u^2 \right\} + \lambda \left\{ 3\left(\hat{e}_i \cdot \vec{u}\right)\left(\hat{e}_i \cdot \nabla\rho\right) + \left[\gamma_i\left(\hat{e}_i \cdot \hat{e}_i\right) + \delta_i\right]\left(\vec{u} \cdot \nabla\rho\right) \right\} \right] + w_i^p P_0 - w_i^t \kappa \rho \nabla^2 \rho$$

as required for collisions and system initialisation. This expression is only suitable for square lattices (i.e. those currently implemented in DL_MESO_LBE) apart from D3Q27, for which no free-energy scheme currently exists.

Parameters

out	feq	Local equilibrium distribution functions for given lattice site
in	v	Fluid velocity at lattice site $\vec{u}$
in	rho	Fluid density at lattice site ρ
in	p0	Bulk pressure at lattice site (determined from equation of state) P_0
in	lambda	Correction parameter for Galilean invariance (determined from equation of state) λ
in	delta	First-order and second-order derivatives of density

fGetEquilibriumFSwiftTwoFluid()

```
int fGetEquilibriumFSwiftTwoFluid (double * feq,
                                    double * v,
                                    double rho,
                                    double phi,
                                    double p0,
                                    double pot,
                                    double lambda,
                                    double * delta)
```

Calculates the local equilibrium distribution functions for two mildly compressible fluids with Swift free-energy interactions [135][103] as required for collisions and system initialisation, both for density distribution functions:

$$f_i^{eq} = w_i^{00}\rho + w_i\left[\rho\left\{(\hat{e}_i\cdot\vec{u}) + \frac{3}{2}(\hat{e}_i\cdot\vec{u})^2 - \frac{1}{2}u^2\right\} + \lambda\left\{3(\hat{e}_i\cdot\vec{u})(\hat{e}_i\cdot\nabla\rho) + [\gamma_i(\hat{e}_i\cdot\hat{e}_i)+\delta_i](\vec{u}\cdot\nabla\rho)\right\}\right] + w_i^p P_0 - w_i^t\kappa(\rho\nabla^2$$

and for concentration distribution functions:

$$g_i^{eq} = w_i^{00}\phi + w_i\phi\left\{(\hat{e}_i\cdot\vec{u}) + \frac{3}{2}(\hat{e}_i\cdot\vec{u})^2 - \frac{1}{2}u^2\right\} + w_i^p\Gamma\mu.$$

These expressions are only suitable for square lattices (i.e. those currently implemented in DL_MESO_LBE) apart from D3Q27, for which no free-energy scheme currently exists.

Parameters

out	feq	Local equilibrium distribution functions for given lattice site
in	v	Fluid velocity at lattice site $\vec{u}$
in	rho	Fluid density at lattice site ρ
in	phi	Fluid concentration at lattice site ϕ
in	p0	Bulk pressure at lattice site (determined from equation of state) P_0
in	pot	Potential at lattice site (determined from equation of state and free energy functional) μ
in	lambda	Correction parameter for Galilean invariance (determined from equation of state) λ
in	delta	First-order and second-order derivatives of density and concentration

fGetEquilibriumT()

```
int fGetEquilibriumT (double * feq,
                      double * v,
                      double rho)
```

Calculates the local equilibrium distribution functions for temperature [62]:

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

as required for collisions and system initialisation. This expression is only suitable for square lattices (i.e. those currently implemented in DL_MESO_LBE).

Parameters

out	feq	Local equilibrium distribution functions for solute at given lattice site
in	v	Fluid velocity at lattice site $\vec{u}$
in	rho	Temperature at lattice site T

fGetLambdaSwift()

```
double fGetLambdaSwift (double rho,
                        double omega,
                        double T)
```

Calculates and returns the correction factor to ensure Galilean invariance for one or two fluids undergoing Swift free-energy interactions, based on the governing equation of state:

$$\lambda = \nu \left(1 - 3 \left(\frac{\Delta t}{\Delta x} \right)^2 \frac{\partial P_0}{\partial \rho} \right)$$

where P_0 is the bulk pressure obtained for the equation of state and ν is the kinematic viscosity of the fluids (which can be obtained using relaxation frequencies). This expression reduces to zero in the case of a lattice gas with the equation of state $P_0 = c_s^2 \rho$.

Parameters

in	rho	Fluid density at lattice site ρ
in	omega	Relaxation frequency of fluid(s) at lattice site ω
in	T	Temperature at lattice site

fGetModel()

```
int fGetModel ()
```

Assigns values for link vectors, weighting parameters for local equilibrium distribution functions, conjugate links and transformation matrices for multiple relaxation time (MRT) collision schemes based on the selected lattice model (numbers of space dimensions and discrete velocities) and mesophase interaction model.

fGetPotentialSwift()

```
double fGetPotentialSwift (double phi, double d2phi)
```

Calculates and returns the potential at a given lattice point for two-fluid Swift free-energy interactions, using the following expression for a double well potential:

$$\mu = a \left(-\phi + \phi^3 \right) - \kappa \nabla^2 \phi,$$

where a is a parameter that can control the surface tension between the two fluids and the interfacial width.

Parameters

in	phi	Fluid concentration at lattice site ϕ
in	d2phi	Second-order derivative of fluid concentration $\nabla^2 \phi$

fInitializeSystem()

```
int fInitializeSystem ()
```

Sets the starting distribution functions for a Lattice Boltzmann Equation (LBE) calculation based on initial values for fluid velocity, densities, solute concentrations and temperatures at each lattice point. Local equilibrium distribution functions are used to obtain initial values for distribution functions: the exact forms for these depend on whether or not the fluids are compressible, if cascaded LBE (CLBE) collisions or Swift free-energy interactions are in use. The initial fluid densities can be varied by random 'noise' with a maximum fluctuation given by the user: this can be used for multiple fluid/phase simulations to give initial density gradients that can instigate separation. This subroutine makes use of initial conditions specified in the input system file (lbin.sys) as defaults for all fluid lattice sites: an initial state input file (lbin.init) can be used to override these defaults at any specified lattice sites.

fMarkBoundArea()

```
int fMarkBoundArea ()
```

Assigns boundary conditions to the edges of a subdomain indicating the lattice points making up the boundary halo for LBE calculations. This subroutine is essential for parallel calculations to indicate lattice points involved in processor-to-processor communications and for serial calculations that use a non-zero boundary halo: serial calculations that do not require a boundary halo do not need to call this subroutine.

fMarkBoundArea2D()

```
int fMarkBoundArea2D ()
```

Assigns boundary conditions to the edges of a subdomain indicating the lattice points making up the boundary halo for LBE calculations in a two-dimensional lattice. The assigned boundary condition (phase field) code indicates lattice sites that are otherwise fluid (non-boundary) points and are treated as such for e.g. collisions.

fMarkBoundArea3D()

```
int fMarkBoundArea3D ()
```

Assigns boundary conditions to the edges of a subdomain indicating the lattice points making up the boundary halo for LBE calculations in a three-dimensional lattice. The assigned boundary condition (phase field) code indicates lattice sites that are otherwise fluid (non-boundary) points and are treated as such for e.g. collisions.

fMemoryAllocation()

```
int fMemoryAllocation ()
```

Allocates the main arrays for a Lattice Boltzmann Equation (LBE) calculation: distribution functions, interaction type-dependent property (Shan-Chen pseudopotentials, Lishchuk interfacial normals, gradients of density/concentration for Swift free-energy interactions), initial and boundary conditions, lattice weighting parameters and link vectors, interaction forces, simulation parameters etc.

fPropagationCombinedSwap()

```
int fPropagationCombinedSwap ()
```

Shifts distribution functions to neighbouring lattice points along link vectors by systematic swapping of post-collisional distribution functions [93], initially at each lattice point and then between them within the same loop. This routine should not be modified unless the storage structure for distribution functions is changed. This routine provides an efficient propagation method for calculations in DL_MESO_LBE, but it can only be used when there is a boundary halo in use, as the swaps between lattice points require both points to have already swapped distribution functions among their conjugate links (which cannot happen when modulo functions are in use). As such, it is therefore the default propagation method in DL_MESO_LBE for parallel calculations when OpenMP is not in use.

fPropagationSwap()

```
int fPropagationSwap ()
```

Shifts distribution functions to neighbouring lattice points along link vectors by systematic swapping of post-collisional distribution functions [93], initially at each lattice point and then between them in two separate loops. This routine should not be modified unless the storage structure for distribution functions is changed. This routine provides an efficient propagation method for both serial and parallel calculations in DL_MESO_LBE as it uses less memory and can be carried out over multiple OpenMP threads: it is therefore the default propagation method in DL_MESO_LBE for serial calculations and parallel calculations when OpenMP is in use.

fPropagationTwoLattice()

```
int fPropagationTwoLattice ()
```

Shifts distribution functions to neighbouring lattice points along link vectors by copying values to an additional second lattice and copying the values back into the main distribution function array afterwards. This routine should not be modified unless the storage structure for distribution functions is changed and provides the least efficient propagation method available in DL_MESO_LBE: for this reason it is not the method used by default.

fSetSerialDomain()

```
int fSetSerialDomain ()
```

Sets the sizes of the lattice based on inputted values, ignoring the boundary halo size (as modulo functions can be used to deal with periodic boundaries), and sets a grid boundary region close to the edge of the domain to find neighbouring lattice points for e.g. interaction forces more efficiently. This subroutine is only used for serial DL_MESO_LBE runs: an alternative subroutine exists for parallel running (*fDefineDomain()*).

fSetSerialDomainBuffer()

```
int fSetSerialDomainBuffer ()
```

Sets the sizes of the lattice based on inputted values, using the boundary halo size to find outer lattice extents, and sets a grid boundary region close to the edge of the domain to find neighbouring lattice points for e.g. interaction forces more efficiently. This subroutine is an alternative to *fSetSerialDomain()* that is not normally used by default and is a serial equivalent to the *fDefineDomain()* subroutine used for parallel calculations.

fsPrintDomainInfo()

```
int fsPrintDomainInfo ()
```

Prints the number of available threads to the standard output if OpenMP is in use. This subroutine is only used for serial calculations: an alternative subroutine for parallel running (*fPrintDomainInfo()*) that also indicates the number of processors in use. If DL_MESO_LBE is not compiled with OpenMP, this message is not printed.

fStartDLMESO()

```
int fStartDLMESO ()
```

Prints messages at the start of a DL_MESO_LBE calculation indicating the code version (with minor revision number and release date), authors, contributors, copyright message and starting time to standard output.

fWeakMemory()

```
inline void fWeakMemory ()
```

Prints error message and stops DL_MESO_LBE if there is insufficient memory available to allocate arrays for LBE calculations.

5.12 lbpMPI.cpp

Module with functions and subroutines required for parallel running using MPI. (Header file available as lbpMPI.hpp.)

Functions and subroutines required to set up parallel LBE calculations using the Message Passing Interface (MPI), including processor-to-processor communications, global summations, simulation setup over multiple processors etc.

5.12.1 Functions

- `int` *fStartMPI()*

  Starts off MPI for a parallel simulation by DL_MESO_LBE.

- `int` *fCloseMPI()*

  Closes MPI after completion of DL_MESO_LBE calculation in a controlled manner.

- `int` *fGetRank()*

  Returns number (rank) of current processor.

- `int` *fGetSize()*

  Returns total number of available processors.

- `int` *fAllReady()*

  Synchronises all processors before continuing.

- `int` *fGlobalValue()*

  Applies a global summation to an integer, long integer or double-precision floating-point array and broadcasts result to all processors in either the original array or as a new array.

- `int` *fGlobalProduct()*

  Applies a global product to an integer or double-precision floating-point array and broadcasts result to all processors.

- `double` *fCheckTimeMPI()*

  Outputs time in seconds.

- `int` *fArrangeProcessors()*

  Arrange processors to best fit system dimensions.

- `int` *fGetProcessCoordinate()*

  Determine processor coordinate for current processor.
- `int` *fGetDomainSize()*

  Distribute lattice points among all available processors.
- `int` *fErrorInArray()*

  Checks for and report errors in determined lattice dimensions.
- `int` *fDefineDomain()*

  Determines the domain parameters for the parallel LBE calculation.
- `int` *fDefineMessage()*

  Defines vector messages for system to communicate calculational properties between processors.
- `int` *fDefineNeighbours()*

  Defines the neighbouring processors and data locations for processor-to-processor communications.
- `int` *fNonBlockComm2DX()*

  Passes distribution functions in x-direction for two-dimensional LBE simulation.
- `int` *fNonBlockComm2DY()*

  Passes distribution functions in y-direction for two-dimensional LBE simulation.
- `int` *fNonBlockComm2D()*

  Passes distribution functions to boundary halos for two-dimensional LBE simulation.
- `int` *fNonBlockComm3DX()*

  Passes distribution functions in x-direction for three-dimensional LBE simulation.
- `int` *fNonBlockComm3DY()*

  Passes distribution functions in y-direction for three-dimensional LBE simulation.
- `int` *fNonBlockComm3DZ()*

  Passes distribution functions in z-direction for three-dimensional LBE simulation.
- `int` *fNonBlockComm3D()*

  Passes distribution functions to boundary halos for three-dimensional LBE simulation.
- `int` *fNonBlockCommunication()*

  Passes distribution functions to boundary halos for LBE simulation.
- `int` *fPrintDomainInfo()*

  Prints information about numbers of processors and threads to standard output.
- `int` *fBroadcast()*

  Broadcasts an integer across all processors.
- `int` *fMPISetoffSteer()*

  Creates file to prevent reading in input files when using computational steering (in parallel).
- `int` *fMPICheckSteer()*

  Checks for file indicating steering is occurring and reads in input files if it does not exist (in parallel).
- `int` *fBoundNonBlockComm2DX()*

  Passes boundary conditions in x-direction for two-dimensional LBE simulation.

- `int` *fBoundNonBlockComm2DY()*

  Passes boundary conditions in y-direction for two-dimensional LBE simulation.

- `int` *fBoundNonBlockComm2D()*

  Passes boundary conditions to boundary halos for two-dimensional LBE simulation.

- `int` *fBoundNonBlockComm3DX()*

  Passes boundary conditions in x-direction for three-dimensional LBE simulation.

- `int` *fBoundNonBlockComm3DY()*

  Passes boundary conditions in y-direction for three-dimensional LBE simulation.

- `int` *fBoundNonBlockComm3DZ()*

  Passes boundary conditions in z-direction for three-dimensional LBE simulation.

- `int` *fBoundNonBlockComm3D()*

  Passes boundary conditions to boundary halos for three-dimensional LBE simulation.

- `int` *fBoundNonBlockCommunication()*

  Passes boundary conditions to boundary halos for LBE simulation.

- `int` *fForceNonBlockComm2DX()*

  Passes interaction forces in x-direction for two-dimensional LBE simulation.

- `int` *fForceNonBlockComm2DY()*

  Passes interaction forces in y-direction for two-dimensional LBE simulation.

- `int` *fForceNonBlockComm2D()*

  Passes interfacial forces to boundary halos for two-dimensional LBE simulation.

- `int` *fForceNonBlockComm3DX()*

  Passes interaction forces in x-direction for three-dimensional LBE simulation.

- `int` *fForceNonBlockComm3DY()*

  Passes interaction forces in y-direction for three-dimensional LBE simulation.

- `int` *fForceNonBlockComm3DZ()*

  Passes interaction forces in z-direction for three-dimensional LBE simulation.

- `int` *fForceNonBlockComm3D()*

  Passes interfacial forces to boundary halos for three-dimensional LBE simulation.

- `int` *fForceNonBlockCommunication()*

  Passes interfacial forces to boundary halos for LBE simulation.

- `int` *fIndexNonBlockComm2DX()*

  Passes Lishchuk phase indices or Swift free-energy density/concentration gradients in x-direction for two-dimensional LBE simulation.

- `int` *fIndexNonBlockComm2DY()*

  Passes Lishchuk phase indices or Swift free-energy density/concentration gradients in y-direction for two-dimensional LBE simulation.

- `int` *fIndexNonBlockComm2D()*

  Passes Lishchuk phase indices or Swift free-energy density/concentration gradients to boundary halos for two-dimensional LBE simulation.

- `int` *fIndexNonBlockComm3DX()*

  Passes Lishchuk phase indices or Swift free-energy density/concentration gradients in x-direction for three-dimensional LBE simulation.

- `int` *fIndexNonBlockComm3DY()*

  Passes Lishchuk phase indices or Swift free-energy density/concentration gradients in y-direction for three-dimensional LBE simulation.

- `int` *fIndexNonBlockComm3DZ()*

  Passes Lishchuk phase indices or Swift free-energy density/concentration gradients in z-direction for three-dimensional LBE simulation.

- `int` *fIndexNonBlockComm3D()*

  Passes Lishchuk phase indices or Swift free-energy density/concentration gradients to boundary halos for three-dimensional LBE simulation.

- `int` *fIndexNonBlockCommunication()*

  Passes Lishchuk phase indices or Swift free-energy density/concentration gradients to boundary halos for LBE simulation.

- `int` *fPrintSystemMass()*

  Calculates and prints total and individual fluid masses in entire system.

- `int` *fPrintSystemMomentum()*

  Calculates and prints total fluid momentum in entire system.

5.12.2 Function Documentation

fAllReady()

```
int fAllReady ()
```

Pauses running until all processors are sychronised and have reached a given point in the code: needed when all processors need to be involved with what happens subsequently (i.e. are all ready to continue).

fArrangeProcessors()

```
int fArrangeProcessors ()
```

Determine the numbers of available processors in each dimension to best fit the lattice grid, i.e. to give as square or cubic subdomains as possible:

$$\frac{N_x}{P_x} \simeq \frac{N_y}{P_y} \simeq \frac{N_z}{P_z}$$

where N_α and P_α are the numbers of lattice points and processors respectively in dimension α. (The total number of processors is $P_x P_y P_z$, where $P_z = 1$ for two-dimensional simulations.)

fBoundNonBlockComm2D()

```
int fBoundNonBlockComm2D ()
```

Sends and receives boundary condition (phase field), neighbouring point and surface normal information between neighbouring processors to generate boundary halos required to complete two-dimensional LBE calculations. This subroutine is used to set off the communications in the required directions.

fBoundNonBlockComm2DX()

```
int fBoundNonBlockComm2DX ()
```

Sends and receives boundary condition (phase field), neighbouring point and surface normal information in +x and -x directions to generate boundary halos required to complete two-dimensional LBE calculations, using unblocked MPI calls. MPI derived datatypes are used to send and receive data vectors that are placed directly into the boundary condition, neighbouring point and surface normal arrays as boundary halos.

fBoundNonBlockComm2DY()

```
int fBoundNonBlockComm2DY ()
```

Sends and receives boundary condition (phase field), neighbouring point and surface normal information in +y and -y directions to generate boundary halos required to complete two-dimensional LBE calculations, using unblocked MPI calls. MPI derived datatypes are used to send and receive data vectors that are placed directly into the boundary condition, neighbouring point and surface normal arrays as boundary halos.

fBoundNonBlockComm3D()

```
int fBoundNonBlockComm3D ()
```

Sends and receives boundary condition (phase field), neighbouring point and surface normal information between neighbouring processors to generate boundary halos required to complete three-dimensional LBE calculations. This subroutine is used to set off the communications in the required directions.

fBoundNonBlockComm3DX()

```
int fBoundNonBlockComm3DX ()
```

Sends and receives boundary condition (phase field), neighbouring point and surface normal information in +x and -x directions to generate boundary halos required to complete three-dimensional LBE calculations, using unblocked MPI calls. MPI derived datatypes are used to send and receive data vectors that are placed directly into the boundary condition, neighbouring point and surface normal arrays as boundary halos.

fBoundNonBlockComm3DY()

```
int fBoundNonBlockComm3DY ()
```

Sends and receives boundary condition (phase field), neighbouring point and surface normal information in +y and -y directions to generate boundary halos required to complete three-dimensional LBE calculations, using unblocked MPI calls. MPI derived datatypes are used to send and receive data vectors that are placed directly into the boundary condition, neighbouring point and surface normal arrays as boundary halos.

fBoundNonBlockComm3DZ()

```
int fBoundNonBlockComm3DZ ()
```

Sends and receives boundary condition (phase field), neighbouring point and surface normal information in +z and -z directions to generate boundary halos required to complete three-dimensional LBE calculations, using unblocked MPI calls. MPI derived datatypes are used to send and receive data vectors that are placed directly into the boundary condition, neighbouring point and surface normal arrays as boundary halos.

fBoundNonBlockCommunication()

```
int fBoundNonBlockCommunication ()
```

Sends and receives boundary condition (phase field), neighbouring point and surface normal information between neighbouring processors to generate boundary halos required to complete LBE calculations. This subroutine is used to choose in which directions the communications should take place based on the number of dimensions in the system.

fBroadcast()

```
int fBroadcast (int * item1)
```

Broadcasts an integer from processor 0 to all processors.

Parameters

in	item1	Integer to be broadcast to all processors

fCheckTimeMPI()

```
double fCheckTimeMPI ()
```

Checks the time since the first call of the function obtained from MPI wall time. This function is used to time DL_MESO_LBE simulations run in parallel: there is an alternative function to do the same for serial calculations (*fCheckTimeSerial()*).

fCloseMPI()

```
int fCloseMPI ()
```

Calls MPI routine to close all communications to end the DL_MESO_LBE after a successful calculation.

fDefineDomain()

```
int fDefineDomain ()
```

Obtains processor rank (number) and the total number of processors, arranges the processors to best fit the specified lattice domain, determines where the current processor is located within the system, distribute lattice points among the processors and check to ensure at least one lattice point is assigned to each processor. This subroutine is only used for parallel DL_MESO_LBE runs: an alternative subroutine exists for serial running (*fSetSerialDomain()*).

fDefineMessage()

```
int fDefineMessage ()
```

Defines MPI derived datatypes to specify vectors of data (distribution functions, boundary condition properties, interaction forces and phase indices or density/concentration gradients) to send and receive between neighbouring processors as boundary halos. The subroutines used by default to create the MPI derived datatypes are based on MPI-2.x or later, although a compile-time option is available to substitute MPI-1.x subroutines. Another compile-time option is available to pack and unpack data into and from one-dimensional buffer arrays as an alternative to using MPI derived datatypes: this subroutine allocates the buffer arrays required to carry out this form of communication.

fDefineNeighbours()

```
int fDefineNeighbours ()
```

Determines which processors are neighbours to the current processor in up to six directions (+x, -x, +y, -y, +z, -z), and the starting locations in memory for data to send to neighbours and data received from neighbours as boundary halos.

fErrorInArray()

```
int fErrorInArray ()
```

Checks that the processor’s lattice dimensions without boundary halos are at least 1 in all dimensions for simulation: if any of them are zero, an error message is printed to standard output and DL_MESO_LBE stops.

fForceNonBlockComm2D()

```
int fForceNonBlockComm2D ()
```

Sends and receives interfacial forces between neighbouring processors to generate boundary halos required to complete two-dimensional LBE calculations. This subroutine is used to set off the communications in the required directions.

fForceNonBlockComm2DX()

```
int fForceNonBlockComm2DX ()
```

Sends and receives interaction forces in +x and -x directions to generate boundary halos required to complete two-dimensional LBE calculations, using unblocked MPI calls. By default, MPI derived datatypes are used to send and receive data vectors that are placed directly into the distribution function array as boundary halos. If the compile-time alternative option is invoked, the data is packed into one-dimensional buffer arrays, communicated and unpacked on arrival: this approach can be sped up using OpenMP multithreading on buffer packing and unpacking.

fForceNonBlockComm2DY()

```
int fForceNonBlockComm2DY ()
```

Sends and receives interaction forces in +y and -y directions to generate boundary halos required to complete two-dimensional LBE calculations, using unblocked MPI calls. By default, MPI derived datatypes are used to send and receive data vectors that are placed directly into the distribution function array as boundary halos. If the compile-time alternative option is invoked, the data is packed into one-dimensional buffer arrays, communicated and unpacked on arrival: this approach can be sped up using OpenMP multithreading on buffer packing and unpacking.

fForceNonBlockComm3D()

```
int fForceNonBlockComm3D ()
```

Sends and receives interfacial forces between neighbouring processors to generate boundary halos required to complete three-dimensional LBE calculations. This subroutine is used to set off the communications in the required directions.

fForceNonBlockComm3DX()

```
int fForceNonBlockComm3DX ()
```

Sends and receives interaction forces in +x and -x directions to generate boundary halos required to complete three-dimensional LBE calculations, using unblocked MPI calls. By default, MPI derived datatypes are used to send and receive data vectors that are placed directly into the distribution function array as boundary halos. If the compile-time alternative option is invoked, the data is packed into one-dimensional buffer arrays, communicated and unpacked on arrival: this approach can be sped up using OpenMP multithreading on buffer packing and unpacking.

fForceNonBlockComm3DY()

```
int fForceNonBlockComm3DY ()
```

Sends and receives interaction forces in +y and -y directions to generate boundary halos required to complete three-dimensional LBE calculations, using unblocked MPI calls. By default, MPI derived datatypes are used to send and receive data vectors that are placed directly into the distribution function array as boundary halos. If the compile-time alternative option is invoked, the data is packed into one-dimensional buffer arrays, communicated and unpacked on arrival: this approach can be sped up using OpenMP multithreading on buffer packing and unpacking.

fForceNonBlockComm3DZ()

```
int fForceNonBlockComm3DZ ()
```

Sends and receives interaction forces in +z and -z directions to generate boundary halos required to complete three-dimensional LBE calculations, using unblocked MPI calls. By default, MPI derived datatypes are used to send and receive data vectors that are placed directly into the distribution function array as boundary halos. If the compile-time alternative option is invoked, the data is packed into one-dimensional buffer arrays, communicated and unpacked on arrival: this approach can be sped up using OpenMP multithreading on buffer packing and unpacking.

fForceNonBlockCommunication()

```
int fForceNonBlockCommunication ()
```

Sends and receives boundary condition (phase field), neighbouring point and surface normal information between neighbouring processors to generate boundary halos required to complete LBE calculations. This subroutine is used to choose in which directions the communications should take place based on the number of dimensions in the system.

fGetDomainSize()

```
int fGetDomainSize ()
```

Determines the number of lattice points for each processor (both including and excluding boundary halo points), based on the best fit to the numbers of processors in each direction, and sets grid boundary regions close to the edges of each processor's subdomain.

fGetProcessCoordinate()

```
int fGetProcessCoordinate ()
```

Find the location of the current processor within the grid of processors as a Cartesian coordinate.

fGetRank()

```
int fGetRank ()
```

Finds rank (number) of current processor, which can range from 0 to the number of processors less 1.

fGetSize()

```
int fGetSize ()
```

Finds the total number of processors available for DL_MESO_LBE to run.

fGlobalProduct()

```
int fGlobalProduct (double * vqua, int nnum)
```

Carries out an MPI_Allreduce operation on an array of double-precision floating-point or integer numbers to find the products for all elements across all processors and share the result in the same array.

Parameters

in,out	vqua	Double-precision floating-point or integer array on which to apply global product
in	nnum	Size of double-precision floating-point or integer array

fGlobalValue()

```
int fGlobalValue (double * vqua, int nnum)
int fGlobalValue (double * vqua, int nnum, double * vtot)
int fGlobalValue (int * vqua, int nnum)
int fGlobalValue (int * vqua, int nnum, int * vtot)
int fGlobalValue (long int * vqua, int nnum)
int fGlobalValue (long int * vqua, int nnum, long int * vtot)
```

Carries out an MPI_Allreduce operation on an array of double-precision floating-point, integer or long integer numbers to find the sums for all elements across all processors and share the result either in the same array or as a new array.

Parameters

in,[out]	vqua	Double-precision floating-point, integer or long integer array on which to apply global summation
in	nnum	Size of double-precision floating-point, integer or long integer array
out	vtot	Double-precision floating-point, integer or long integer array with resulting global summation

fIndexNonBlockComm2D()

```
int fIndexNonBlockComm2D ()
```

Sends and receives Lishchuk phase indices or Swift free-energy density/concentration gradients between neighbouring processors to generate boundary halos required to complete two-dimensional LBE calculations. This subroutine is used to set off the communications in the required directions.

fIndexNonBlockComm2DX()

```
int fIndexNonBlockComm2DX ()
```

Sends and receives Lishchuk phase indices or Swift-free energy density and concentration gradients in +x and -x directions to generate boundary halos required to complete two-dimensional LBE calculations, using unblocked MPI calls. By default, MPI derived datatypes are used to send and receive data vectors that are placed directly into the phase index/gradient array as boundary halos. If the compile-time alternative option is invoked, the data is packed into one-dimensional buffer arrays, communicated and unpacked on arrival: this approach can be sped up using OpenMP multithreading on buffer packing and unpacking.

fIndexNonBlockComm2DY()

```
int fIndexNonBlockComm2DY ()
```

Sends and receives Lishchuk phase indices or Swift-free energy density and concentration gradients in +y and -y directions to generate boundary halos required to complete two-dimensional LBE calculations, using unblocked MPI calls. By default, MPI derived datatypes are used to send and receive data vectors that are placed directly into the phase index/gradient array as boundary halos. If the compile-time alternative option is invoked, the data is packed into one-dimensional buffer arrays, communicated and unpacked on arrival: this approach can be sped up using OpenMP multithreading on buffer packing and unpacking.

fIndexNonBlockComm3D()

```
int fIndexNonBlockComm3D ()
```

Sends and receives Lishchuk phase indices or Swift free-energy density/concentration gradients between neighbouring processors to generate boundary halos required to complete three-dimensional LBE calculations. This subroutine is used to set off the communications in the required directions.

fIndexNonBlockComm3DX()

```
int fIndexNonBlockComm3DX ()
```

Sends and receives Lishchuk phase indices or Swift-free energy density and concentration gradients in +x and -x directions to generate boundary halos required to complete three-dimensional LBE calculations, using unblocked MPI calls. By default, MPI derived datatypes are used to send and receive data vectors that are placed directly into the phase index/gradient array as boundary halos. If the compile-time alternative option is invoked, the data is packed into one-dimensional buffer arrays, communicated and unpacked on arrival: this approach can be sped up using OpenMP multithreading on buffer packing and unpacking.

fIndexNonBlockComm3DY()

```
int fIndexNonBlockComm3DY ()
```

Sends and receives Lishchuk phase indices or Swift-free energy density and concentration gradients in +y and -y directions to generate boundary halos required to complete three-dimensional LBE calculations, using unblocked MPI calls. By default, MPI derived datatypes are used to send and receive data vectors that are placed directly into the phase index/gradient array as boundary halos. If the compile-time alternative option is invoked, the data is packed into one-dimensional buffer arrays, communicated and unpacked on arrival: this approach can be sped up using OpenMP multithreading on buffer packing and unpacking.

fIndexNonBlockComm3DZ()

```
int fIndexNonBlockComm3DZ ()
```

Sends and receives Lishchuk phase indices or Swift-free energy density and concentration gradients in +z and -z directions to generate boundary halos required to complete three-dimensional LBE calculations, using unblocked MPI calls. By default, MPI derived datatypes are used to send and receive data vectors that are placed directly into the phase index/gradient array as boundary halos. If the compile-time alternative option is invoked, the data is packed into one-dimensional buffer arrays, communicated and unpacked on arrival: this approach can be sped up using OpenMP multithreading on buffer packing and unpacking.

fIndexNonBlockCommunication()

```
int fIndexNonBlockCommunication ()
```

Sends and receives Lishchuk phase indices or Swift free-energy density/concentration gradients between neighbouring processors to generate boundary halos required to complete LBE calculations. This subroutine is used to choose in which directions the communications should take place based on the number of dimensions in the system.

fMPICheckSteer()

```
int fMPICheckSteer ()
```

Checks for the existence of a file called notsteer, which was created to prevent DL_MESO_LBE from starting a new simulation when computaional steering is applied. If the files does not exist, read in system and space property files. This routine is for parallel calculations: an atlnerative routine exists for serial running (*fCheckSteer()*), but neither routine is currently in use in the main DL_MESO_LBE code.

fMPISetoffSteer()

```
int fMPISetoffSteer ()
```

Creates a file called notsteer to prevent DL_MESO_LBE from starting a new simulation by reading in system and space property files when a LBE simulation is computationally steered. This routine is for parallel calculations: an alternative routine exists for serial running (*fSetoffSteer()*), but neither routine is currently in use in the main DL_MESO_LBE code.

fNonBlockComm2D()

```
int fNonBlockComm2D ()
```

Sends and receives distribution functions between neighbouring processors to generate boundary halos required to complete two-dimensional LBE calculations. This subroutine is used to set off the communications in the required directions.

fNonBlockComm2DX()

```
int fNonBlockComm2DX ()
```

Sends and receives distribution functions in +x and -x directions to generate boundary halos required to complete two-dimensional LBE calculations, using unblocked MPI calls. By default, MPI derived datatypes are used to send and receive data vectors that are placed directly into the distribution function array as boundary halos. If the compile-time alternative option is invoked, the data is packed into one-dimensional buffer arrays, communicated and unpacked on arrival: this approach can be sped up using OpenMP multithreading on buffer packing and unpacking.

fNonBlockComm2DY()

```
int fNonBlockComm2DY ()
```

Sends and receives distribution functions in +y and -y directions to generate boundary halos required to complete two-dimensional LBE calculations, using unblocked MPI calls. By default, MPI derived datatypes are used to send and receive data vectors that are placed directly into the distribution function array as boundary halos. If the compile-time alternative option is invoked, the data is packed into one-dimensional buffer arrays, communicated and unpacked on arrival: this approach can be sped up using OpenMP multithreading on buffer packing and unpacking.

fNonBlockComm3D()

```
int fNonBlockComm3D ()
```

Sends and receives distribution functions between neighbouring processors to generate boundary halos required to complete three-dimensional LBE calculations. This subroutine is used to set off the communications in the required directions.

fNonBlockComm3DX()

```
int fNonBlockComm3DX ()
```

Sends and receives distribution functions in +x and -x directions to generate boundary halos required to complete three-dimensional LBE calculations, using unblocked MPI calls. By default, MPI derived datatypes are used to send and receive data vectors that are placed directly into the distribution function array as boundary halos. If the compile-time alternative option is invoked, the data is packed into one-dimensional buffer arrays, communicated and unpacked on arrival: this approach can be sped up using OpenMP multithreading on buffer packing and unpacking.

fNonBlockComm3DY()

```
int fNonBlockComm3DY ()
```

Sends and receives distribution functions in +y and -y directions to generate boundary halos required to complete three-dimensional LBE calculations, using unblocked MPI calls. By default, MPI derived datatypes are used to send and receive data vectors that are placed directly into the distribution function array as boundary halos. If the compile-time alternative option is invoked, the data is packed into one-dimensional buffer arrays, communicated and unpacked on arrival: this approach can be sped up using OpenMP multithreading on buffer packing and unpacking.

fNonBlockComm3DZ()

```
int fNonBlockComm3DZ ()
```

Sends and receives distribution functions in +z and -z directions to generate boundary halos required to complete three-dimensional LBE calculations, using unblocked MPI calls. By default, MPI derived datatypes are used to send and receive data vectors that are placed directly into the distribution function array as boundary halos. If the compile-time alternative option is invoked, the data is packed into one-dimensional buffer arrays, communicated and unpacked on arrival: this approach can be sped up using OpenMP multithreading on buffer packing and unpacking.

fNonBlockCommunication()

```
int fNonBlockCommunication ()
```

Sends and receives distribution functions between neighbouring processors to generate boundary halos required to complete LBE calculations. This subroutine is used to choose in which directions the communications should take place based on the number of dimensions in the system.

fPrintDomainInfo()

```
int fPrintDomainInfo ()
```

Prints the number of available OpenMP threads per processor and the number of processors (including the extents of their subdomains within the overall lattice) to the standard output. This subroutine is only used for parallel calculations: an alternative subroutine for serial running (*fsPrintDomainInfo()*).

fPrintSystemMass()

```
int fPrintSystemMass ()
```

Calculates both the total mass and the individual masses of all fluids in the entire simulation lattice and prints the results to the standard output. This subroutine can only be used for parallel calculations: an alternative routine (*fPrintDomainMass()*) is available for printing total and individual fluid masses in serial.

fPrintSystemMomentum()

```
int fPrintSystemMomentum ()
```

Calculates the total momentum of all fluids in the entire simulation lattice and prints the result to the standard output. This subroutine can only be used for parallel calculations: an alternative routine (*fPrintDomainMomentum()*) is available for printing the total momentum in serial.

fStartMPI()

```
int fStartMPI (int argc, char * argv[])
```

Starts the Message Passing Interface (MPI) for the instance of DL_MESO_LBE.

Parameters

in	argc	Number of command-line arguments included in command to launch DL_MESO_LBE
in	argv	Character array of command-line arguments

5.13 lbpBGK.cpp

Module with routines for Bhatnagar-Gross-Krook (BGK) single relaxation time collisions. (Header file available as lbpBGK.hpp.)

Applies collisions to grid points using a single relaxation time scheme, known as Bhatnagar-Gross-Krook (BGK), on each fluid, i.e.

$$f_i\left(\vec{x},t^{+}\right) = f_i\left(\vec{x},t\right) - \frac{f_i\left(\vec{x},t\right) - f_i^{eq}\left(\rho\left(\vec{x},t\right),\vec{u}\left(\vec{x},t\right)\right)}{\tau}.$$

where f_i^{eq} is the local equilibrium distribution function (dependent on macroscopic fluid density ρ and velocity $\vec{u}$). Similar collisions can also be applied to solutes and temperature fields.

To apply forces to each fluid, one of four options can be applied. The standard (Martys-Chen) *[34] :cite:`martys1996* force scheme applies a modified velocity for calculating local equilibrium distribution functions:

$$\vec{v} = \vec{u} + \frac{\tau_f \vec{F} \Delta t}{\rho},$$

while the Equal Difference Method (EDM) [69] applies an additional forcing term that can be calculated as a difference in local equilibrium distribution functions:

$$F_i = f_i^{eq}\left(\rho, \vec{u} + \frac{\vec{F}\Delta t}{\rho}\right) - f_i^{eq}\left(\rho, \vec{u}\right).$$

The Guo scheme [49] both adjusts the velocity for local equilibrium distribution functions to $\vec{v} = \vec{u} + \frac{\vec{F}\Delta t}{2\rho}$ and includes the following forcing term for BGK collisions:

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

and the He scheme [54] uses the same adjusted velocity and the following forcing term for BGK collisions:

$$F_i = \left(1 - \frac{1}{2\tau_f}\right) \frac{f_i^{eq}}{\rho c_s^2} \left(\hat{e}_i - \vec{v}\right) \cdot \vec{F}.$$

5.13.1 Functions

- `int` *fSiteFluidCollisionBGK()*

  Applies BGK collisions to all compressible fluids at a given lattice site with standard forcing.

- `int` *fSiteFluidIncomCollisionBGK()*

  Applies BGK collisions to all incompressible fluids at a given lattice site with standard forcing.

- `int` *fSiteFluidCollisionBGKEDM()*

  Applies BGK collisions to all compressible fluids at a given lattice site with EDM forcing.

- `int` *fSiteFluidIncomCollisionBGKEDM()*

  Applies BGK collisions to all incompressible fluids at a given lattice site with EDM forcing.

- `int` *fSiteFluidCollisionBGKGuo()*

  Applies BGK collisions to all compressible fluids at a given lattice site with Guo forcing.

- `int` *fSiteFluidIncomCollisionBGKGuo()*

  Applies BGK collisions to all incompressible fluids at a given lattice site with Guo forcing.

- `int` *fSiteFluidCollisionBGKHe()*

  Applies BGK collisions to all compressible fluids at a given lattice site with He forcing.

- `int` *fSiteFluidIncomCollisionBGKHe()*

  Applies BGK collisions to all incompressible fluids at a given lattice site with He forcing.

- `int` *fSiteFluidCollisionBGKLishchuk()*

  Applies BGK collisions to all compressible fluids at a given lattice site with standard forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidIncomCollisionBGKLishchuk()*

  Applies BGK collisions to all incompressible fluids at a given lattice site with standard forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidCollisionBGKEDMLishchuk()*

  Applies BGK collisions to all compressible fluids at a given lattice site with EDM forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidIncomCollisionBGKEDMLishchuk()*

  Applies BGK collisions to all incompressible fluids at a given lattice site with EDM forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidCollisionBGKGuoLishchuk()*

  Applies BGK collisions to all compressible fluids at a given lattice site with Guo forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidIncomCollisionBGKGuoLishchuk()*

  Applies BGK collisions to all incompressible fluids at a given lattice site with Guo forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidCollisionBGKHeLishchuk()*

  Applies BGK collisions to all compressible fluids at a given lattice site with He forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidIncomCollisionBGKHeLishchuk()*

  Applies BGK collisions to all incompressible fluids at a given lattice site with He forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidCollisionBGKLishchukLocal()*

  Applies BGK collisions to all compressible fluids at a given lattice site with standard forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidIncomCollisionBGKLishchukLocal()*

  Applies BGK collisions to all incompressible fluids at a given lattice site with standard forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidCollisionBGKEDMLishchukLocal()*

  Applies BGK collisions to all compressible fluids at a given lattice site with EDM forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidIncomCollisionBGKEDMLishchukLocal()*

  Applies BGK collisions to all incompressible fluids at a given lattice site with EDM forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidCollisionBGKGuoLishchukLocal()*

  Applies BGK collisions to all compressible fluids at a given lattice site with Guo forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidIncomCollisionBGKGuoLishchukLocal()*

  Applies BGK collisions to all incompressible fluids at a given lattice site with Guo forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidCollisionBGKHeLishchukLocal()*

  Applies BGK collisions to all compressible fluids at a given lattice site with He forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidIncomCollisionBGKHeLishchukLocal()*

  Applies BGK collisions to all incompressible fluids at a given lattice site with He forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidCollisionBGKSwiftOneFluid()*

  Applies BGK collisions to one compressible fluid at a given lattice site with standard forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionBGKSwiftTwoFluid()*

  Applies BGK collisions to two compressible fluids at a given lattice site with standard forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionBGKEDMSwiftOneFluid()*

  Applies BGK collisions to one compressible fluid at a given lattice site with EDM forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionBGKEDMSwiftTwoFluid()*

  Applies BGK collisions to two compressible fluids at a given lattice site with EDM forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionBGKGuoSwiftOneFluid()*

  Applies BGK collisions to one compressible fluid at a given lattice site with Guo forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionBGKGuoSwiftTwoFluid()*

  Applies BGK collisions to two compressible fluids at a given lattice site with Guo forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionBGKHeSwiftOneFluid()*

  Applies BGK collisions to one compressible fluid at a given lattice site with He forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionBGKHeSwiftTwoFluid()*

  Applies BGK collisions to two compressible fluids at a given lattice site with He forcing and Swift free-energy interactions.

- `int` *fSiteSoluteCollisionBGK()*

  Applies BGK collisions to all solutes at a given lattice site with standard forcing.

- `int` *fSiteThermalCollisionBGK()*

  Applies BGK collisions to temperature field at a given lattice site with standard forcing.

- `int` *fCollisionBGK()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with standard forcing.

- `int` *fCollisionBGKEDM()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with EDM forcing.

- `int` *fCollisionBGKGuo()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with Guo forcing.

- `int` *fCollisionBGKHe()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with He forcing.

- `int` *fCollisionBGKShanChen()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with standard forcing for Shan-Chen interactions.

- `int` *fCollisionBGKEDMShanChen()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with EDM forcing for Shan-Chen interactions.

- `int` *fCollisionBGKGuoShanChen()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with Guo forcing for Shan-Chen interactions.

- `int` *fCollisionBGKHeShanChen()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with He forcing for Shan-Chen interactions.

- `int` *fCollisionBGKLishchuk()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with standard forcing and Lishchuk interactions provided as interfacial forces.

- `int` *fCollisionBGKEDMLishchuk()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with EDM forcing and Lishchuk interactions provided as interfacial forces.

- `int` *fCollisionBGKGuoLishchuk()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with Guo forcing and Lishchuk interactions provided as interfacial forces.

- `int` *fCollisionBGKHeLishchuk()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with He forcing and Lishchuk interactions provided as interfacial forces.

- `int` *fCollisionBGKLishchukLocal()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with standard forcing and Lishchuk interactions provided as an additional forcing term.

- `int` *fCollisionBGKEDMLishchukLocal()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with EDM forcing and Lishchuk interactions provided as an additional forcing term.

- `int` *fCollisionBGKGuoLishchukLocal()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with Guo forcing and Lishchuk interactions provided as an additional forcing term.

- `int` *fCollisionBGKHeLishchukLocal()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with He forcing and Lishchuk interactions provided as an additional forcing term.

- `int` *fCollisionBGKSwift()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with standard forcing for Swift free-energy interactions.

- `int` *fCollisionBGKEDMSwift()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with EDM forcing for Swift free-energy interactions.

- `int` *fCollisionBGKGuoSwift()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with Guo forcing for Swift free-energy interactions.

- `int` *fCollisionBGKHeSwift()*

  Applies collision steps for all fluids, solutes and temperature fields using BGK scheme with He forcing for Swift free-energy interactions.

5.13.2 Function Documentation

fCollisionBGK()

```
int fCollisionBGK ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with standard (Martys-Chen) [91] forcing. This version of the collisions uses the standard values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}.$$

fCollisionBGKEDM()

```
int fCollisionBGKEDM ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with Equal Difference Method (EDM) [69] forcing. This version of the collisions uses the standard values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}.$$

fCollisionBGKEDMLishchuk()

```
int fCollisionBGKEDMLishchuk ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with Equal Difference Method (EDM) [69] forcing, achromatic fluid collisions and segregation. The interfacial forces are applied using the main forcing scheme: this approach can be used with the original Lishchuk and Lishchuk-Spencer interaction models.

fCollisionBGKEDMLishchukLocal()

```
int fCollisionBGKEDMLishchukLocal ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with Equal Difference Method (EDM) [69] forcing, achromatic fluid collisions and segregation. The interfacial forces are applied using separate forcing terms: this approach can be used with the Lishchuk 'Spencer tensor' and local Lishchuk interaction models.

fCollisionBGKEDMShanChen()

```
int fCollisionBGKEDMShanChen ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with Equal Difference Method (EDM) [69] forcing. This version of the collisions uses the following values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} \frac{f_i^a \hat{e}_i}{\tau_f^a}}{\sum_{i,a} \frac{f_i^a}{\tau_f^a}}.$$

fCollisionBGKEDMSwift()

```
int fCollisionBGKEDMSwift ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with Equal Difference Method (EDM) [69] forcing and Swift free-energy interactions (enacted using modified local equilibrium distribution functions to incorporate density and concentration gradients).

fCollisionBGKGuo()

```
int fCollisionBGKGuo ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with Guo [49] forcing. This version of the collisions uses the standard values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}.$$

fCollisionBGKGuoLishchuk()

```
int fCollisionBGKGuoLishchuk ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions, using Guo [49] forcing, achromatic fluid collisions and segregation. The interfacial forces are applied using the main forcing scheme: this approach can be used with the original Lishchuk and Lishchuk-Spencer interaction models.

fCollisionBGKGuoLishchukLocal()

```
int fCollisionBGKGuoLishchukLocal ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with Guo [49] forcing, achromatic fluid collisions and segregation. The interfacial forces are applied using separate forcing terms: this approach can be used with the Lishchuk 'Spencer tensor' and local Lishchuk interaction models.

fCollisionBGKGuoShanChen()

```
int fCollisionBGKGuoShanChen ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with Guo [49] forcing. This version of the collisions uses the following values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} \frac{f_i^a \hat{e}_i}{\tau_f^a}}{\sum_{i,a} \frac{f_i^a}{\tau_f^a}}.$$

fCollisionBGKGuoSwift()

```
int fCollisionBGKGuoSwift ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with Guo [49] forcing and Swift free-energy interactions (enacted using modified local equilibrium distribution functions to incorporate density and concentration gradients).

fCollisionBGKHe()

```
int fCollisionBGKHe ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with He [54] forcing. This version of the collisions uses the standard values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}.$$

fCollisionBGKHeLishchuk()

```
int fCollisionBGKHeLishchuk ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions, using He [54] forcing, achromatic fluid collisions and segregation. The interfacial forces are applied using the main forcing scheme: this approach can be used with the original Lishchuk and Lishchuk-Spencer interaction models.

fCollisionBGKHeLishchukLocal()

```
int fCollisionBGKHeLishchukLocal ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with He [54] forcing, achromatic fluid collisions and segregation. The interfacial forces are applied using separate forcing terms: this approach can be used with the Lishchuk 'Spencer tensor' and local Lishchuk interaction models.

fCollisionBGKHeShanChen()

```
int fCollisionBGKHeShanChen ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with He [54] forcing. This version of the collisions uses the following values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} \frac{f_i^a \hat{e}_i}{\tau_f^a}}{\sum_{i,a} \frac{f_i^a}{\tau_f^a}}.$$

fCollisionBGKHeSwift()

```
int fCollisionBGKHeSwift ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with He [54] forcing and Swift free-energy interactions (enacted using modified local equilibrium distribution functions to incorporate density and concentration gradients).

fCollisionBGKLishchuk()

```
int fCollisionBGKLishchuk ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with standard (Martys-Chen) [91] forcing, achromatic fluid collisions and segregation. The interfacial forces are applied using the main forcing scheme: this approach can be used with the original Lishchuk and Lishchuk-Spencer interaction models.

fCollisionBGKLishchukLocal()

```
int fCollisionBGKLishchukLocal ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with standard (Martys-Chen) [91] forcing, achromatic fluid collisions and segregation. The interfacial forces are applied using separate forcing terms: this approach can be used with the Lishchuk 'Spencer tensor' and local Lishchuk interaction models.

fCollisionBGKShanChen()

```
int fCollisionBGKShanChen ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with standard (Martys-Chen) [91] forcing. This version of the collisions uses the following values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} \frac{f_i^a \hat{e}_i}{\tau_f^a}}{\sum_{i,a} \frac{f_i^a}{\tau_f^a}}.$$

fCollisionBGKSwift()

```
int fCollisionBGKSwift ()
```

Loops through all available lattice sites and applies collisions to all fluids, all solutes and any temperature field using single relaxation time BGK collisions with standard (Martys-Chen) [91] forcing and Swift free-energy interactions (enacted using modified local equilibrium distribution functions to incorporate density and concentration gradients).

fSiteFluidCollisionBGK()

```
int fSiteFluidCollisionBGK (double * startpos,
                            double * sitespeed,
                            double * omega,
                            double * rho,
                            double * bodyforce)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for mildly compressible fluids and applying standard (Martys-Chen) [91] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidCollisionBGKEDM()

```
int fSiteFluidCollisionBGKEDM (double * startpos,
                               double * sitespeed,
                               double * omega,
                               double * rho,
                               double * bodyforce)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for mildly compressible fluids and applying Equal Difference Method (EDM) [69] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidCollisionBGKEDMLishchuk()

```
int fSiteFluidCollisionBGKEDMLishchuk (double * startpos,
                                       double * sitespeed,
                                       double * omega,
                                       double * rho,
                                       double * bodyforce,
                                       double * phaseindex)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying Equal Difference Method (EDM) [69] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidCollisionBGKEDMLishchukLocal()

```
int fSiteFluidCollisionBGKEDMLishchukLocal (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * rho,
                                            double * bodyforce,
                                            double * phaseindex,
                                            int threed)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying Equal Difference Method (EDM) [69] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i \hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidCollisionBGKEDMSwiftOneFluid()

```
int fSiteFluidCollisionBGKEDMSwiftOneFluid (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * rho,
                                            double * gradient,
                                            double * bodyforce,
                                            double T)
```

Applies single relaxation time (BGK) collisions to one fluid at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for a compressible fluid undergoing Swift free-energy interactions and applying Equal Difference Method (EDM) [69] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequency for fluid at lattice site
in	rho	Macroscopic fluid density at lattice site
in	gradient	Density gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluid at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionBGKEDMSwiftTwoFluid()

```
int fSiteFluidCollisionBGKEDMSwiftTwoFluid (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * rho,
                                            double * gradient,
                                            double * bodyforce,
                                            double T)
```

Applies single relaxation time (BGK) collisions to two fluid at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for two compressible fluids undergoing Swift free-energy interactions (for fluid density and concentration calculations) and applying Equal Difference Method (EDM) [69] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid density and concentration at lattice site
in	gradient	Density and concentration gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluids at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionBGKGuo()

```
int fSiteFluidCollisionBGKGuo (double * startpos,
                               double * sitespeed,
                               double * omega,
                               double * rho,
                               double * bodyforce)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for mildly compressible fluids and applying Guo [49] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidCollisionBGKGuoLishchuk()

```
int fSiteFluidCollisionBGKGuoLishchuk (double * startpos,
                                       double * sitespeed,
                                       double * omega,
                                       double * rho,
                                       double * bodyforce,
                                       double * phaseindex)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying Guo [49] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidCollisionBGKGuoLishchukLocal()

```
int fSiteFluidCollisionBGKGuoLishchukLocal (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * rho,
                                            double * bodyforce,
                                            double * phaseindex,
                                            int threed)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying Guo [49] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i \hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidCollisionBGKGuoSwiftOneFluid()

```
int fSiteFluidCollisionBGKGuoSwiftOneFluid (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * rho,
                                            double * gradient,
                                            double * bodyforce,
                                            double T)
```

Applies single relaxation time (BGK) collisions to one fluid at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for a compressible fluid undergoing Swift free-energy interactions and applying Guo [49] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequency for fluid at lattice site
in	rho	Macroscopic fluid density at lattice site
in	gradient	Density gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluid at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionBGKGuoSwiftTwoFluid()

```
int fSiteFluidCollisionBGKGuoSwiftTwoFluid (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * rho,
                                            double * gradient,
                                            double * bodyforce,
                                            double T)
```

Applies single relaxation time (BGK) collisions to two fluid at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for two compressible fluids undergoing Swift free-energy interactions (for fluid density and concentration calculations) and applying Guo [49] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid density and concentration at lattice site
in	gradient	Density and concentration gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluids at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionBGKHe()

```
int fSiteFluidCollisionBGKHe (double * startpos,
                              double * sitespeed,
                              double * omega,
                              double * rho,
                              double * bodyforce)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for mildly compressible fluids and applying He [54] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidCollisionBGKHeLishchuk()

```
int fSiteFluidCollisionBGKHeLishchuk (double * startpos,
                                      double * sitespeed,
                                      double * omega,
                                      double * rho,
                                      double * bodyforce,
                                      double * phaseindex)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying He [54] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidCollisionBGKHeLishchukLocal()

```
int fSiteFluidCollisionBGKHeLishchukLocal (double * startpos,
                                           double * sitespeed,
                                           double * omega,
                                           double * rho,
                                           double * bodyforce,
                                           double * phaseindex,
                                           int threed)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying He [54] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i \hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidCollisionBGKHeSwiftOneFluid()

```
int fSiteFluidCollisionBGKHeSwiftOneFluid (double * startpos,
                                           double * sitespeed,
                                           double * omega,
                                           double * rho,
                                           double * gradient,
                                           double * bodyforce,
                                           double T)
```

Applies single relaxation time (BGK) collisions to one fluid at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for a compressible fluid undergoing Swift free-energy interactions and applying He [54] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequency for fluid at lattice site
in	rho	Macroscopic fluid density at lattice site
in	gradient	Density gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluid at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionBGKHeSwiftTwoFluid()

```
int fSiteFluidCollisionBGKHeSwiftTwoFluid (double * startpos,
                                           double * sitespeed,
                                           double * omega,
                                           double * rho,
                                           double * gradient,
                                           double * bodyforce,
                                           double T)
```

Applies single relaxation time (BGK) collisions to two fluid at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for two compressible fluids undergoing Swift free-energy interactions (for fluid density and concentration calculations) and applying He [54] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid density and concentration at lattice site
in	gradient	Density and concentration gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluids at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionBGKLishchuk()

```
int fSiteFluidCollisionBGKLishchuk (double * startpos,
                                    double * sitespeed,
                                    double * omega,
                                    double * rho,
                                    double * bodyforce,
                                    double * phaseindex)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying standard (Martys-Chen) [91] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidCollisionBGKLishchukLocal()

```
int fSiteFluidCollisionBGKLishchukLocal (double * startpos,
                                         double * sitespeed,
                                         double * omega,
                                         double * rho,
                                         double * bodyforce,
                                         double * phaseindex,
                                         int threed)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying standard (Martys-Chen) [91] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i\hat{e}_i - c_s^2\mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidCollisionBGKSwiftOneFluid()

```
int fSiteFluidCollisionBGKSwiftOneFluid (double * startpos,
                                         double * sitespeed,
                                         double * omega,
                                         double * rho,
                                         double * gradient,
                                         double * bodyforce,
                                         double T)
```

Applies single relaxation time (BGK) collisions to one fluid at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for a compressible fluid undergoing Swift free-energy interactions and applying standard (Martys-Chen) [91] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequency for fluid at lattice site
in	rho	Macroscopic fluid density at lattice site
in	gradient	Density gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluid at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionBGKSwiftTwoFluid()

```
int fSiteFluidCollisionBGKSwiftTwoFluid (double * startpos,
                                         double * sitespeed,
                                         double * omega,
                                         double * rho,
                                         double * gradient,
                                         double * bodyforce,
                                         double T)
```

Applies single relaxation time (BGK) collisions to two fluid at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for two compressible fluids undergoing Swift free-energy interactions (for fluid density and concentration calculations) and applying standard (Martys-Chen) [91] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid density and concentration at lattice site
in	gradient	Density and concentration gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluids at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidIncomCollisionBGK()

```
int fSiteFluidIncomCollisionBGK (double * startpos,
                                 double * sitespeed,
                                 double * omega,
                                 double * rho,
                                 double * bodyforce)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for fully incompressible fluids and applying standard (Martys-Chen) [91] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidIncomCollisionBGKEDM()

```
int fSiteFluidIncomCollisionBGKEDM (double * startpos,
                                    double * sitespeed,
                                    double * omega,
                                    double * rho,
                                    double * bodyforce)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for fully incompressible fluids and applying Equal Difference Method (EDM) [69] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidIncomCollisionBGKEDMLishchuk()

```
int fSiteFluidIncomCollisionBGKEDMLishchuk (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * rho,
                                            double * bodyforce,
                                            double * phaseindex)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying Equal Difference Method (EDM) [69] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidIncomCollisionBGKEDMLishchukLocal()

```
int fSiteFluidIncomCollisionBGKEDMLishchukLocal (double * startpos,
                                                 double * sitespeed,
                                                 double * omega,
                                                 double * rho,
                                                 double * bodyforce,
                                                 double * phaseindex,
                                                 int thread)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying Equal Difference Method (EDM) [69] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i\hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidIncomCollisionBGKGuo()

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

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for fully incompressible fluids and applying Guo [49] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidIncomCollisionBGKGuoLishchuk()

```
int fSiteFluidIncomCollisionBGKGuoLishchuk (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * rho,
                                            double * bodyforce,
                                            double * phaseindex)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying Guo [49] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidIncomCollisionBGKGuoLishchukLocal()

```
int fSiteFluidIncomCollisionBGKGuoLishchukLocal (double * startpos,
                                                 double * sitespeed,
                                                 double * omega,
                                                 double * rho,
                                                 double * bodyforce,
                                                 double * phaseindex,
                                                 int threed)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying Guo [49] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i \hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidIncomCollisionBGKHe()

```
int fSiteFluidIncomCollisionBGKHe (double * startpos,
                                   double * sitespeed,
                                   double * omega,
                                   double * rho,
                                   double * bodyforce)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for fully incompressible fluids and applying He [54] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidIncomCollisionBGKHeLishchuk()

```
int fSiteFluidIncomCollisionBGKHeLishchuk (double * startpos,
                                           double * sitespeed,
                                           double * omega,
                                           double * rho,
                                           double * bodyforce,
                                           double * phaseindex)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying He [54] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidIncomCollisionBGKHeLishchukLocal()

```
int fSiteFluidIncomCollisionBGKHeLishchukLocal (double * startpos,
                                                double * sitespeed,
                                                double * omega,
                                                double * rho,
                                                double * bodyforce,
                                                double * phaseindex,
                                                int threed)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying He [54] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i\hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidIncomCollisionBGKLishchuk()

```
int fSiteFluidIncomCollisionBGKLishchuk (double * startpos,
                                         double * sitespeed,
                                         double * omega,
                                         double * rho,
                                         double * bodyforce,
                                         double * phaseindex)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying standard (Martys-Chen) [91] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidIncomCollisionBGKLishchukLocal()

```
int fSiteFluidIncomCollisionBGKLishchukLocal (double * startpos,
                                              double * sitespeed,
                                              double * omega,
                                              double * rho,
                                              double * bodyforce,
                                              double * phaseindex,
                                              int threed)
```

Applies single relaxation time (BGK) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying standard (Martys-Chen) [91] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129] :

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i\hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	p hasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteSoluteCollisionBGK()

```
int fSiteSoluteCollisionBGK (double * startpos, double * sitespeed)
```

Applies single relaxation time (BGK) collisions to all solutes at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for diffusion of solutes.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site

fSiteThermalCollisionBGK()

```
int fSiteThermalCollisionBGK (double * startpos, double * sitespeed)
```

Applies single relaxation time (BGK) collisions to temperature field at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for heat diffusion.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site

5.14 lbpTRT.cpp

Module with routines for two relaxation time (TRT) collisions.

Applies collisions to grid points using a two relaxation time (TRT) scheme [42] on each fluid. This scheme is based on splitting distribution functions (including local equilibrium values) into symmetric $f_i^+ = \frac{1}{2}(f_i + f_j)$ and anti-symmetric $f_i^+ = \frac{1}{2}(f_i - f_j)$ parts, using values in conjugate links to construct them, i.e. $\hat{e}_j = -\hat{e}_i$. The collision operator is defined as:

$$f_i\left(\vec{x}, t^+\right) = f_i\left(\vec{x}, t\right) - \frac{f_i^+\left(\vec{x}, t\right) - f_i^{eq,+}\left(\rho\left(\vec{x}, t\right), \vec{u}\left(\vec{x}, t\right)\right)}{\tau^+} - \frac{f_i^-\left(\vec{x}, t\right) - f_i^{eq,-}\left(\rho\left(\vec{x}, t\right), \vec{u}\left(\vec{x}, t\right)\right)}{\tau^-}.$$

where τ^+ and τ^- are respectively the symmetric and anti-symmetric relaxation times. The symmetric relaxation time controls fluid kinematic viscosity in an identical fashion to the single relaxation time τ used in BGK collisions, while the anti-symmetric relaxation time is chosen to enhance numerical stability: this can be specified using a 'magic number':

$$\Lambda_{eo} = \Lambda_e \Lambda_o = \left(\tau^+ - \frac{1}{2}\right)\left(\tau^- - \frac{1}{2}\right).$$

To simplify the implementation, the collsions can be re-expressed using full distribution functions:

$$f_i\left(\vec{x}, t^+\right) = f_i\left(\vec{x}, t\right) - \frac{f_i\left(\vec{x}, t\right) - f_i^{eq}\left(\rho\left(\vec{x}, t\right), \vec{u}\left(\vec{x}, t\right)\right)}{\tau_p} - \frac{f_j\left(\vec{x}, t\right) - f_j^{eq}\left(\rho\left(\vec{x}, t\right), \vec{u}\left(\vec{x}, t\right)\right)}{\tau_n},$$

where $\tau_p = \frac{2\tau^+\tau^-}{\tau^+ + \tau^-}$ and $\tau_n = \frac{2\tau^+\tau^-}{\tau^+ - \tau^-}$.

To apply forces to each fluid, one of four options can be applied. The standard (Martys-Chen) [91] force scheme applies a modified velocity for calculating local equilibrium moments:

$$\vec{v} = \vec{u} + \frac{\tau_f \vec{F} \Delta t}{\rho},$$

while the Equal Difference Method (EDM) [69] applies an additional forcing term that can be calculated as a difference in local equilibrium distribution functions:

$$F_i = f_i^{eq}\left(\rho, \vec{u} + \frac{\vec{F}\Delta t}{\rho}\right) - f_i^{eq}\left(\rho, \vec{u}\right).$$

The Guo scheme [49] both adjusts the velocity for local equilibrium distribution functions to $\vec{v} = \vec{u} + \frac{\vec{F}\Delta t}{2\rho}$ and includes the following forcing term for TRT collisions [115]:

$$F_i = \left(1 - \frac{1}{2\tau_{f,p}}\right) w_i \left[\frac{\hat{e}_i - \vec{v}}{c_s^2} + \frac{\hat{e}_i \cdot \vec{v}}{c_s^4}\hat{e}_i\right] \cdot \vec{F} - \frac{1}{2\tau_{f,n}} w_j \left[\frac{\hat{e}_j - \vec{v}}{c_s^2} + \frac{\hat{e}_j \cdot \vec{v}}{c_s^4}\hat{e}_j\right] \cdot \vec{F}$$

and the He scheme [54] uses the same adjusted velocity and the following forcing term for TRT collisions:

$$F_i = \left(1 - \frac{1}{2\tau_{f,p}}\right) \frac{f_i^{eq}}{\rho c_s^2}\left(\hat{e}_i - \vec{v}\right) \cdot \vec{F} - \frac{1}{2\tau_{f,p}} \frac{f_j^{eq}}{\rho c_s^2}\left(\hat{e}_j - \vec{v}\right) \cdot \vec{F}.$$

5.14.1 Functions

- `double` *fTRTOmegaAntisymmetric()*

  Calculates antisymmetric relaxation frequency for TRT collisions.

- `int` *fSiteFluidCollisionTRT()*

  Applies TRT collisions to all compressible fluids at a given lattice site with standard forcing.

- `int` *fSiteFluidIncomCollisionTRT()*

  Applies TRT collisions to all incompressible fluids at a given lattice site with standard forcing.

- `int` *fSiteFluidCollisionTRTEDM()*

  Applies TRT collisions to all compressible fluids at a given lattice site with EDM forcing.

- `int` *fSiteFluidIncomCollisionTRTEDM()*

  Applies TRT collisions to all incompressible fluids at a given lattice site with EDM forcing.

- `int` *fSiteFluidCollisionTRTGuo()*

  Applies TRT collisions to all compressible fluids at a given lattice site with Guo forcing.

- `int` *fSiteFluidIncomCollisionTRTGuo()*

  Applies TRT collisions to all incompressible fluids at a given lattice site with Guo forcing.

- `int` *fSiteFluidCollisionTRTHe()*

  Applies TRT collisions to all compressible fluids at a given lattice site with He forcing.

- `int` *fSiteFluidIncomCollisionTRTHe()*

  Applies TRT collisions to all incompressible fluids at a given lattice site with He forcing.

- `int` *fSiteFluidCollisionTRTLishchuk()*

  Applies TRT collisions to all compressible fluids at a given lattice site with standard forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidIncomCollisionTRTLishchuk()*

  Applies TRT collisions to all incompressible fluids at a given lattice site with standard forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidCollisionTRTEDMLishchuk()*

  Applies TRT collisions to all compressible fluids at a given lattice site with EDM forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidIncomCollisionTRTEDMLishchuk()*

  Applies TRT collisions to all incompressible fluids at a given lattice site with EDM forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidCollisionTRTGuoLishchuk()*

  Applies TRT collisions to all compressible fluids at a given lattice site with Guo forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidIncomCollisionTRTGuoLishchuk()*

  Applies TRT collisions to all incompressible fluids at a given lattice site with Guo forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidCollisionTRTHeLishchuk()*

  Applies TRT collisions to all compressible fluids at a given lattice site with He forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidIncomCollisionTRTHeLishchuk()*

  Applies TRT collisions to all incompressible fluids at a given lattice site with He forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidCollisionTRTLishchukLocal()*

  Applies TRT collisions to all compressible fluids at a given lattice site with standard forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidIncomCollisionTRTLishchukLocal()*

  Applies TRT collisions to all incompressible fluids at a given lattice site with standard forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidCollisionTRTEDMLishchukLocal()*

  Applies TRT collisions to all compressible fluids at a given lattice site with EDM forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidIncomCollisionTRTEDMLishchukLocal()*

  Applies TRT collisions to all incompressible fluids at a given lattice site with EDM forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidCollisionTRTGuoLishchukLocal()*

  Applies TRT collisions to all compressible fluids at a given lattice site with Guo forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidIncomCollisionTRTGuoLishchukLocal()*

  Applies TRT collisions to all incompressible fluids at a given lattice site with Guo forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidCollisionTRTHeLishchukLocal()*

  Applies TRT collisions to all compressible fluids at a given lattice site with He forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidIncomCollisionTRTHeLishchukLocal()*

  Applies TRT collisions to all incompressible fluids at a given lattice site with He forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidCollisionTRTSwiftOneFluid()*

  Applies TRT collisions to one compressible fluid at a given lattice site with standard forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionTRTSwiftTwoFluid()*

  Applies TRT collisions to two compressible fluids at a given lattice site with standard forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionTRTEDMSwiftOneFluid()*

  Applies TRT collisions to one compressible fluid at a given lattice site with EDM forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionTRTEDMSwiftTwoFluid()*

  Applies TRT collisions to two compressible fluids at a given lattice site with EDM forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionTRTGuoSwiftOneFluid()*

  Applies TRT collisions to one compressible fluid at a given lattice site with Guo forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionTRTGuoSwiftTwoFluid()*

  Applies TRT collisions to two compressible fluids at a given lattice site with Guo forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionTRTHeSwiftOneFluid()*

  Applies TRT collisions to one compressible fluid at a given lattice site with He forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionTRTHeSwiftTwoFluid()*

  Applies TRT collisions to two compressible fluids at a given lattice site with He forcing and Swift free-energy interactions.

- `int` *fCollisionTRT()*

  Applies collision steps for all fluids using TRT scheme with standard forcing, and solutes and temperature fields using BGK scheme..

- `int` *fCollisionTRTEDM()*

  Applies collision steps for all fluids using TRT scheme with EDM forcing, and solutes and temperature fields using BGK scheme..

- `int` *fCollisionTRTGuo()*

  Applies collision steps for all fluids using TRT scheme with Guo forcing, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTHe()*

  Applies collision steps for all fluids using TRT scheme with He forcing, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTShanChen()*

  Applies collision steps for all fluids using TRT scheme with standard forcing for Shan-Chen interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTEDMShanChen()*

  Applies collision steps for all fluids using TRT scheme with EDM forcing for Shan-Chen interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTGuoShanChen()*

  Applies collision steps for all fluids using TRT scheme with Guo forcing for Shan-Chen interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTHeShanChen()*

  Applies collision steps for all fluids using TRT scheme with He forcing for Shan-Chen interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTLishchuk()*

  Applies collision steps for all fluids using TRT scheme with standard forcing and Lishchuk interactions provided as interfacial forces, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTEDMLishchuk()*

  Applies collision steps for all fluids using TRT scheme with EDM forcing and Lishchuk interactions provided as interfacial forces, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTGuoLishchuk()*

  Applies collision steps for all fluids using TRT scheme with Guo forcing and Lishchuk interactions provided as interfacial forces, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTHeLishchuk()*

  Applies collision steps for all fluids using TRT scheme with He forcing and Lishchuk interactions provided as interfacial forces, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTLishchukLocal()*

  Applies collision steps for all fluids using TRT scheme with standard forcing and Lishchuk interactions provided as an additional forcing term, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTEDMLishchukLocal()*

  Applies collision steps for all fluids using TRT scheme with EDM forcing and Lishchuk interactions provided as an additional forcing term, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTGuoLishchukLocal()*

  Applies collision steps for all fluids using TRT scheme with Guo forcing and Lishchuk interactions provided as an additional forcing term, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTHeLishchukLocal()*

  Applies collision steps for all fluids using TRT scheme with He forcing and Lishchuk interactions provided as an additional forcing term, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTSwift()*

  Applies collision steps for all fluids using TRT scheme with standard forcing for Swift free-energy interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTEDMSwift()*

  Applies collision steps for all fluids using TRT scheme with EDM forcing for Swift free-energy interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTGuoSwift()*

  Applies collision steps for all fluids using TRT scheme with Guo forcing for Swift free-energy interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionTRTHeSwift()*

  Applies collision steps for all fluids using TRT scheme with He forcing for Swift free-energy interactions, and solutes and temperature fields using BGK scheme.

5.14.2 Function Documentation

fCollisionTRT()

```
int fCollisionTRT ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with standard (Martys-Chen) [91] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the standard values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}.$$

fCollisionTRTEDM()

```
int fCollisionTRTEDM ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with Equal Difference Method (EDM) [69] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the standard values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}.$$

fCollisionTRTEDMLishchuk()

```
int fCollisionTRTEDMLishchuk ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with Equal Difference Method (EDM) [69] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using the main forcing scheme: this approach can be used with the original Lishchuk and Lishchuk-Spencer interaction models.

fCollisionTRTEDMLishchukLocal()

```
int fCollisionTRTEDMLishchukLocal ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with Equal Difference Method (EDM) [69] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using separate forcing terms: this approach can be used with the Lishchuk 'Spencer tensor' and local Lishchuk interaction models.

fCollisionTRTEDMShanChen()

```
int fCollisionTRTEDMShanChen ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with with Equal Difference Method (EDM) [69] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the following values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} \frac{f_i^a \hat{e}_i}{\tau_f^a}}{\sum_{i,a} \frac{f_i^a}{\tau_f^a}}.$$

fCollisionTRTEDMSwift()

```
int fCollisionTRTEDMSwift ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with Equal DIfference Method (EDM) [69] forcing and Swift free-energy interactions (enacted using modified local equilibrium distribution functions to incorporate density and concentration gradients), and all solutes and any temperature field using single relaxation time BGK collisions.

fCollisionTRTGuo()

```
int fCollisionTRTGuo ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with Guo [49] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the standard values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}.$$

fCollisionTRTGuoLishchuk()

```
int fCollisionTRTGuoLishchuk ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with Guo [49] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using the main forcing scheme: this approach can be used with the original Lishchuk and Lishchuk-Spencer interaction models.

fCollisionTRTGuoLishchukLocal()

```
int fCollisionTRTGuoLishchukLocal ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with Guo [49] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using separate forcing terms: this approach can be used with the Lishchuk 'Spencer tensor' and local Lishchuk interaction models.

fCollisionTRTGuoShanChen()

```
int fCollisionTRTGuoShanChen ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with with Guo [49] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the following values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} \frac{f_i^a \hat{e}_i}{\tau_f^a}}{\sum_{i,a} \frac{f_i^a}{\tau_f^a}}.$$

fCollisionTRTGuoSwift()

```
int fCollisionTRTGuoSwift ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with Guo [49] forcing and Swift free-energy interactions (enacted using modified local equilibrium distribution functions to incorporate density and concentration gradients), and all solutes and any temperature field using single relaxation time BGK collisions.

fCollisionTRTHe()

```
int fCollisionTRTHe ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with He [54] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the standard values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}.$$

fCollisionTRTHeLishchuk()

```
int fCollisionTRTHeLishchuk ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with He [54] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using the main forcing scheme: this approach can be used with the original Lishchuk and Lishchuk-Spencer interaction models.

fCollisionTRTHeLishchukLocal()

```
int fCollisionTRTHeLishchukLocal ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with He [54] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using separate forcing terms: this approach can be used with the Lishchuk ‘Spencer tensor’ and local Lishchuk interaction models.

fCollisionTRTHeShanChen()

```
int fCollisionTRTHeShanChen ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with with He [54] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the following values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} \frac{f_i^a \hat{e}_i}{\tau_f^a}}{\sum_{i,a} \frac{f_i^a}{\tau_f^a}}.$$

fCollisionTRTHeSwift()

```
int fCollisionTRTHeSwift ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with He [54] forcing and Swift free-energy interactions (enacted using modified local equilibrium distribution functions to incorporate density and concentration gradients), and all solutes and any temperature field using single relaxation time BGK collisions.

fCollisionTRTLishchuk()

```
int fCollisionTRTLishchuk ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with standard (Martys-Chen) [91] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using the main forcing scheme: this approach can be used with the original Lishchuk and Lishchuk-Spencer interaction models.

fCollisionTRTLishchukLocal()

```
int fCollisionTRTLishchukLocal ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with standard (Martys-Chen) [91] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using separate forcing terms: this approach can be used with the Lishchuk 'Spencer tensor' and local Lishchuk interaction models.

fCollisionTRTShanChen()

```
int fCollisionTRTShanChen ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with standard (Martys-Chen) [91] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the following values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} \frac{f_i^a \hat{e}_i}{\tau_f^a}}{\sum_{i,a} \frac{f_i^a}{\tau_f^a}}.$$

fCollisionTRTSwift()

```
int fCollisionTRTSwift ()
```

Loops through all available lattice sites and applies collisions to all fluids using two relaxation time (TRT) collisions with standard (Martys-Chen) [91] forcing and Swift free-energy interactions (enacted using modified local equilibrium distribution functions to incorporate density and concentration gradients), and all solutes and any temperature field using single relaxation time BGK collisions.

fSiteFluidCollisionTRT()

```
int fSiteFluidCollisionTRT (double * startpos,
                            double * sitespeed,
                            double * omega,
                            double * rho,
                            double * bodyforce)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for mildly compressible fluids and applying standard (Martys-Chen) [91] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidCollisionTRTEDM()

```
int fSiteFluidCollisionTRTEDM (double * startpos,
                               double * sitespeed,
                               double * omega,
                               double * rho,
                               double * bodyforce)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for mildly compressible fluids and applying Equal Difference Method (EDM) [69] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidCollisionTRTEDMLishchuk()

```
int fSiteFluidCollisionTRTEDMLishchuk (double * startpos,
                                       double * sitespeed,
                                       double * omega,
                                       double * rho,
                                       double * bodyforce,
                                       double * phaseindex)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying Equal Difference Method (EDM) [69] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidCollisionTRTEDMLishchukLocal()

```
int fSiteFluidCollisionTRTEDMLishchukLocal (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * rho,
                                            double * bodyforce,
                                            double * phaseindex,
                                            int threed)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying Equal Difference Method (EDM) [69] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i\beta^{ab}g^{ab}\rho^a\rho^b}{c_s^4\rho^3\tau_f\Delta t}\left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i\hat{e}_i - c_s^2\mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidCollisionTRTEDMSwiftOneFluid()

```
int fSiteFluidCollisionTRTEDMSwiftOneFluid (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * rho,
                                            double * gradient,
                                            double * bodyforce,
                                            double T)
```

Applies two relaxation time (TRT) collisions to one fluid at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for a compressible fluid undergoing Swift free-energy interactions and applying Equal Difference Method (EDM) [69] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluid at lattice site
in	rho	Macroscopic fluid density at lattice site
in	gradient	Density gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluid at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionTRTEDMSwiftTwoFluid()

```
int fSiteFluidCollisionTRTEDMSwiftTwoFluid (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * rho,
                                            double * gradient,
                                            double * bodyforce,
                                            double T)
```

Applies two relaxation time (TRT) collisions to two fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for two compressible fluids undergoing Swift free-energy interactions (for fluid density and concentration calculations) and applying Equal Difference Method (EDM) [69] forcing. Collisions of distribution functions for fluid concentration are carried out using a BGK single relaxation time scheme.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid density and concentration at lattice site
in	gradient	Density and concentration gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluids at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionTRTGuo()

```
int fSiteFluidCollisionTRTGuo (double * startpos,
                               double * sitespeed,
                               double * omega,
                               double * rho,
                               double * bodyforce)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for mildly compressible fluids and applying Guo [49] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidCollisionTRTGuoLishchuk()

```
int fSiteFluidCollisionTRTGuoLishchuk (double * startpos,
                                       double * sitespeed,
                                       double * omega,
                                       double * rho,
                                       double * bodyforce,
                                       double * phaseindex)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying Guo [49] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25] :

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidCollisionTRTGuoLishchukLocal()

```
int fSiteFluidCollisionTRTGuoLishchukLocal (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * rho,
                                            double * bodyforce,
                                            double * phaseindex,
                                            int threed)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying Guo [49] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab} \hat{n}_{ab} - \mathbf{I} \right) : \left(\hat{e}_i \hat{e}_i - c_s^2 \mathbf{I} \right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidCollisionTRTGuoSwiftOneFluid()

```
int fSiteFluidCollisionTRTGuoSwiftOneFluid (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * rho,
                                            double * gradient,
                                            double * bodyforce,
                                            double T)
```

Applies two relaxation time (TRT) collisions to one fluid at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for a compressible fluid undergoing Swift free-energy interactions and applying Guo [49] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequency for fluid at lattice site
in	rho	Macroscopic fluid density at lattice site
in	gradient	Density gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluid at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionTRTGuoSwiftTwoFluid()

```
int fSiteFluidCollisionTRTGuoSwiftTwoFluid (double * startpos,
                                             double * sitespeed,
                                             double * omega,
                                             double * rho,
                                             double * gradient,
                                             double * bodyforce,
                                             double T)
```

Applies two relaxation time (TRT) collisions to two fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for two compressible fluids undergoing Swift free-energy interactions (for fluid density and concentration calculations) and applying Guo [49] forcing. Collisions of distribution functions for fluid concentration are carried out using a BGK single relaxation time scheme.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid density and concentration at lattice site
in	gradient	Density and concentration gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluids at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionTRTHe()

```
int fSiteFluidCollisionTRTHe (double * startpos,
                               double * sitespeed,
                               double * omega,
                               double * rho,
                               double * bodyforce)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for mildly compressible fluids and applying He [54] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidCollisionTRTHeLishchuk()

```
int fSiteFluidCollisionTRTHeLishchuk (double * startpos,
                                       double * sitespeed,
                                       double * omega,
                                       double * rho,
                                       double * bodyforce,
                                       double * phaseindex)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying He [54] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidCollisionTRTHeLishchukLocal()

```
int fSiteFluidCollisionTRTHeLishchukLocal (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * rho,
                                            double * bodyforce,
                                            double * phaseindex,
                                            int threed)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying He [54] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i\hat{e}_i - c_s^2\mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidCollisionTRTHeSwiftOneFluid()

```
int fSiteFluidCollisionTRTHeSwiftOneFluid (double * startpos,
                                           double * sitespeed,
                                           double * omega,
                                           double * rho,
                                           double * gradient,
                                           double * bodyforce,
                                           double T)
```

Applies two relaxation time (TRT) collisions to one fluid at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for a compressible fluid undergoing Swift free-energy interactions and applying He [54] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequency for fluid at lattice site
in	rho	Macroscopic fluid density at lattice site
in	gradient	Density gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluid at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionTRTHeSwiftTwoFluid()

```
int fSiteFluidCollisionTRTHeSwiftTwoFluid (double * startpos,
                                           double * sitespeed,
                                           double * omega,
                                           double * rho,
                                           double * gradient,
                                           double * bodyforce,
                                           double T)
```

Applies two relaxation time (TRT) collisions to two fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for two compressible fluids undergoing Swift free-energy interactions (for fluid density and concentration calculations) and applying He [54] forcing. Collisions of distribution functions for fluid concentration are carried out using a BGK single relaxation time scheme.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid density and concentration at lattice site
in	gradient	Density and concentration gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluids at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionTRTLishchuk()

```
int fSiteFluidCollisionTRTLishchuk (double * startpos,
                                    double * sitespeed,
                                    double * omega,
                                    double * rho,
                                    double * bodyforce,
                                    double * phaseindex)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying standard (Martys-Chen) [91] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidCollisionTRTLishchukLocal()

```
int fSiteFluidCollisionTRTLishchukLocal (double * startpos,
                                         double * sitespeed,
                                         double * omega,
                                         double * rho,
                                         double * bodyforce,
                                         double * phaseindex,
                                         int threed)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying standard (Martys-Chen) [91] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i\hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidCollisionTRTSwiftOneFluid()

```
int fSiteFluidCollisionTRTSwiftOneFluid (double * startpos,
                                         double * sitespeed,
                                         double * omega,
                                         double * rho,
                                         double * gradient,
                                         double * bodyforce,
                                         double T)
```

Applies two relaxation time (TRT) collisions to one fluid at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for a compressible fluid undergoing Swift free-energy interactions and applying standard (Martys-Chen) [91] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequency for fluid at lattice site
in	rho	Macroscopic fluid density at lattice site
in	gradient	Density gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluid at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionTRTSwiftTwoFluid()

```
int fSiteFluidCollisionTRTSwiftTwoFluid (double * startpos,
                                         double * sitespeed,
                                         double * omega,
                                         double * rho,
                                         double * gradient,
                                         double * bodyforce,
                                         double T)
```

Applies two relaxation time (TRT) collisions to two fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for two compressible fluids undergoing Swift free-energy interactions (for fluid density and concentration calculations) and applying standard (Martys-Chen) [91] forcing. Collisions of distribution functions for fluid concentration are carried out using a BGK single relaxation time scheme.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid density and concentration at lattice site
in	gradient	Density and concentration gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluids at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidIncomCollisionTRT()

```
int fSiteFluidIncomCollisionTRT (double * startpos,
                                 double * sitespeed,
                                 double * omega,
                                 double * rho,
                                 double * bodyforce)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for fully incompressible fluids and applying standard (Martys-Chen) [91] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidIncomCollisionTRTEDM()

```
int fSiteFluidIncomCollisionTRTEDM (double * startpos,
                                    double * sitespeed,
                                    double * omega,
                                    double * rho,
                                    double * bodyforce)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for fully incompressible fluids and applying Equal Difference Method (EDM) [69] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidIncomCollisionTRTEDMLishchuk()

```
int fSiteFluidIncomCollisionTRTEDMLishchuk (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * rho,
                                            double * bodyforce,
                                            double * phaseindex)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying Equal Difference Method (EDM) [69] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidIncomCollisionTRTEDMLishchukLocal()

```
int fSiteFluidIncomCollisionTRTEDMLishchukLocal (double * startpos,
                                                 double * sitespeed,
                                                 double * omega,
                                                 double * rho,
                                                 double * bodyforce,
                                                 double * phaseindex,
                                                 int threed)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying Equal Difference Method (EDM) [69] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i\hat{e}_i - c_s^2\mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidIncomCollisionTRTGuo()

```
int fSiteFluidIncomCollisionTRTGuo (double * startpos,
                                    double * sitespeed,
                                    double * omega,
                                    double * rho,
                                    double * bodyforce)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for fully incompressible fluids and applying Guo [49] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidIncomCollisionTRTGuoLishchuk()

```
int fSiteFluidIncomCollisionTRTGuoLishchuk (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * rho,
                                            double * bodyforce,
                                            double * phaseindex)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying Guo [49] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidIncomCollisionTRTGuoLishchukLocal()

```
int fSiteFluidIncomCollisionTRTGuoLishchukLocal (double * startpos,
                                                 double * sitespeed,
                                                 double * omega,
                                                 double * rho,
                                                 double * bodyforce,
                                                 double * phaseindex,
                                                 int threed)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying Guo [49] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i\hat{e}_i - c_s^2\mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidIncomCollisionTRTHe()

```
int fSiteFluidIncomCollisionTRTHe (double * startpos,
                                   double * sitespeed,
                                   double * omega,
                                   double * rho,
                                   double * bodyforce)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for fully incompressible fluids and applying He [54] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidIncomCollisionTRTHeLishchuk()

```
int fSiteFluidIncomCollisionTRTHeLishchuk (double * startpos,
                                           double * sitespeed,
                                           double * omega,
                                           double * rho,
                                           double * bodyforce,
                                           double * phaseindex)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying He [54] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidIncomCollisionTRTHeLishchukLocal()

```
int fSiteFluidIncomCollisionTRTHeLishchukLocal (double * startpos,
                                                double * sitespeed,
                                                double * omega,
                                                double * rho,
                                                double * bodyforce,
                                                double * phaseindex,
                                                int threed)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying He [54] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i \hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidIncomCollisionTRTLishchuk()

```
int fSiteFluidIncomCollisionTRTLishchuk (double * startpos,
                                         double * sitespeed,
                                         double * omega,
                                         double * rho,
                                         double * bodyforce,
                                         double * phaseindex)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying standard (Martys-Chen) [91] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidIncomCollisionTRTLishchukLocal()

```
int fSiteFluidIncomCollisionTRTLishchukLocal (double * startpos,
                                               double * sitespeed,
                                               double * omega,
                                               double * rho,
                                               double * bodyforce,
                                               double * phaseindex,
                                               int threed)
```

Applies two relaxation time (TRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying standard (Martys-Chen) [91] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab} \hat{n}_{ab} - \mathbf{I} \right) : \left(\hat{e}_i \hat{e}_i - c_s^2 \mathbf{I} \right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	(Symmetric) relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fTRTOmegaAntisymmetric()

```
double fTRTOmegaAntisymmetric (double omegasymmetric, double magic)
```

Calculates and returns the antisymmetric relaxation frequency $\omega_f^- = \frac{1}{\tau_f^-}$ for two relaxation time (TRT) collisions using the symmetric relaxation frequency $\omega_f^+ = \frac{1}{\tau_f^+}$ and the TRT 'magic number' Λ_{eo}.

Parameters

in	omegasymmetric	Symmetric relaxation frequency ω_f^+
in	magic	TRT 'magic number' Λ_{eo}

5.15 lbpMRT.cpp

Module with routines for moment-based multiple relaxation time (MRT) collisions. (Header file available as lbpMRT.hpp.)

Applies collisions to grid points using a (raw) moment-based multiple relaxation time (MRT) scheme on each fluid. This scheme starts by defining a number of moments of distribution functions involving some combination of the lattice vectors $\hat{e}_i$. A vector of moments can be obtained by transforming a vector of distribution functions using a transformation matrix $\mathbf{T}$:

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

The equilibrium moments $\vec{M}^{eq}$ can be found by using the same transformation matrix with the local equilibrium distribution functions $\vec{f}^{eq}$: these can be expressed as functions of fluid density and velocity. (The fluid density and components of momentum make up some of the moments used in this collision scheme.)

A collision matrix $\mathbf{\Lambda}$ is then used to collide the moments:

$$\vec{M}\left(\vec{x}, t^{+}\right) = \vec{M}\left(\vec{x}, t\right) - \mathbf{\Lambda}\left(\vec{M}\left(\vec{x}, t\right) - \vec{M}^{eq}\left(\rho\left(\vec{x}, t\right), \vec{u}\left(\vec{x}, t\right)\right)\right).$$

Since the collision matrix is diagonal, the collisions can be carried out individually on each moment, i.e.

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

The post-collisional moments are then transformed back by using inverses of the transformation matrices to produce post-collisional distribution functions.

To apply forces to each fluid, one of four options can be applied. The standard (Martys-Chen) [91] force scheme applies a modified velocity for calculating local equilibrium moments:

$$\vec{v} = \vec{u} + \frac{\tau_f \vec{F} \Delta t}{\rho},$$

while the Equal Difference Method (EDM) [69] applies an additional forcing term that can be calculated as a difference in local equilibrium distribution functions:

$$F_i = f_i^{eq}\left(\rho, \vec{u} + \frac{\vec{F}\Delta t}{\rho}\right) - f_i^{eq}\left(\rho, \vec{u}\right).$$

The Guo [49] and He schemes [54] both adjust the velocity for local equilibrium distribution functions to $\vec{v} = \vec{u} + \frac{\vec{F}\Delta t}{2\rho}$ and include the following forcing terms for MRT collisions as additional moment terms for collisions:

$$\Delta\vec{M} = \left(1 - \frac{1}{2}\mathbf{\Lambda}\right)\vec{S}^m \Delta t,$$

where $\vec{S}^m$ are the source terms for the required forcing scheme transformed into moments [104].

MRT collision schemes exist for D2Q9 [73], D3Q15, D3Q19 [159] and D3Q27 [134] lattices: variations also exist for fully incompressible fluids and Swift free-energy interactions.

5.15.1 Functions

- `int` *fGetMomentEquilibriumF()*

  Calculates local equilibrium moments for multiple relaxation time (MRT) collisions of compressible fluids.

- `int` *fGetMomentEquilibriumFIncom()*

  Calculates local equilibrium moments for multiple relaxation time (MRT) collisions of incompressible fluids.

- `int` *fGetMomentEquilibriumFSwiftOneFluid()*

  Calculates local equilibrium moments for multiple relaxation time (MRT) collisions of one fluid with Swift free-energy interactions.

- `int` *fGetMomentEquilibriumFSwiftTwoFluid()*

  Calculates local equilibrium moments for multiple relaxation time (MRT) collisions of two fluids with Swift free-energy interactions.

- `int` *fGetMomentForceGuo()*

  Calculates Guo forcing terms in terms of moments for multiple relaxation time (MRT) collisions.

- `int` *fGetMomentForceHe()*

  Calculates He forcing terms in terms of moments for multiple relaxation time (MRT) collisions.

- `int` *fGetMRTCollide()*

  Calculates the collision frequencies for multiple relaxation time (MRT) collisions based on provided relaxation frequencies.

- `int` *fSiteFluidCollisionMRT()*

  Applies MRT collisions to all compressible fluids at a given lattice site with standard forcing.

- `int` *fSiteFluidIncomCollisionMRT()*

  Applies MRT collisions to all incompressible fluids at a given lattice site with standard forcing.

- `int` *fSiteFluidCollisionMRTEDM()*

  Applies MRT collisions to all compressible fluids at a given lattice site with EDM forcing.

- `int` *fSiteFluidIncomCollisionMRTEDM()*

  Applies MRT collisions to all incompressible fluids at a given lattice site with EDM forcing.

- `int` *fSiteFluidCollisionMRTGuo()*

  Applies MRT collisions to all compressible fluids at a given lattice site with Guo forcing.

- `int` *fSiteFluidIncomCollisionMRTGuo()*

  Applies MRT collisions to all incompressible fluids at a given lattice site with Guo forcing.

- `int` *fSiteFluidCollisionMRTHe()*

  Applies MRT collisions to all compressible fluids at a given lattice site with He forcing.

- `int` *fSiteFluidIncomCollisionMRTHe()*

  Applies MRT collisions to all incompressible fluids at a given lattice site with He forcing.

- `int` *fSiteFluidCollisionMRTLishchuk()*

  Applies MRT collisions to all compressible fluids at a given lattice site with standard forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidIncomCollisionMRTLishchuk()*

  Applies MRT collisions to all incompressible fluids at a given lattice site with standard forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidCollisionMRTEDMLishchuk()*

  Applies MRT collisions to all compressible fluids at a given lattice site with EDM forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidIncomCollisionMRTEDMLishchuk()*

  Applies MRT collisions to all incompressible fluids at a given lattice site with EDM forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidCollisionMRTGuoLishchuk()*

  Applies MRT collisions to all compressible fluids at a given lattice site with Guo forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidIncomCollisionMRTGuoLishchuk()*

  Applies MRT collisions to all incompressible fluids at a given lattice site with Guo forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidCollisionMRTHeLishchuk()*

  Applies MRT collisions to all compressible fluids at a given lattice site with He forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidIncomCollisionMRTHeLishchuk()*

  Applies MRT collisions to all incompressible fluids at a given lattice site with He forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidCollisionMRTLishchukLocal()*

  Applies MRT collisions to all compressible fluids at a given lattice site with standard forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidIncomCollisionMRTLishchukLocal()*

  Applies MRT collisions to all incompressible fluids at a given lattice site with standard forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidCollisionMRTEDMLishchukLocal()*

  Applies MRT collisions to all compressible fluids at a given lattice site with EDM forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidIncomCollisionMRTEDMLishchukLocal()*

  Applies MRT collisions to all incompressible fluids at a given lattice site with EDM forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidCollisionMRTGuoLishchukLocal()*

  Applies MRT collisions to all compressible fluids at a given lattice site with Guo forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidIncomCollisionMRTGuoLishchukLocal()*

  Applies MRT collisions to all incompressible fluids at a given lattice site with Guo forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidCollisionMRTHeLishchukLocal()*

  Applies MRT collisions to all compressible fluids at a given lattice site with He forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidIncomCollisionMRTHeLishchukLocal()*

  Applies MRT collisions to all incompressible fluids at a given lattice site with He forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidCollisionMRTSwiftOneFluid()*

  Applies MRT collisions to one compressible fluid at a given lattice site with standard forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionMRTSwiftTwoFluid()*

  Applies MRT collisions to two compressible fluids at a given lattice site with standard forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionMRTEDMSwiftOneFluid()*

  Applies MRT collisions to one compressible fluid at a given lattice site with EDM forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionMRTEDMSwiftTwoFluid()*

  Applies MRT collisions to two compressible fluids at a given lattice site with EDM forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionMRTGuoSwiftOneFluid()*

  Applies MRT collisions to one compressible fluid at a given lattice site with Guo forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionMRTGuoSwiftTwoFluid()*

  Applies MRT collisions to two compressible fluids at a given lattice site with Guo forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionMRTHeSwiftOneFluid()*

  Applies MRT collisions to one compressible fluid at a given lattice site with He forcing and Swift free-energy interactions.

- `int` *fSiteFluidCollisionMRTHeSwiftTwoFluid()*

  Applies MRT collisions to two compressible fluids at a given lattice site with He forcing and Swift free-energy interactions.

- `int` *fCollisionMRT()*

  Applies collision steps for all fluids using MRT scheme with standard forcing, and solutes and temperature fields using BGK scheme..

- `int` *fCollisionMRTEDM()*

  Applies collision steps for all fluids using MRT scheme with EDM forcing, and solutes and temperature fields using BGK scheme..

- `int` *fCollisionMRTGuo()*

  Applies collision steps for all fluids using MRT scheme with Guo forcing, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTHe()*

  Applies collision steps for all fluids using MRT scheme with He forcing, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTShanChen()*

  Applies collision steps for all fluids using MRT scheme with standard forcing for Shan-Chen interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTEDMShanChen()*

  Applies collision steps for all fluids using MRT scheme with EDM forcing for Shan-Chen interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTGuoShanChen()*

  Applies collision steps for all fluids using MRT scheme with Guo forcing for Shan-Chen interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTHeShanChen()*

  Applies collision steps for all fluids using MRT scheme with He forcing for Shan-Chen interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTLishchuk()*

  Applies collision steps for all fluids using MRT scheme with standard forcing and Lishchuk interactions provided as interfacial forces, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTEDMLishchuk()*

  Applies collision steps for all fluids using MRT scheme with EDM forcing and Lishchuk interactions provided as interfacial forces, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTGuoLishchuk()*

  Applies collision steps for all fluids using MRT scheme with Guo forcing and Lishchuk interactions provided as interfacial forces, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTHeLishchuk()*

  Applies collision steps for all fluids using MRT scheme with He forcing and Lishchuk interactions provided as interfacial forces, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTLishchukLocal()*

  Applies collision steps for all fluids using MRT scheme with standard forcing and Lishchuk interactions provided as an additional forcing term, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTEDMLishchukLocal()*

  Applies collision steps for all fluids using MRT scheme with EDM forcing and Lishchuk interactions provided as an additional forcing term, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTGuoLishchukLocal()*

  Applies collision steps for all fluids using MRT scheme with Guo forcing and Lishchuk interactions provided as an additional forcing term, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTHeLishchukLocal()*

  Applies collision steps for all fluids using MRT scheme with He forcing and Lishchuk interactions provided as an additional forcing term, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTSwift()*

  Applies collision steps for all fluids using MRT scheme with standard forcing for Swift free-energy interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTEDMSwift()*

  Applies collision steps for all fluids using MRT scheme with EDM forcing for Swift free-energy interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTGuoSwift()*

  Applies collision steps for all fluids using MRT scheme with Guo forcing for Swift free-energy interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionMRTHeSwift()*

  Applies collision steps for all fluids using MRT scheme with He forcing for Swift free-energy interactions, and solutes and temperature fields using BGK scheme.

5.15.2 Function Documentation

fCollisionMRT()

```
int fCollisionMRT ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with standard (Martys-Chen) [91] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the standard values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}.$$

fCollisionMRTEDM()

```
int fCollisionMRTEDM ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with Equal Difference Method (EDM) [69] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the standard values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}.$$

fCollisionMRTEDMLishchuk()

```
int fCollisionMRTEDMLishchuk ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with Equal Difference Method (EDM) [69] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using the main forcing scheme: this approach can be used with the original Lishchuk and Lishchuk-Spencer interaction models.

fCollisionMRTEDMLishchukLocal()

```
int fCollisionMRTEDMLishchukLocal ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with Equal Difference Method (EDM) [69] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using separate forcing terms: this approach can be used with the Lishchuk 'Spencer tensor' and local Lishchuk interaction models.

fCollisionMRTEDMShanChen()

```
int fCollisionMRTEDMShanChen ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with with Equal Difference Method (EDM) [69] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the following values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} \frac{f_i^a \hat{e}_i}{\tau_f^a}}{\sum_{i,a} \frac{f_i^a}{\tau_f^a}}.$$

fCollisionMRTEDMSwift()

```
int fCollisionMRTEDMSwift ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with Equal DIfference Method (EDM) [69] forcing and Swift free-energy interactions (enacted using modified local equilibrium distribution functions to incorporate density and concentration gradients), and all solutes and any temperature field using single relaxation time BGK collisions.

fCollisionMRTGuo()

```
int fCollisionMRTGuo ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with Guo [49] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the standard values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}.$$

fCollisionMRTGuoLishchuk()

```
int fCollisionMRTGuoLishchuk ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with Guo [49] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using the main forcing scheme: this approach can be used with the original Lishchuk and Lishchuk-Spencer interaction models.

fCollisionMRTGuoLishchukLocal()

```
int fCollisionMRTGuoLishchukLocal ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with Guo [49] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using separate forcing terms: this approach can be used with the Lishchuk ‘Spencer tensor’ and local Lishchuk interaction models.

fCollisionMRTGuoShanChen()

```
int fCollisionMRTGuoShanChen ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with with Guo [49] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the following values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} \frac{f_i^a \hat{e}_i}{\tau_f^a}}{\sum_{i,a} \frac{f_i^a}{\tau_f^a}}.$$

fCollisionMRTGuoSwift()

```
int fCollisionMRTGuoSwift ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with Guo [49] forcing and Swift free-energy interactions (enacted using modified local equilibrium distribution functions to incorporate density and concentration gradients), and all solutes and any temperature field using single relaxation time BGK collisions.

fCollisionMRTHe()

```
int fCollisionMRTHe ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with He [54] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the standard values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}.$$

fCollisionMRTHeLishchuk()

```
int fCollisionMRTHeLishchuk ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with He [54] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using the main forcing scheme: this approach can be used with the original Lishchuk and Lishchuk-Spencer interaction models.

fCollisionMRTHeLishchukLocal()

```
int fCollisionMRTHeLishchukLocal ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with He [54] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using separate forcing terms: this approach can be used with the Lishchuk 'Spencer tensor' and local Lishchuk interaction models.

fCollisionMRTHeShanChen()

```
int fCollisionMRTHeShanChen ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with with He [54] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the following values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} \frac{f_i^a \hat{e}_i}{\tau_f^a}}{\sum_{i,a} \frac{f_i^a}{\tau_f^a}}.$$

fCollisionMRTHeSwift()

```
int fCollisionMRTHeSwift ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with He [54] forcing and Swift free-energy interactions (enacted using modified local equilibrium distribution functions to incorporate density and concentration gradients), and all solutes and any temperature field using single relaxation time BGK collisions.

fCollisionMRTLishchuk()

```
int fCollisionMRTLishchuk ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with standard (Martys-Chen) [91] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using the main forcing scheme: this approach can be used with the original Lishchuk and Lishchuk-Spencer interaction models.

fCollisionMRTLishchukLocal()

```
int fCollisionMRTLishchukLocal ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with standard (Martys-Chen) [91] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using separate forcing terms: this approach can be used with the Lishchuk 'Spencer tensor' and local Lishchuk interaction models.

fCollisionMRTShanChen()

```
int fCollisionMRTShanChen ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with standard (Martys-Chen) [91] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the following values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} \frac{f_i^a \hat{e}_i}{\tau_f^a}}{\sum_{i,a} \frac{f_i^a}{\tau_f^a}}.$$

fCollisionMRTSwift()

```
int fCollisionMRTSwift ()
```

Loops through all available lattice sites and applies collisions to all fluids using multiple relaxation time (MRT) collisions with standard (Martys-Chen) [91] forcing and Swift free-energy interactions (enacted using modified local equilibrium distribution functions to incorporate density and concentration gradients), and all solutes and any temperature field using single relaxation time BGK collisions.

fGetMomentEquilibriumF()

```
int fGetMomentEquilibriumF (double * meq,
                            double * p,
                            double rho)
```

Calculates local equilibrium values for moments required for multiple relaxation time (MRT) collisions, based on transforming the local equilibrium distribution function for mildly compressible fluids. The exact values for each moment depends on the lattice scheme in use - for D2Q9 [73], D3Q15, D3Q19 [159] and D3Q27 [134] - but these include the fluid density and components of fluid momentum.

Parameters

out	meq	Local equilibrium central moments
in	p	Fluid momentum at lattice site
in	rho	Fluid density at lattice site

fGetMomentEquilibriumFIncom()

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

Calculates local equilibrium values for moments required for multiple relaxation time (MRT) collisions, based on transforming the local equilibrium distribution function for fully incompressible fluids. The exact values for each moment depends on the lattice scheme in use - for D2Q9 [73], D3Q15, D3Q19 [159] and D3Q27 [134] - but these include the fluid density and components of fluid momentum.

Parameters

out	meq	Local equilibrium central moments
in	p	Fluid momentum at lattice site
in	rho	Variable fluid density at lattice site
in	rho0	Constant fluid density at lattice site

fGetMomentEquilibriumFSwiftOneFluid()

```
int fGetMomentEquilibriumFSwiftOneFluid (double * meq,
                                         double * p,
                                         double rho,
                                         double pb,
                                         double lambda,
                                         double * grad)
```

Calculates local equilibrium values for moments required for multiple relaxation time (MRT) collisions, based on transforming the local equilibrium distribution function for a single fluid with Swift free-energy interactions. The exact values for each moment depends on the lattice scheme in use - for D2Q9 [73], D3Q15 and D3Q19 [159] - but these include the fluid density, components of fluid momentum and bulk pressure (which depends on the equation of state being applied).

Parameters

out	meq	Local equilibrium central moments
in	p	Fluid momentum at lattice site
in	rho	Fluid density at lattice site
in	pb	Bulk pressure at lattice site (based on equation of state)
in	lambda	Galilean invariance parameter for lattice site (based on equation of state)
in	grad	First-order and second-order gradients of fluid density at lattice site

fGetMomentEquilibriumFSwiftTwoFluid()

```
int fGetMomentEquilibriumFSwiftTwoFluid (double * meq,
                                          double * p,
                                          double rho,
                                          double phi,
                                          double pb,
                                          double mu,
                                          double lambda,
                                          double * grad)
```

Calculates local equilibrium values for moments required for multiple relaxation time (MRT) collisions, based on transforming the local equilibrium distribution function for two fluids with Swift free-energy interactions. The exact values for each moment depends on the lattice scheme in use - for D2Q9 [73], D3Q15 and D3Q19 [159] - but these include the fluid density, components of fluid momentum, bulk pressure (which depends on the equation of state being applied) and chemical potential (which depends on the free energy functional). The local equilibrium distribution functions for concentration are also calculated and included in the output array for BGK single relaxation time collisions of these distribution functions.

Parameters

out	meq	Local equilibrium central moments for density, local equilibrium distribution functions for concentration
in	p	Fluid momentum at lattice site
in	rho	Fluid density at lattice site
in	phi	Fluid concentration at lattice site
in	pb	Bulk pressure at lattice site (based on equation of state)
in	mu	Chemical potential at lattice site (based on free energy functional)
in	lambda	Galilean invariance parameter for lattice site (based on equation of state)
in	grad	First-order and second-order gradients of fluid density and concentration at lattice site

fGetMomentForceGuo()

```
int fGetMomentForceGuo (double * source,
                        double * v,
                        double * force)
```

Calculates moment-based Guo forcing terms for use in multiple relaxation time (MRT) collisions, as obtained by applying the transformation matrix to Guo source terms [104]:

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

i.e. $\vec{S}^m = \mathbf{T}\vec{S}$. The exact form of moment-based forcing terms will depend on the lattice scheme in use (D2Q9, D3Q15, D3Q19 or D3Q27) but require both forces and the velocity for each lattice point to calculate.

Parameters

out	source	Moment-based Guo forcing terms $\vec{S}^m$
in	v	Force-corrected fluid velocity at lattice point
in	force	Forces acting at lattice point

fGetMomentForceHe()

```
int fGetMomentForceHe (double * source,
                       double * v,
                       double * force)
```

Calculates moment-based He forcing terms for use in multiple relaxation time (MRT) collisions, as obtained by applying the transformation matrix to He source terms [104]:

$$S_i = \frac{f_i^{eq}}{\rho c_s^2} (\hat{e}_i - \vec{v}) \cdot \vec{F}$$

i.e. $\vec{S}^m = \mathbf{T}\vec{S}$. The exact form of moment-based forcing terms will depend on the lattice scheme in use (D2Q9, D3Q15, D3Q19 or D3Q27) but require both forces and the velocity for each lattice point to calculate.

Parameters

out	source	Moment-based He forcing terms $\vec{S}^m$
in	v	Force-corrected fluid velocity at lattice point
in	force	Forces acting at lattice point

fGetMRTCollide()

```
int fGetMRTCollide (double * collide,
                    double omegashear,
                    double omegabulk)
```

Calculates the diagonal collision matrix $\mathbf{\Lambda}$ used in multiple relaxation time (MRT) collisions. This subroutine requires inputs for two relaxation frequencies at each lattice point and fluid: the main relaxation frequency $\omega = \tau_f^{-1}$ and the bulk relaxation frequency $\omega_b = \tau_{f,bulk}^{-1}$. Other system-wide relaxation frequencies - used as tuneable parameters to enhance numerical stability of calculations - are assigned in this subroutine. Moments for fluid density and momentum have their relaxation frequencies set to 1 to ensure these properties are conserved.

Parameters

out	collide	Diagonal of relaxation frequencies for MRT collision matrix
in	omegashear	Relaxation frequency for fluid (giving kinetic viscosity)
in	omegabulk	Bulk relaxation frequency for fluid (giving bulk viscosity)

fSiteFluidCollisionMRT()

```
int fSiteFluidCollisionMRT (double * startpos,
                            double * sitespeed,
                            double * omega,
                            double * omegabulk,
                            double * rho,
                            double * bodyforce)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for mildly compressible fluids and applying standard (Martys-Chen) [91] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidCollisionMRTEDM()

```
int fSiteFluidCollisionMRTEDM (double * startpos,
                               double * sitespeed,
                               double * omega,
                               double * omegabulk,
                               double * rho,
                               double * bodyforce)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for mildly compressible fluids and applying Equal Difference Method (EDM) [69] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidCollisionMRTEDMLishchuk()

```
int fSiteFluidCollisionMRTEDMLishchuk (double * startpos,
                                       double * sitespeed,
                                       double * omega,
                                       double * omegabulk,
                                       double * rho,
                                       double * bodyforce,
                                       double * phaseindex)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying Equal Difference Method (EDM) [69] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidCollisionMRTEDMLishchukLocal()

```
int fSiteFluidCollisionMRTEDMLishchukLocal (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * omegabulk,
                                            double * rho,
                                            double * bodyforce,
                                            double * phaseindex,
                                            int threed)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying Equal Difference Method (EDM) [69] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i\hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegab-ulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidCollisionMRTEDMSwiftOneFluid()

```
int fSiteFluidCollisionMRTEDMSwiftOneFluid (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * omegabulk,
                                            double * rho,
                                            double * gradient,
                                            double * bodyforce,
                                            double T)
```

Applies multiple relaxation time (MRT) collisions to one fluid at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for a compressible fluid undergoing Swift free-energy interactions and applying Equal Difference Method (EDM) [69] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequency for fluid at lattice site
in	omegabulk	Bulk relaxation frequency for fluid at lattice site
in	rho	Macroscopic fluid density at lattice site
in	gradient	Density gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluid at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionMRTEDMSwiftTwoFluid()

```
int fSiteFluidCollisionMRTEDMSwiftTwoFluid (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * omegabulk,
                                            double * rho,
                                            double * gradient,
                                            double * bodyforce,
                                            double T)
```

Applies multiple relaxation time (MRT) collisions to two fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for two compressible fluids undergoing Swift free-energy interactions (for fluid density and concentration calculations) and applying Equal Difference Method (EDM) [69] forcing. Collisions of distribution functions for fluid concentration are carried out using a BGK single relaxation time scheme.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid density and concentration at lattice site
in	gradient	Density and concentration gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluids at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionMRTGuo()

```
int fSiteFluidCollisionMRTGuo (double * startpos,
                               double * sitespeed,
                               double * omega,
                               double * omegabulk,
                               double * rho,
                               double * bodyforce)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for mildly compressible fluids and applying Guo [49] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidCollisionMRTGuoLishchuk()

```
int fSiteFluidCollisionMRTGuoLishchuk (double * startpos,
                                       double * sitespeed,
                                       double * omega,
                                       double * omegabulk,
                                       double * rho,
                                       double * bodyforce,
                                       double * phaseindex)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying Guo [49] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidCollisionMRTGuoLishchukLocal()

```
int fSiteFluidCollisionMRTGuoLishchukLocal (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * omegabulk,
                                            double * rho,
                                            double * bodyforce,
                                            double * phaseindex,
                                            int threed)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying Guo [49] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i \hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegab-ulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidCollisionMRTGuoSwiftOneFluid()

```
int fSiteFluidCollisionMRTGuoSwiftOneFluid (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * omegabulk,
                                            double * rho,
                                            double * gradient,
                                            double * bodyforce,
                                            double T)
```

Applies multiple relaxation time (MRT) collisions to one fluid at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for a compressible fluid undergoing Swift free-energy interactions and applying Guo [49] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequency for fluid at lattice site
in	omegabulk	Bulk relaxation frequency for fluid at lattice site
in	rho	Macroscopic fluid density at lattice site
in	gradient	Density gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluid at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionMRTGuoSwiftTwoFluid()

```
int fSiteFluidCollisionMRTGuoSwiftTwoFluid (double * startpos,
                                             double * sitespeed,
                                             double * omega,
                                             double * omegabulk,
                                             double * rho,
                                             double * gradient,
                                             double * bodyforce,
                                             double T)
```

Applies multiple relaxation time (MRT) collisions to two fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for two compressible fluids undergoing Swift free-energy interactions (for fluid density and concentration calculations) and applying Guo [49] forcing. Collisions of distribution functions for fluid concentration are carried out using a BGK single relaxation time scheme.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid density and concentration at lattice site
in	gradient	Density and concentration gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluids at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionMRTHe()

```
int fSiteFluidCollisionMRTHe (double * startpos,
                              double * sitespeed,
                              double * omega,
                              double * omegabulk,
                              double * rho,
                              double * bodyforce)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for mildly compressible fluids and applying He [54] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidCollisionMRTHeLishchuk()

```
int fSiteFluidCollisionMRTHeLishchuk (double * startpos, double * sitespeed,␣
↪double * omega, double * omegabulk, double * rho, double * bodyforce, double *␣
↪phaseindex)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying He [54] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidCollisionMRTHeLishchukLocal()

```
int fSiteFluidCollisionMRTHeLishchukLocal (double * startpos,
                                           double * sitespeed,
                                           double * omega,
                                           double * omegabulk,
                                           double * rho,
                                           double * bodyforce,
                                           double * phaseindex,
                                           int threed)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying He [54] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i\hat{e}_i - c_s^2\mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegab-ulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidCollisionMRTHeSwiftOneFluid()

```
int fSiteFluidCollisionMRTHeSwiftOneFluid (double * startpos,
                                           double * sitespeed,
                                           double * omega,
                                           double * omegabulk,
                                           double * rho,
                                           double * gradient,
                                           double * bodyforce,
                                           double T)
```

Applies multiple relaxation time (MRT) collisions to one fluid at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for a compressible fluid undergoing Swift free-energy interactions and applying He [54] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequency for fluid at lattice site
in	omegabulk	Bulk relaxation frequency for fluid at lattice site
in	rho	Macroscopic fluid density at lattice site
in	gradient	Density gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluid at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionMRTHeSwiftTwoFluid()

```
int fSiteFluidCollisionMRTHeSwiftTwoFluid (double * startpos,
                                           double * sitespeed,
                                           double * omega,
                                           double * omegabulk,
                                           double * rho,
                                           double * gradient,
                                           double * bodyforce,
                                           double T)
```

Applies multiple relaxation time (MRT) collisions to two fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for two compressible fluids undergoing Swift free-energy interactions (for fluid density and concentration calculations) and applying He [54] forcing. Collisions of distribution functions for fluid concentration are carried out using a BGK single relaxation time scheme.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluid at lattice site
in	rho	Macroscopic fluid density and concentration at lattice site
in	gradient	Density and concentration gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluids at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionMRTLishchuk()

```
int fSiteFluidCollisionMRTLishchuk (double * startpos,
                                    double * sitespeed,
                                    double * omega,
                                    double * omegabulk,
                                    double * rho,
                                    double * bodyforce,
                                    double * phaseindex)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying standard (Martys-Chen) [91] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidCollisionMRTLishchukLocal()

```
int fSiteFluidCollisionMRTLishchukLocal (double * startpos,
                                         double * sitespeed,
                                         double * omega,
                                         double * omegabulk,
                                         double * rho,
                                         double * bodyforce,
                                         double * phaseindex,
                                         int threed)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying standard (Martys-Chen) [91] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i \hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegab-ulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidCollisionMRTSwiftOneFluid()

```
int fSiteFluidCollisionMRTSwiftOneFluid (double * startpos,
                                         double * sitespeed,
                                         double * omega,
                                         double * omegabulk,
                                         double * rho,
                                         double * gradient,
                                         double * bodyforce,
                                         double T)
```

Applies multiple relaxation time (MRT) collisions to one fluid at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for a compressible fluid undergoing Swift free-energy interactions and applying standard (Martys-Chen) [91] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequency for fluid at lattice site
in	omegabulk	Bulk relaxation frequency for fluid at lattice site
in	rho	Macroscopic fluid density at lattice site
in	gradient	Density gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluid at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidCollisionMRTSwiftTwoFluid()

```
int fSiteFluidCollisionMRTSwiftTwoFluid (double * startpos,
                                         double * sitespeed,
                                         double * omega,
                                         double * omegabulk,
                                         double * rho,
                                         double * gradient,
                                         double * bodyforce,
                                         double T)
```

Applies multiple relaxation time (MRT) collisions to two fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution function for two compressible fluids undergoing Swift free-energy interactions (for fluid density and concentration calculations) and applying standard (Martys-Chen) [91] forcing. Collisions of distribution functions for fluid concentration are carried out using a BGK single relaxation time scheme.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid density and concentration at lattice site
in	gradient	Density and concentration gradients (first and second order) at lattice site
in	bodyforce	Forces to apply to fluids at lattice site
in	T	Temperature at lattice site (used for calculating bulk pressure of fluid)

fSiteFluidIncomCollisionMRT()

```
int fSiteFluidIncomCollisionMRT (double * startpos,
                                 double * sitespeed,
                                 double * omega,
                                 double * omegabulk,
                                 double * rho,
                                 double * bodyforce)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for fully incompressible fluids and applying standard (Martys-Chen) [91] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidIncomCollisionMRTEDM()

```
int fSiteFluidIncomCollisionMRTEDM (double * startpos,
                                    double * sitespeed,
                                    double * omega,
                                    double * omegabulk,
                                    double * rho,
                                    double * bodyforce)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for fully incompressible fluids and applying Equal Difference Method (EDM) [69] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidIncomCollisionMRTEDMLishchuk()

```
int fSiteFluidIncomCollisionMRTEDMLishchuk (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * omegabulk,
                                            double * rho,
                                            double * bodyforce,
                                            double * phaseindex)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying Equal Difference Method (EDM) [69] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidIncomCollisionMRTEDMLishchukLocal()

```
int fSiteFluidIncomCollisionMRTEDMLishchukLocal (double * startpos,
                                                 double * sitespeed,
                                                 double * omega,
                                                 double * omegabulk,
                                                 double * rho,
                                                 double * bodyforce,
                                                 double * phaseindex,
                                                 int threed)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying Equal Difference Method (EDM) [69] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i\hat{e}_i - c_s^2\mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegab-ulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidIncomCollisionMRTGuo()

```
int fSiteFluidIncomCollisionMRTGuo (double * startpos,
                                    double * sitespeed,
                                    double * omega,
                                    double * omegabulk,
                                    double * rho,
                                    double * bodyforce)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for fully incompressible fluids and applying Guo [49] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidIncomCollisionMRTGuoLishchuk()

```
int fSiteFluidIncomCollisionMRTGuoLishchuk (double * startpos,
                                            double * sitespeed,
                                            double * omega,
                                            double * omegabulk,
                                            double * rho,
                                            double * bodyforce,
                                            double * phaseindex)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying Guo [49] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidIncomCollisionMRTGuoLishchukLocal()

```
int fSiteFluidIncomCollisionMRTGuoLishchukLocal (double * startpos,
                                                 double * sitespeed,
                                                 double * omega,
                                                 double * omegabulk,
                                                 double * rho,
                                                 double * bodyforce,
                                                 double * phaseindex,
                                                 int threed)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying Guo [49] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i \hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegab-ulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidIncomCollisionMRTHe()

```
int fSiteFluidIncomCollisionMRTHe (double * startpos,
                                   double * sitespeed,
                                   double * omega,
                                   double * omegabulk,
                                   double * rho,
                                   double * bodyforce)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the local equilibrium distribution functions for fully incompressible fluids and applying He [54] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidIncomCollisionMRTHeLishchuk()

```
int fSiteFluidIncomCollisionMRTHeLishchuk (double * startpos,
                                           double * sitespeed,
                                           double * omega,
                                           double * omegabulk,
                                           double * rho,
                                           double * bodyforce,
                                           double * phaseindex)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying He [54] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidIncomCollisionMRTHeLishchukLocal()

```
int fSiteFluidIncomCollisionMRTHeLishchukLocal (double * startpos,
                                                double * sitespeed,
                                                double * omega,
                                                double * omegabulk,
                                                double * rho,
                                                double * bodyforce,
                                                double * phaseindex,
                                                int threed)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying He [54] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab} \hat{n}_{ab} - \mathbf{I} \right) : \left(\hat{e}_i \hat{e}_i - c_s^2 \mathbf{I} \right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegab-ulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidIncomCollisionMRTLishchuk()

```
int fSiteFluidIncomCollisionMRTLishchuk (double * startpos,
                                          double * sitespeed,
                                          double * omega,
                                          double * omegabulk,
                                          double * rho,
                                          double * bodyforce,
                                          double * phaseindex)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying standard (Martys-Chen) [91] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidIncomCollisionMRTLishchukLocal()

```
int fSiteFluidIncomCollisionMRTLishchukLocal (double * startpos,
                                              double * sitespeed,
                                              double * omega,
                                              double * omegabulk,
                                              double * rho,
                                              double * bodyforce,
                                              double * phaseindex,
                                              int threed)
```

Applies multiple relaxation time (MRT) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for fully incompressible fluids, applying standard (Martys-Chen) [91] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i \hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegab-ulk	Bulk relaxation frequencies for fluids at lattice site
in	rho	Macroscopic variable fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

5.16 lbpCLBE.cpp

Module with routines for cascaded lattice Boltzmann equation (CLBE) collisions. (Header file available as lbpCLBE.hpp.)

Applies collisions to grid points using a central moment multiple relaxation time scheme, known as cascaded LBE (CLBE) [38], on each fluid. This scheme starts by defining a number of central moments of distribution functions, i.e.

$$\tilde{M}_{pqr} = \sum_i f_i \left(e_{i,x} - u_x\right)^p \left(e_{i,y} - u_y\right)^q \left(e_{i,z} - u_z\right)^r$$

which can be related to raw moments, $M_{pqr} = \sum_i f_i e_{i,x}^p e_{i,y}^q e_{i,z}^r$. In a similar fashion to multiple relaxation time (MRT) collision schemes, the distribution functions can be transformed into central moments using transformation matrices:

$$\vec{\tilde{M}} = \mathbf{N}\mathbf{T}\vec{f}$$

where the matrix $\mathbf{T}$ transforms distribution functions into raw moments and the lower triangular matrix $\mathbf{N}$ transforms raw moments into central moments and depends upon the fluid velocity at the lattice grid point. A collision matrix $\mathbf{\Lambda}$ that is mostly diagonal (aside from a block diagonal used for second-order moments) is then used to collide the central moments:

$$\vec{\tilde{M}}\left(\vec{x},t^{+}\right)=\vec{\tilde{M}}\left(\vec{x},t\right)-\mathbf{\Lambda}\left(\vec{\tilde{M}}\left(\vec{x},t\right)-\vec{\tilde{M}}^{eq}\left(\rho\left(\vec{x},t\right),\vec{u}\left(\vec{x},t\right)\right)\right).$$

where $\vec{\tilde{M}}^{eq}$ - the local equilibrium central moments - are obtained by transforming the Maxwell-Boltzmann (general) local equilibrium distribution function. The post-collisional central moments are then transformed back by using inverses of the transformation matrices to produce post-collisional distribution functions.

To apply forces to each fluid, one of four options can be applied. The standard (Martys-Chen) [91] force scheme applies a modified velocity for calculating local equilibrium central moments:

$$\vec{v}=\vec{u}+\frac{\tau_f\vec{F}\Delta t}{\rho},$$

while the Equal Difference Method (EDM) [69] applies an additional forcing term that can be calculated as a difference in local equilibrium distribution functions:

$$F_i=f_i^{eq}\left(\rho,\vec{u}+\frac{\vec{F}\Delta t}{\rho}\right)-f_i^{eq}\left(\rho,\vec{u}\right).$$

although extended functions with third-order terms in velocity [84] are used for CLBE collisions. The Guo [49] and He schemes [54] both adjust the velocity for local equilibrium distribution functions to $\vec{v}=\vec{u}+\frac{\vec{F}\Delta t}{2\rho}$ and include the following forcing terms for CLBE collisions as additional terms for central moment collisions:

$$\Delta\tilde{M}_{pqr}=\left(1-\frac{1}{2}\omega_{pqr}\right)\tilde{S}_{pqr}\Delta t.$$

Guo forcing source terms can be obtained by transforming the standard terms with the transformation matrices, while those for He forcing are derived in a similar manner to local equilibrium central moments.

CLBE collision schemes exist for D2Q9 [35], D3Q19 and D3Q27 [33] lattices: no such schemes currently exist for D3Q15 lattices, fully incompressible fluids or Swift free-energy interactions.

5.16.1 Functions

- `int` *fGetCentralMomentEquilibriumF()*

  Calculates local equilibrium central moments for cascaded LBE (CLBE) collisions.

- `int` *fGetCentralMomentTransformMatrix()*

  Calculates full transform matrices (forward and inverse) for cascaded LBE (CLBE) collisions to obtain central moments from distribution functions and vice versa.

- `int` *fGetCentralMomentForceGuo()*

  Calculates Guo forcing terms in terms of central moments for cascaded LBE (CLBE) collisions.

- `int` *fGetCentralMomentForceHe()*

  Calculates Guo forcing terms in terms of central moments for cascaded LBE (CLBE) collisions.

- `int` *fGetCLBECollide()*

  Calculates the main diagonal collision matrix for cascaded LBE (CLBE) based on provided relaxation frequencies.

- `int` *fSiteFluidCollisionCLBE()*

  Applies CLBE collisions to all compressible fluids at a given lattice site with standard forcing.

- `int` *fSiteFluidCollisionCLBEEDM()*

  Applies CLBE collisions to all compressible fluids at a given lattice site with EDM forcing.

- `int` *fSiteFluidCollisionCLBEGuo()*

  Applies CLBE collisions to all compressible fluids at a given lattice site with Guo forcing.

- `int` *fSiteFluidCollisionCLBEHe()*

  Applies CLBE collisions to all compressible fluids at a given lattice site with He forcing.

- `int` *fSiteFluidCollisionCLBELishchuk()*

  Applies CLBE collisions to all compressible fluids at a given lattice site with standard forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidCollisionCLBEEDMLishchuk()*

  Applies CLBE collisions to all compressible fluids at a given lattice site with EDM forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidCollisionCLBEGuoLishchuk()*

  Applies CLBE collisions to all compressible fluids at a given lattice site with Guo forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidCollisionCLBEHeLishchuk()*

  Applies CLBE collisions to all compressible fluids at a given lattice site with He forcing and phase segregation when using Lishchuk interactions with calculated interfacial forces.

- `int` *fSiteFluidCollisionCLBELishchukLocal()*

  Applies CLBE collisions to all compressible fluids at a given lattice site with standard forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidCollisionCLBEEDMLishchukLocal()*

  Applies CLBE collisions to all compressible fluids at a given lattice site with EDM forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidCollisionCLBEGuoLishchukLocal()*

  Applies CLBE collisions to all compressible fluids at a given lattice site with Guo forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fSiteFluidCollisionCLBEHeLishchukLocal()*

  Applies CLBE collisions to all compressible fluids at a given lattice site with He forcing and phase segregation when using Lishchuk interactions with direct interfacial forcing.

- `int` *fCollisionCLBE()*

  Applies collision steps for all fluids using CLBE scheme with standard forcing, and solutes and temperature fields using BGK scheme..

- `int` *fCollisionCLBEEDM()*

  Applies collision steps for all fluids using CLBE scheme with EDM forcing, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionCLBEGuo()*

  Applies collision steps for all fluids using CLBE scheme with Guo forcing, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionCLBEHe()*

  Applies collision steps for all fluids using CLBE scheme with He forcing, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionCLBEShanChen()*

  Applies collision steps for all fluids using CLBE scheme with standard forcing for Shan-Chen interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionCLBEEDMShanChen()*

  Applies collision steps for all fluids using CLBE scheme with EDM forcing for Shan-Chen interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionCLBEGuoShanChen()*

  Applies collision steps for all fluids using CLBE scheme with Guo forcing for Shan-Chen interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionCLBEHeShanChen()*

  Applies collision steps for all fluids using CLBE scheme with He forcing for Shan-Chen interactions, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionCLBELishchuk()*

  Applies collision steps for all fluids using CLBE scheme with standard forcing and Lishchuk interactions provided as interfacial forces, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionCLBEEDMLishchuk()*

  Applies collision steps for all fluids using CLBE scheme with EDM forcing and Lishchuk interactions provided as interfacial forces, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionCLBEGuoLishchuk()*

  Applies collision steps for all fluids using CLBE scheme with Guo forcing and Lishchuk interactions provided as interfacial forces, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionCLBEHeLishchuk()*

  Applies collision steps for all fluids using CLBE scheme with He forcing and Lishchuk interactions provided as interfacial forces, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionCLBELishchukLocal()*

  Applies collision steps for all fluids using CLBE scheme with standard forcing and Lishchuk interactions provided as an additional forcing term, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionCLBEEDMLishchukLocal()*

  Applies collision steps for all fluids using CLBE scheme with EDM forcing and Lishchuk interactions provided as an additional forcing term, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionCLBEGuoLishchukLocal()*

  Applies collision steps for all fluids using CLBE scheme with Guo forcing and Lishchuk interactions provided as an additional forcing term, and solutes and temperature fields using BGK scheme.

- `int` *fCollisionCLBEHeLishchukLocal()*

  Applies collision steps for all fluids using CLBE scheme with He forcing and Lishchuk interactions provided as an additional forcing term, and solutes and temperature fields using BGK scheme.

5.16.2 Function Documentation

fCollisionCLBE()

```
int fCollisionCLBE ()
```

Loops through all available lattice sites and applies collisions to all fluids using cascaded LBE (CLBE) collisions with standard (Martys-Chen) [91] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the standard values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}.$$

fCollisionCLBEEDM()

```
int fCollisionCLBEEDM ()
```

Loops through all available lattice sites and applies collisions to all fluids using cascaded LBE (CLBE) collisions with Equal Difference Method (EDM) [69] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the standard values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}.$$

fCollisionCLBEEDMLishchuk()

```
int fCollisionCLBEEDMLishchuk ()
```

Loops through all available lattice sites and applies collisions to all fluids using cascaded LBE (CLBE) collisions with Equal Difference Method (EDM) [69] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using the main forcing scheme: this approach can be used with the original Lishchuk and Lishchuk-Spencer interaction models.

fCollisionCLBEEDMLishchukLocal()

```
int fCollisionCLBEEDMLishchukLocal ()
```

Loops through all available lattice sites and applies collisions to all fluids using cascaded LBE (CLBE) collisions with Equal Difference Method (EDM) [69] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using separate forcing terms: this approach can be used with the Lishchuk 'Spencer tensor' and local Lishchuk interaction models.

fCollisionCLBEEDMShanChen()

```
int fCollisionCLBEEDMShanChen ()
```

Loops through all available lattice sites and applies collisions to all fluids using cascaded LBE (CLBE) collisions with with Equal Difference Method (EDM) [69] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the following values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} \frac{f_i^a \hat{e}_i}{\tau_f^a}}{\sum_{i,a} \frac{f_i^a}{\tau_f^a}}.$$

fCollisionCLBEGuo()

```
int fCollisionCLBEGuo ()
```

Loops through all available lattice sites and applies collisions to all fluids using cascaded LBE (CLBE) collisions with Guo [49] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the standard values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}.$$

fCollisionCLBEGuoLishchuk()

```
int fCollisionCLBEGuoLishchuk ()
```

Loops through all available lattice sites and applies collisions to all fluids using cascaded LBE (CLBE) collisions with Guo [49] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using the main forcing scheme: this approach can be used with the original Lishchuk and Lishchuk-Spencer interaction models.

fCollisionCLBEGuoLishchukLocal()

```
int fCollisionCLBEGuoLishchukLocal ()
```

Loops through all available lattice sites and applies collisions to all fluids using cascaded LBE (CLBE) collisions with Guo [49] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using separate forcing terms: this approach can be used with the Lishchuk 'Spencer tensor' and local Lishchuk interaction models.

fCollisionCLBEGuoShanChen()

```
int fCollisionCLBEGuoShanChen ()
```

Loops through all available lattice sites and applies collisions to all fluids using cascaded LBE (CLBE) collisions with with Guo [49] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the following values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} \frac{f_i^a \hat{e}_i}{\tau_f^a}}{\sum_{i,a} \frac{f_i^a}{\tau_f^a}}.$$

fCollisionCLBEHe()

```
int fCollisionCLBEHe ()
```

Loops through all available lattice sites and applies collisions to all fluids using cascaded LBE (CLBE) collisions with He [54] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the standard values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} f_i^a \hat{e}_i}{\sum_{i,a} f_i^a}.$$

fCollisionCLBEHeLishchuk()

```
int fCollisionCLBEHeLishchuk ()
```

Loops through all available lattice sites and applies collisions to all fluids using cascaded LBE (CLBE) collisions with He [54] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using the main forcing scheme: this approach can be used with the original Lishchuk and Lishchuk-Spencer interaction models.

fCollisionCLBEHeLishchukLocal()

```
int fCollisionCLBEHeLishchukLocal ()
```

Loops through all available lattice sites and applies collisions to all fluids using cascaded LBE (CLBE) collisions with He [54] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using separate forcing terms: this approach can be used with the Lishchuk 'Spencer tensor' and local Lishchuk interaction models.

fCollisionCLBEHeShanChen()

```
int fCollisionCLBEHeShanChen ()
```

Loops through all available lattice sites and applies collisions to all fluids using cascaded LBE (CLBE) collisions with with He [54] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the following values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} \frac{f_i^a \hat{e}_i}{\tau_f^a}}{\sum_{i,a} \frac{f_i^a}{\tau_f^a}}.$$

fCollisionCLBELishchuk()

```
int fCollisionCLBELishchuk ()
```

Loops through all available lattice sites and applies collisions to all fluids using cascaded LBE (CLBE) collisions with standard (Martys-Chen) [91] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using the main forcing scheme: this approach can be used with the original Lishchuk and Lishchuk-Spencer interaction models.

fCollisionCLBELishchukLocal()

```
int fCollisionCLBELishchukLocal ()
```

Loops through all available lattice sites and applies collisions to all fluids using cascaded LBE (CLBE) collisions with standard (Martys-Chen) [91] forcing, achromatic fluid collisions and segregation, and all solutes and any temperature field using single relaxation time BGK collisions. The interfacial forces are applied using separate forcing terms: this approach can be used with the Lishchuk 'Spencer tensor' and local Lishchuk interaction models.

fCollisionCLBEShanChen()

```
int fCollisionCLBEShanChen ()
```

Loops through all available lattice sites and applies collisions to all fluids using cascaded LBE (CLBE) collisions with standard (Martys-Chen) [91] forcing, and all solutes and any temperature field using single relaxation time BGK collisions. This version of the collisions uses the following values for macroscopic fluid velocity at each site, i.e.

$$\vec{u} = \frac{\sum_{i,a} \frac{f_i^a \hat{e}_i}{\tau_f^a}}{\sum_{i,a} \frac{f_i^a}{\tau_f^a}}.$$

fGetCentralMomentEquilibriumF()

```
int fGetCentralMomentEquilibriumF (double * meq, double rho)
```

Calculates local equilibrium values for central moments required for cascaded LBE (CLBE) collisions, based on transforming the Maxwell-Boltzmann local equilibrium distribution function. The exact values for each moment depends on the lattice scheme in use - for D2Q9 [35], D3Q19 and D3Q27 [33] - but non-zero values are products of fluid density and even powers of the speed of sound.

Parameters

out	meq	Local equilibrium central moments
in	rho	Fluid density at lattice site

fGetCentralMomentForceGuo()

```
int fGetCentralMomentForceGuo (double * source,
                               double * v,
                               double * force)
```

Calculates central moment-based Guo forcing terms for use in cascaded LBE (CLBE) collisions, as obtained by applying the transformation and shift matrices to Guo source terms:

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

i.e. $\vec{\tilde{S}} = \mathbf{NT}\vec{S}$. The exact form of central moment forcing terms will depend on the lattice scheme in use (D2Q9, D3Q19 or D3Q27) but require both forces and the velocity for each lattice point to calculate.

Parameters

out	source	Central moment-based Guo forcing terms $\vec{\tilde{S}}$
in	v	Force-corrected fluid velocity at lattice point
in	force	Forces acting at lattice point

fGetCentralMomentForceHe()

```
int fGetCentralMomentForceHe (double * source, double * force)
```

Calculates central moment-based He forcing terms for use in cascaded LBE (CLBE) collisions, as obtained by applying the transformation and shift matrices to He source terms:

$$S_i = \frac{f_i^{eq}}{\rho c_s^2} \left(\hat{e}_i - \vec{v} \right) \cdot \vec{F}$$

i.e. :math:` vec{tilde{S}} = mathbf{N} mathbf{T} vec{S}`, which use the generalised Maxwell-Boltzmann local equilibrium distribution. The exact form of central moment forcing terms will depend on the lattice scheme in use (D2Q9 [35], D3Q19 or D3Q27 [33]) but only require forces for each lattice point to calculate.

Parameters

out	source	Central moment-based He forcing terms $\vec{\tilde{S}}$
in	force	Forces acting at lattice point

fGetCentralMomentTransformMatrix()

```
int fGetCentralMomentTransformMatrix (double * rcsh,
                                      double * rcshinv,
                                      double * u)
```

Calculates the products of the transform and shift matrices $\mathbf{NT}$ and the product of their inverses $\mathbf{T}^{-1}\mathbf{N}^{-1}$ for the required lattice scheme to transform distribution functions into central moments and vice versa for cascaded LBE (CLBE) collisions. These matrices are dependent on the fluid velocity and must be calculated for each lattice point and at each timestep.

Parameters

out	rcsh	Product of transform and shift matrices $\mathbf{NT}$
out	rcshinv	Product of inverses of shift and transform matrices $\mathbf{T}^{-1}\mathbf{N}^{-1}$
in	u	Fluid velocity at lattice point

fGetCLBECollide()

```
int fGetCLBECollide (double * collide,
                     double omegashear,
                     double omegabulk,
                     double omegathree,
                     double omegafour)
```

Calculates the main diagonal for the collision matrix $\mathbf{\Lambda}$ used in cascaded LBE (CLBE) collisions. This subroutine requires inputs for four relaxation frequencies at each lattice point and fluid: the main relaxation frequency $\omega = \tau_f^{-1}$, the bulk relaxation frequency $\omega_b = \tau_{f,bulk}^{-1}$, the third and fourth order relaxation frequencies ω_3 and ω_4. For some second-order central moment terms, only the symmetric relaxation frequencies are included in the results for this routine: the anti-symmetric terms for these moments are calculated and applied in the main site collision routines. Moments for fluid density and momentum have their relaxation frequencies set to 1 to ensure these properties are conserved.

Parameters

out	collide	Diagonal of relaxation frequencies for cascaded LBE collision matrix
in	omegashear	Relaxation frequency for fluid (giving kinetic viscosity)
in	omegabulk	Bulk relaxation frequency for fluid (giving bulk viscosity)
in	omegathree	Third-order relaxation frequency for fluid
in	omegafour	Fourth-order relaxation frequency for fluid

fSiteFluidCollisionCLBE()

```
int fSiteFluidCollisionCLBE (double * startpos,
                             double * sitespeed,
                             double * omega,
                             double * omegabulk,
                             double * omega3,
                             double * omega4,
                             double * rho,
                             double * bodyforce)
```

Applies cascaded LBE (CLBE) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the generalised Maxwell-Boltzmann equilibrium distribution functions for mildly compressible fluids and applying standard (Martys-Chen) [91] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	omega3	Third-order relaxation frequencies for fluids at lattice site
in	omega4	Four-order relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidCollisionCLBEEDM()

```
int fSiteFluidCollisionCLBEEDM (double * startpos,
                                double * sitespeed,
                                double * omega,
                                double * omegabulk,
                                double * omega3,
                                double * omega4,
                                double * rho,
                                double * bodyforce)
```

Applies cascaded LBE (CLBE) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the generalised Maxwell-Boltzmann equilibrium distribution functions for mildly compressible fluids and applying Equal Difference Method (EDM) [69] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	omega3	Third-order relaxation frequencies for fluids at lattice site
in	omega4	Four-order relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidCollisionCLBEEDMLishchuk()

```
int fSiteFluidCollisionCLBEEDMLishchuk (double * startpos,
                                        double * sitespeed,
                                        double * omega,
                                        double * omegabulk,
                                        double * omega3,
                                        double * omega4,
                                        double * rho,
                                        double * bodyforce,
                                        double * phaseindex)
```

Applies cascaded LBE (CLBE) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and generalised Maxwell-Boltzmann equilibrium distribution functions for mildly compressible fluids, applying Equal Difference Method (EDM) [69] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	omega3	Third-order relaxation frequencies for fluids at lattice site
in	omega4	Four-order relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidCollisionCLBEEDMLishchukLocal()

```
int fSiteFluidCollisionCLBEEDMLishchukLocal (double * startpos,
                                             double * sitespeed,
                                             double * omega,
                                             double * omegabulk,
                                             double * omega3,
                                             double * omega4,
                                             double * rho,
                                             double * bodyforce,
                                             double * phaseindex,
                                             int thread)
```

Applies cascaded LBE (CLBE) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying Equal Difference Method (EDM) [69] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i \hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegab-ulk	Bulk relaxation frequencies for fluids at lattice site
in	omega3	Third-order relaxation frequencies for fluids at lattice site
in	omega4	Four-order relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidCollisionCLBEGuo()

```
int fSiteFluidCollisionCLBEGuo (double * startpos,
                                double * sitespeed,
                                double * omega,
                                double * omegabulk,
                                double * omega3,
                                double * omega4,
                                double * rho,
                                double * bodyforce)
```

Applies cascaded LBE (CLBE) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the generalised Maxwell-Boltzmann equilibrium distribution functions for mildly compressible fluids and applying Guo [49] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	omega3	Third-order relaxation frequencies for fluids at lattice site
in	omega4	Four-order relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidCollisionCLBEGuoLishchuk()

```
int fSiteFluidCollisionCLBEGuoLishchuk (double * startpos,
                                        double * sitespeed,
                                        double * omega,
                                        double * omegabulk,
                                        double * omega3,
                                        double * omega4,
                                        double * rho,
                                        double * bodyforce,
                                        double * phaseindex)
```

Applies cascaded LBE (CLBE) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and generalised Maxwell-Boltzmann equilibrium distribution functions for mildly compressible fluids, applying Guo [49] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	omega3	Third-order relaxation frequencies for fluids at lattice site
in	omega4	Four-order relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidCollisionCLBEGuoLishchukLocal()

```
int fSiteFluidCollisionCLBEGuoLishchukLocal (double * startpos,
                                             double * sitespeed,
                                             double * omega,
                                             double * omegabulk,
                                             double * omega3,
                                             double * omega4,
                                             double * rho,
                                             double * bodyforce,
                                             double * phaseindex,
                                             int threed)
```

Applies cascaded LBE (CLBE) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying Guo [49] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i \hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegab-ulk	Bulk relaxation frequencies for fluids at lattice site
in	omega3	Third-order relaxation frequencies for fluids at lattice site
in	omega4	Four-order relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidCollisionCLBEHe()

```
int fSiteFluidCollisionCLBEHe (double * startpos,
                               double * sitespeed,
                               double * omega,
                               double * omegabulk,
                               double * omega3,
                               double * omega4,
                               double * rho,
                               double * bodyforce)
```

Applies cascaded LBE (CLBE) collisions to all fluids at a given lattice site, operating on the distribution functions provided, using the generalised Maxwell-Boltzmann equilibrium distribution functions for mildly compressible fluids and applying He [54] forcing.

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	omega3	Third-order relaxation frequencies for fluids at lattice site
in	omega4	Four-order relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site

fSiteFluidCollisionCLBEHeLishchuk()

```
int fSiteFluidCollisionCLBEHeLishchuk (double * startpos,
                                       double * sitespeed,
                                       double * omega,
                                       double * omegabulk,
                                       double * omega3,
                                       double * omega4,
                                       double * rho,
                                       double * bodyforce,
                                       double * phaseindex)
```

Applies cascaded LBE (CLBE) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and generalised Maxwell-Boltzmann equilibrium distribution functions for mildly compressible fluids, applying He [54] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	omega3	Third-order relaxation frequencies for fluids at lattice site
in	omega4	Four-order relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidCollisionCLBEHeLishchukLocal()

```
int fSiteFluidCollisionCLBEHeLishchukLocal (double * startpos,
                                             double * sitespeed,
                                             double * omega,
                                             double * omegabulk,
                                             double * omega3,
                                             double * omega4,
                                             double * rho,
                                             double * bodyforce,
                                             double * phaseindex,
                                             int threed)
```

Applies cascaded LBE (CLBE) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying He [54] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i \hat{e}_i - c_s^2 \mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegab-ulk	Bulk relaxation frequencies for fluids at lattice site
in	omega3	Third-order relaxation frequencies for fluids at lattice site
in	omega4	Four-order relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

fSiteFluidCollisionCLBELishchuk()

```
int fSiteFluidCollisionCLBELishchuk (double * startpos,
                                     double * sitespeed,
                                     double * omega,
                                     double * omegabulk,
                                     double * omega3,
                                     double * omega4,
                                     double * rho,
                                     double * bodyforce,
                                     double * phaseindex)
```

Applies cascaded LBE (CLBE) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and generalised Maxwell-Boltzmann equilibrium distribution functions for mildly compressible fluids, applying standard (Martys-Chen) [91] forcing for all forces (including Lishchuk interfacial forces) and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegabulk	Bulk relaxation frequencies for fluids at lattice site
in	omega3	Third-order relaxation frequencies for fluids at lattice site
in	omega4	Four-order relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phaseindex	Phase indices for all fluid pairs at lattice site

fSiteFluidCollisionCLBELishchukLocal()

```
int fSiteFluidCollisionCLBELishchukLocal (double * startpos,
                                          double * sitespeed,
                                          double * omega,
                                          double * omegabulk,
                                          double * omega3,
                                          double * omega4,
                                          double * rho,
                                          double * bodyforce,
                                          double * phaseindex,
                                          int thread)
```

Applies cascaded LBE (CLBE) collisions to all fluids at a given lattice site, operating on achromatic distribution functions (summed over all fluids for each lattice link) and using the local equilibrium distribution functions for mildly compressible fluids, applying standard (Martys-Chen) [91] forcing for all forces except Lishchuk interfacial forces, which are applied using a direct forcing term [129]:

$$F_i^{ab} = \frac{w_i \beta^{ab} g^{ab} \rho^a \rho^b}{c_s^4 \rho^3 \tau_f \Delta t} \left(\hat{n}_{ab}\hat{n}_{ab} - \mathbf{I}\right) : \left(\hat{e}_i\hat{e}_i - c_s^2\mathbf{I}\right)$$

and re-separating the fluids using D'Ortona segregation [25]:

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

Parameters

in,out	startpos	Pointer to distribution functions at current lattice site for applying collision
in	sitespeed	Fluid velocity at lattice site
in	omega	Relaxation frequencies for fluids at lattice site
in	omegab-ulk	Bulk relaxation frequencies for fluids at lattice site
in	omega3	Third-order relaxation frequencies for fluids at lattice site
in	omega4	Four-order relaxation frequencies for fluids at lattice site
in	rho	Macroscopic fluid densities at lattice site
in	bodyforce	Forces to apply to each fluid at lattice site
in	phasein-dex	Phase indices for all fluid pairs at lattice site
in	threed	Flag to indicate whether or not the lattice is three-dimensional (affects the direct forcing term)

5.17 lbpFORCE.cpp

Module with routines to calculate non-constant interaction and heat buoyancy forces. (Header file available as lbpFORCE.hpp.)

Calculates forces and other contributions to determine fluid interactions and Boussinesq buoyancy for heat convection, i.e. any emergent force that does not remain constant during the simulation.

In the cases of Shan-Chen pseudopotential and Swift free-energy interactions, these can make fluids behave according to specific equations of state:

- Ideal lattice gas:

$$p = \rho c_s^2$$

- Shan-Chen 1993 model [118]:

$$p = \rho c_s^2 + \frac{1}{2} c_s^2 g \rho_0^2 \left(1 - e^{-\frac{\rho}{\rho_0}}\right)$$

- Shan-Chen 1994 model [119]:

$$p = \rho c_s^2 + \frac{1}{2} c_s^2 g \psi_0^2 e^{-\frac{2\rho_0}{\rho}}$$

- Qian model [106]:

$$p = \rho c_s^2 + \frac{c_s^2 g \rho_0^2 \rho^2}{(\rho_0 + \rho)^2}$$

- Density model:

$$p = \rho c_s^2 + \frac{1}{2} c_s^2 g \rho^2$$

- Ideal gas:

$$p = \rho RT$$

- van der Waals:

$$p = \frac{\rho RT}{1 - b\rho} - a\rho^2$$

- Carnahan-Starling-van der Waals [16]:

$$p = \rho RT\left(\frac{1+\phi+\phi^2-\phi^3}{(1-\phi)^3}\right) - a\rho^2$$

- Redlich-Kwong [110]:

$$p = \frac{\rho RT}{1-b\rho} - \frac{a\rho^2}{\sqrt{T}\,(1+b\rho)}$$

- Soave-Redlich-Kwong [128]:

$$p = \frac{\rho RT}{1-b\rho} - \frac{a\alpha\left(T_r,\omega\right)\rho^2}{1+b\rho}$$

- Peng-Robinson *[37]* :

$$p = \frac{\rho RT}{1-b\rho} - \frac{a\alpha\left(T_r,\omega\right)\rho^2}{1+2b\rho-b^2\rho^2}$$

- Carnahan-Starling-Redlich-Kwong [16]:

$$p = \rho RT\left(\frac{1+\phi+\phi^2-\phi^3}{(1-\phi)^3}\right) - \frac{a\rho^2}{\sqrt{T}\,(1+b\rho)}$$

where R is the universal gas constant, a and b are species-dependent coefficients, α is a function dependent on the ratio of temperature to critical temperature $T_r = T/T_c$ and acentric factor ω, and $\phi = \frac{b\rho}{4}$ for Carnahan-Starling equations of state. The temperatures used in some equatios of state can either be specified system-wide or at each lattice point if heat effects are coupled to fluid flows with an additional lattice.

5.17.1 Functions

- `int` *fInteractionForceZero()*

  Resets all interaction and heat forces to zero before calculations.

- `int` *fCalcPotential_ShanChen()*

  Calculates Shan-Chen pseudopotentials for each fluid at each lattice point.

- `int` *fCalcInteraction_ShanChen()*

  Calculates interaction forces for Shan-Chen pseudopotential model (including solid-fluid wetting forces) at fluid points away from subdomain boundaries.

- `int` *fCalcInteraction_ShanChen_Boundary()*

  Calculates interaction forces for Shan-Chen pseudopotential model (including solid-fluid wetting forces) at fluid points close to subdomain boundaries.

- `int` *fCalcInteraction_ShanChenQuadratic()*

  Calculates interaction forces for a quadratic Shan-Chen pseudopotential model (including solid-fluid wetting forces) at fluid points away from subdomain boundaries.

- `int` *fCalcInteraction_ShanChenQuadratic_Boundary()*

  Calculates interaction forces for a quadratic Shan-Chen pseudopotential model (including solid-fluid wetting forces) at fluid points close to subdomain boundaries.

- `int` *fInteractionForceShanChen()*

  Calculates interaction forces for all fluids based on the Shan-Chen pseudopodential model for parallel running.

- `int` *fsInteractionForceShanChen()*

  Calculates interaction forces for all fluids based on the Shan-Chen pseudopodential model for serial running.

- `int` *fInteractionForceShanChenQuadratic()*

  Calculates interaction forces for all fluids based on the quadratic Shan-Chen pseudopodential model for parallel running.

- `int` *fsInteractionForceShanChenQuadratic()*

  Calculates interaction forces for all fluids based on the quadratic Shan-Chen pseudopodential model for serial running.

- `int` *fCalcPhaseIndex_Lishchuk()*

  Calculate interfacial normal vectors for Lishchuk interaction model using non-local derivative calculations at all fluid points away from subdomain boundaries.

- `int` *fsCalcPhaseIndex_Lishchuk()*

  Calculate interfacial normal vectors for Lishchuk interaction model using non-local derivative calculations at all fluid points including those in subdomain boundaries.

- `int` *fCalcPhaseIndex_LishchukLocal()*

  Calculate interfacial normal vectors for Lishchuk interaction model using local derivative calculations at all fluid points.

- `int` *fCalcInteraction_Lishchuk()*

  Calculates interaction forces between fluids for original Lishchuk interaction model for fluid points away from subdomain boundaries.

- `int` *fCalcInteraction_Lishchuk_Boundary()*

  Calculates interaction forces between fluids for original Lishchuk interaction model for fluid points close to subdomain boundaries.

- `int` *fCalcInteraction_LishchukSpencer()*

  Calculates interaction forces between fluids for Lishchuk-Spencer interaction model for fluid points away from subdomain boundaries.

- `int` *fCalcInteraction_LishchukSpencer_Boundary()*

  Calculates interaction forces between fluids for Lishchuk-Spencer interaction model for fluid points close to subdomain boundaries.

- `int` *fWallInteractionForceLishchukLocal()*

  Calculates interaction forces between walls and fluids for Lishchuk interactions without force calculations.

- `int` *fInteractionForceLishchuk()*

  Calculates interaction forces for all fluids based on the original Lishchuk continuum-based interaction model for parallel running.

- `int` *fsInteractionForceLishchuk()*

  Calculates interaction forces for all fluids based on the original Lishchuk continuum-based interaction model for serial running.

- `int` *fInteractionForceLishchukSpencer()*

  Calculates interaction forces for all fluids based on the Lishchuk-Spencer continuum-based interaction model for parallel running.

- `int` *fsInteractionForceLishchukSpencer()*

  Calculates interaction forces for all fluids based on the Lishchuk-Spencer continuum-based interaction model for serial running.

- `int` *fCalcDensityGradient_Swift()*

  Calculates first- and second-order derivatives of fluid density for the one-fluid Swift free-energy model at fluid points away from subdomain boundaries.

- `int` *fsCalcDensityGradient_Swift()*

  Calculates first- and second-order derivatives of fluid density for the one-fluid Swift free-energy model at all fluid points.

- `int` *fCalcDensityConcentrationGradient_Swift()*

  Calculates first- and second-order derivatives of fluid density and concentration for the two-fluid Swift free-energy model at fluid points away from subdomain boundaries.

- `int` *fsCalcDensityConcentrationGradient_Swift()*

  Calculates first- and second-order derivatives of fluid density and concentration for the two-fluid Swift free-energy model at all fluid points.

- `int` *fCalcGradient_Swift()*

  Calculate density (and concentration) gradients for Swift free-energy interactions when running in parallel.

- `int` *fsCalcGradient_Swift()*

  Calculate density (and concentration) gradients for Swift free-energy interactions when running in serial.

- `int` *fCalcForce_Boussinesq()*

  Calculates buoyancy-driven thermal convection force according to the Boussinesq approximation.

- `int` *fConvectionForceBoussinesq()*

  Calculates buoyancy-driven thermal convection forces at all fluid lattice points according to the Boussinesq approximation.

5.17.2 Function Documentation

fCalcDensityConcentrationGradient_Swift()

```
int fCalcDensityConcentrationGradient_Swift ()
```

Calculates the first-order and second-order derivatives of fluid density and concentration for the two-fluid Swift free-energy interaction model at all fluid lattice points away from the edges of the processor's subdomain using either a stencil to reduce spurious microcurrents [102] or central difference and one-sided difference approximations for wet boundary nodes and grid points next to bounce-back boundary points. A quadratic surface wetting potential in density and concentration can also be applied, which modifies density and concentration derivatives at or near boundary points [13][103]. Since boundary halo points are omitted by this subroutine (intended for use in parallel running), density and concentration derivatives for these lattice points have to be communicated from neighbouring processors.

fCalcDensityGradient_Swift()

```
int fCalcDensityGradient_Swift ()
```

Calculates the first-order and second-order derivatives of fluid density for the one-fluid Swift free-energy interaction model at all fluid lattice points away from the edges of the processor's subdomain using either a stencil to reduce spurious microcurrents [102] or central difference and one-sided difference approximations for wet boundary nodes and grid points next to bounce-back boundary points. A quadratic surface wetting potential in density can also be applied, which modifies density derivatives at or near boundary points [13][103]. Since boundary halo points are omitted by this subroutine (intended for use in parallel running), density derivatives for these lattice points have to be communicated from neighbouring processors.

fCalcForce_Boussinesq()

```
int fCalcForce_Boussinesq (long tpos,
                           double temph,
                           double templ)
```

Calculates temperature-dependent forces at a given lattice point based on the Boussinesq approximation [48]:

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

where $T_0 = \frac{1}{2}(T_h + T_l)$ is the reference temperature, T_h and T_l are respectively the high and low system temperatures and β^a is the volumetric expansion coefficient for fluid a. (The product of gravitational acceleration and volumetric expansion $\vec{g}\beta^a$ is supplied by the user.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	temph	High temperature value for system T_h
in	templ	Low temperature value for system T_l

fCalcGradient_Swift()

```
int fCalcGradient_Swift ()
```

Calculates first-order and second-order density (and concentration) derivatives at all lattice points apart from boundary halo grid points for Swift free-energy interactions [136][135]. This subroutine avoids any lattice points that make up the boundary halo when running in parallel: the gradients for these points are to be communicated from neighbouring processors prior to collisions.

fCalcInteraction_Lishchuk()

```
int fCalcInteraction_Lishchuk (int xpos,
                               int ypos,
                               int zpos)
```

Calculates the interaction forces between pairs of immiscible fluids for the Lishchuk interaction scheme [81]:

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

where $K_{ab} = -\nabla_S \cdot \hat{n}_{ab}$ is the local curvative between fluids a and b and the first-order differential of phase index can be obtained using the interfacial normal, densities of the two fluids and of all fluids and the segregation parameter between the two fluids :math:beta^{ab}` [50]:

$$\nabla \rho^N_{ab} = \frac{4\beta^{ab}\rho^a \rho^b}{\rho^3} \hat{n}_{ab}.$$

Forces are also calculated and applied for surface wetting effects by assuming fluid 0 is the background fluid and wets the walls, which applies an uncompensated Young stress on each fluid [28]:

$$\vec{F}^{0a}_{wet} = -\frac{1}{2} g_{wall,a} \nabla_{S,wall} \rho^N_{0a} = \frac{2 g_{wall,a} \beta^{0a} \rho^0 \rho^a}{\rho^3} \left(\hat{n}_w \left(\hat{n}_{0a} \cdot \hat{n}_w \right) - \hat{n}_{0a} \right)$$

where $\hat{n}_w$ is the normal vector to the solid surface.

This subroutine omits calculations of interfacial curvatures and forces at grid points close to the subdomain boundaries, as the former requires non-local gradient calculations that would use modulo functions to find neighbouring grid points: as such, this subroutine can be used for parallel calculations, although communication of interfacial forces for boundary halo points from neighbourint processors is required prior to collisions.

Parameters

in	xpos	Position of lattice point (x-component)
in	ypos	Position of lattice point (y-component)
in	zpos	Position of lattice point (z-component)

fCalcInteraction_Lishchuk_Boundary()

```
int fCalcInteraction_Lishchuk_Boundary (int xpos,
                                        int ypos,
                                        int zpos)
```

Calculates the interaction forces between pairs of immiscible fluids for the Lishchuk interaction scheme [81]:

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

where $K_{ab} = -\nabla_S \cdot \hat{n}_{ab}$ is the local curvative between fluids a and b and the first-order differential of phase index can be obtained using the interfacial normal, densities of the two fluids and of all fluids and the segregation parameter between the two fluids :math:beta^{ab}` [50]:

$$\nabla \rho^N_{ab} = \frac{4\beta^{ab}\rho^a\rho^b}{\rho^3}\hat{n}_{ab}.$$

Forces are also calculated and applied for surface wetting effects by assuming fluid 0 is the background fluid and wets the walls, which applies an uncompensated Young stress on each fluid [28]:

$$\vec{F}^{0a}_{wet} = -\frac{1}{2} g_{wall,a} \nabla_{S,wall} \rho^N_{0a} = \frac{2 g_{wall,a} \beta^{0a} \rho^0 \rho^a}{\rho^3} \left(\hat{n}_w \left(\hat{n}_{0a} \cdot \hat{n}_w \right) - \hat{n}_{0a} \right)$$

where $\hat{n}_w$ is the normal vector to the solid surface.

This subroutine calculates interfacial curvatures and forces at grid points close to the subdomain boundaries: the former requires non-local gradient calculations with modulo functions to find neighbouring grid points: as such, this subroutine can be used for serial calculations.

Parameters

in	xpos	Position of lattice point (x-component)
in	ypos	Position of lattice point (y-component)
in	zpos	Position of lattice point (z-component)

fCalcInteraction_LishchukSpencer()

```
int fCalcInteraction_LishchukSpencer (int xpos,
                                      int ypos,
                                      int zpos,
                                      double * rho)
```

Calculates the interaction forces between pairs of immiscible fluids for the Lishchuk-Spencer interaction scheme [129]:

$$\vec{F}^{ab} = -2 g_{ab} \beta^{ab} \nabla \cdot \left(\frac{\rho^a \rho^b}{\rho^3} \hat{n}_{ab} \hat{n}_{ab} \right)$$

where β^{ab} is the segregation parameter between the two fluids. Compared to the standard (original) Lishchuk interaction force, no interfacial curvature is required and correct behaviour can thus be achieved for lattice points with more than two fluids, although a reduction of density of up to $\frac{g_{ab}\beta^{ab}}{2c_s^2}$ can be observed in the interfacial regions between fluids.

Forces are also calculated and applied for surface wetting effects by assuming fluid 0 is the background fluid and wets the walls, which applies an uncompensated Young stress on each fluid [28]:

$$\vec{F}_{wet}^{0a} = -\frac{1}{2} g_{wall,a} \nabla_{S,wall} \rho_{0a}^N = \frac{2 g_{wall,a} \beta^{0a} \rho^0 \rho^a}{\rho^3} \left(\hat{n}_w \left(\hat{n}_{0a} \cdot \hat{n}_w \right) - \hat{n}_{0a} \right)$$

where $\hat{n}_w$ is the normal vector to the solid surface.

This subroutine omits calculations of interfacial curvatures and forces at grid points close to the subdomain boundaries, as the former requires non-local gradient calculations that would use modulo functions to find neighbouring grid points: as such, this subroutine can be used for parallel calculations, although communication of interfacial forces for boundary halo points from neighbourint processors is required prior to collisions.

Parameters

in	xpos	Position of lattice point (x-component)
in	ypos	Position of lattice point (y-component)
in	zpos	Position of lattice point (z-component)
in	rho	Densities of fluids at lattice point

fCalcInteraction_LishchukSpencer_Boundary()

```
int fCalcInteraction_LishchukSpencer_Boundary (int xpos,
                                               int ypos,
                                               int zpos,
                                               double * rho)
```

Calculates the interaction forces between pairs of immiscible fluids for the Lishchuk-Spencer interaction scheme [129]:

$$\vec{F}^{ab} = -2 g_{ab} \beta^{ab} \nabla \cdot \left(\frac{\rho^a \rho^b}{\rho^3} \hat{n}_{ab} \hat{n}_{ab} \right)$$

where β^{ab} is the segregation parameter between the two fluids. Compared to the standard (original) Lishchuk interaction force, no interfacial curvature is required and correct behaviour can thus be achieved for lattice points with more than two fluids, although a reduction of density of up to $\frac{g_{ab}\beta^{ab}}{2c_s^2}$ can be observed in the interfacial regions between fluids.

Forces are also calculated and applied for surface wetting effects by assuming fluid 0 is the background fluid and wets the walls, which applies an uncompensated Young stress on each fluid [28]:

$$\vec{F}_{wet}^{0a} = -\frac{1}{2} g_{wall,a} \nabla_{S,wall} \rho_{0a}^N = \frac{2 g_{wall,a} \beta^{0a} \rho^0 \rho^a}{\rho^3} \left(\hat{n}_w \left(\hat{n}_{0a} \cdot \hat{n}_w \right) - \hat{n}_{0a} \right)$$

where $\hat{n}_w$ is the normal vector to the solid surface.

This subroutine calculates interfacial curvatures and forces at grid points close to the subdomain boundaries: the former requires non-local gradient calculations with modulo functions to find neighbouring grid points: as such, this subroutine can be used for serial calculations.

Parameters

in	xpos	Position of lattice point (x-component)
in	ypos	Position of lattice point (y-component)
in	zpos	Position of lattice point (z-component)
in	rho	Densities of fluids at lattice point

fCalcInteraction_ShanChen()

```
int fCalcInteraction_ShanChen (int xpos,
                               int ypos,
                               int zpos)
```

Determines interaction forces for the Shan-Chen pseudopotential model [118][119] based on gradients of pseudopotentials:

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

at all lattice points away from the edges of the subdomain held by each processor to avoid both links crossing periodic boundaries and the use of modulo functions to find neighbouring grid points.

Surface wetting forces can also be calculated at grid points next to boundaries by using a switching function $s\left(\vec{x}\right)$ to represent surfaces (1) or fluid (0) with one of the following wetting forces:

- Density-based interactions [91]:

$$\vec{F}^a_{wet}\left(\vec{x}\right) = -g_{a,wall}\rho^a\left(\vec{x}\right)\sum_i w_i s\left(\vec{x}+\hat{e}_i\right)\hat{e}_i$$

- Potential-based interactions [108][109]:

$$\vec{F}^a_{wet} = -g_{a,wall}\phi^a\left(\vec{x}\right)\sum_i w_i s\left(\vec{x}+\hat{e}_i\right)\hat{e}_i$$

- Screened potential interactions [80]:

$$\vec{F}^a_{wet} = -g_{a,wall}\phi^a\left(\vec{x}\right)\sum_i w_i \phi^a\left(\vec{x}\right) s\left(\vec{x}+\hat{e}_i\right)\hat{e}_i$$

Parameters

in	xpos	Position of lattice point (x-component)
in	ypos	Position of lattice point (y-component)
in	zpos	Position of lattice point (z-component)

fCalcInteraction_ShanChen_Boundary()

```
int fCalcInteraction_ShanChen_Boundary (int xpos,
                                        int ypos,
                                        int zpos)
```

Determines interaction forces for the Shan-Chen pseudopotential model [118][119] based on gradients of pseudopotentials:

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

at all lattice points close to the edges of the subdomain held by each processor, which require the use of modulo functions to find neighbouring grid points.

Surface wetting forces can also be calculated at grid points next to boundaries by using a switching function $s\left(\vec{x}\right)$ to represent surfaces (1) or fluid (0) with one of the following wetting forces:

- Density-based interactions [91]:

$$\vec{F}^a_{wet}\left(\vec{x}\right) = -g_{a,wall}\rho^a\left(\vec{x}\right)\sum_i w_i s\left(\vec{x}+\hat{e}_i\right)\hat{e}_i$$

- Potential-based interactions [108][109]:

$$\vec{F}^a_{wet} = -g_{a,wall}\phi^a\left(\vec{x}\right)\sum_i w_i s\left(\vec{x}+\hat{e}_i\right)\hat{e}_i$$

- Screened potential interactions [80]:

$$\vec{F}^a_{wet} = -g_{a,wall}\phi^a\left(\vec{x}\right)\sum_i w_i \phi^a\left(\vec{x}\right) s\left(\vec{x}+\hat{e}_i\right)\hat{e}_i$$

Parameters

in	xpos	Position of lattice point (x-component)
in	ypos	Position of lattice point (y-component)
in	zpos	Position of lattice point (z-component)

fCalcInteraction_ShanChenQuadratic()

```
int fCalcInteraction_ShanChenQuadratic (int xpos,
                                        int ypos,
                                        int zpos)
```

Determines interaction forces for a form of the Shan-Chen pseudopotential model with quadratic pseudopotential terms [70][43][61]:

$$\vec{F}^a\left(\vec{x}\right) = -\psi^a\left(\vec{x}\right)\sum_b \beta_{ab} g_{ab}\sum_i w_i\psi^b\left(\vec{x}+\hat{e}_i\right)\hat{e}_i - \frac{1}{2}\sum_b g_{ab}\left(1-\beta_{ab}\right)\sum_i w_i\left(\psi^b\left(\vec{x}+\hat{e}_i\right)\right)^2\hat{e}_i$$

where β_{ab} is a weighting factor between linear and quadratic pseudopotential terms that can be adjusted for different equations of state. These forces are calculated at all lattice points away from the edges of the subdomain held by each processor to avoid both links crossing periodic boundaries and the use of modulo functions to find neighbouring grid points.

Surface wetting forces can also be calculated at grid points next to boundaries by using a switching function $s\left(\vec{x}\right)$ to represent surfaces (1) or fluid (0) with one of the following wetting forces:

- Density-based interactions [91]:

$$\vec{F}^a_{wet}\left(\vec{x}\right) = -g_{a,wall}\rho^a\left(\vec{x}\right)\sum_i w_i s\left(\vec{x}+\hat{e}_i\right)\hat{e}_i$$

- Potential-based interactions [108][109]:

$$\vec{F}^a_{wet} = -g_{a,wall}\phi^a\left(\vec{x}\right)\sum_i w_i s\left(\vec{x}+\hat{e}_i\right)\hat{e}_i$$

- Screened potential interactions [80]:

$$\vec{F}^a_{wet} = -g_{a,wall}\phi^a\left(\vec{x}\right)\sum_i w_i \phi^a\left(\vec{x}\right) s\left(\vec{x}+\hat{e}_i\right)\hat{e}_i$$

Parameters

in	xpos	Position of lattice point (x-component)
in	ypos	Position of lattice point (y-component)
in	zpos	Position of lattice point (z-component)

fCalcInteraction_ShanChenQuadratic_Boundary()

```
int fCalcInteraction_ShanChenQuadratic_Boundary (int xpos,
                                                  int ypos,
                                                  int zpos)
```

Determines interaction forces for a form of the Shan-Chen pseudopotential model with quadratic pseudopotential terms [70][43][61]:

$$\vec{F}^a\left(\vec{x}\right) = -\psi^a\left(\vec{x}\right)\sum_b \beta_{ab} g_{ab} \sum_i w_i \psi^b\left(\vec{x}+\hat{e}_i\right)\hat{e}_i - \frac{1}{2}\sum_b g_{ab}\left(1-\beta_{ab}\right)\sum_i w_i \left(\psi^b\left(\vec{x}+\hat{e}_i\right)\right)^2 \hat{e}_i$$

where β_{ab} is a weighting factor between linear and quadratic pseudopotential terms that can be adjusted for different equations of state. These forces are calculated at all lattice points close to the edges of the subdomain held by each processor, which require the use of modulo functions to find neighbouring grid points.

Surface wetting forces can also be calculated at grid points next to boundaries by using a switching function $s\left(\vec{x}\right)$ to represent surfaces (1) or fluid (0) with one of the following wetting forces:

- Density-based interactions [91]:

$$\vec{F}^a_{wet}\left(\vec{x}\right) = -g_{a,wall}\rho^a\left(\vec{x}\right)\sum_i w_i s\left(\vec{x}+\hat{e}_i\right)\hat{e}_i$$

- Potential-based interactions [108][109]:

$$\vec{F}^a_{wet} = -g_{a,wall}\phi^a\left(\vec{x}\right)\sum_i w_i s\left(\vec{x}+\hat{e}_i\right)\hat{e}_i$$

- Screened potential interactions [80]:

$$\vec{F}^a_{wet} = -g_{a,wall}\phi^a\left(\vec{x}\right)\sum_i w_i \phi^a\left(\vec{x}\right) s\left(\vec{x}+\hat{e}_i\right)\hat{e}_i$$

Parameters

in	xpos	Position of lattice point (x-component)
in	ypos	Position of lattice point (y-component)
in	zpos	Position of lattice point (z-component)

fCalcPhaseIndex_Lishchuk()

```
int fCalcPhaseIndex_Lishchuk ()
```

Calculates the interfacial normal vectors between pairs of fluid species required for Lishchuk interactions [81] by determing phase indices at every available grid point:

$$\rho^N_{ab} = \frac{\rho^a - \rho^b}{\rho^a + \rho^b},$$

using stencils to approximate first-order gradients at each grid point [75], ordinarily one that uses values at nearest neighbouring grid points:

$$\nabla\rho^N_{ab}\left(\vec{x}\right) \approx \frac{1}{c_s^2\Delta t}\sum_i w_i \rho^N_{ab}\left(\vec{x}+\hat{e}_i\right)\hat{e}_i,$$

and normalising these gradients to obtain the interfacial normals:

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

This subroutine omits calculations of phase index gradients and interfacial normals at grid points close to the subdomain boundaries to avoid using modulo functions to find neighbouring grid points: as such, this subroutine can be used for parallel calculations, although communication of phase indices for boundary halo points from neighbouring processors is required.

fCalcPhaseIndex_LishchukLocal()

```
int fCalcPhaseIndex_LishchukLocal ()
```

Calculates the interfacial normal vectors between pairs of fluid species required for Lishchuk interactions [81] by approximating phase indices for each lattice link by using distribution functions [130]:

$$\rho_{ab,i}^N \approx -\frac{f_i^a - f_i^b}{f_i^a + f_i^b},$$

using stencils to approximate first-order gradients at each grid point:

$$\nabla \rho_{ab}^N (\vec{x}) \approx \frac{1}{c_s^2 \Delta t} \sum_i w_i \rho_{ab,i}^N (\vec{x}) \hat{e}_i,$$

and normalising these gradients to obtain the interfacial normals:

$$\hat{n}_{ab} = \frac{\nabla \rho_{ab}^N}{|\nabla \rho_{ab}^N|}.$$

While the approximated phase index differentials may not be very accurate, the resulting interfacial normals are generally sufficiently accurate for Lishchuk interaction calculations. There is a risk of the approximated phase indices being out of range, so the gradients are only calculated when $|\rho_{ab}^N| \leq 1 - \epsilon$, where ϵ is a small value. As this subroutine only requires distribution functions at each lattice point, these calculations can be carried out safely in boundary halos and thus no communication of interfacial normals from neighbouring processors into halos is required.

fCalcPotential_ShanChen()

```
int fCalcPotential_ShanChen ()
```

Calculates the required pseudopotentials ϕ^a for each fluid for all available lattice points to enable Shan-Chen interaction forces [118][119] to be calculated. Specific equations of state can be obtained by specifying the form of pseudopotential ϕ^a to provide it [155].

fConvectionForceBoussinesq()

```
int fConvectionForceBoussinesq (double temph, double templ)
```

Calculates temperature-dependent forces at all fluid lattice points (excluding wet boundary nodes covered by boundary conditions) based on the Boussinesq approximation [48], specifying T_h and T_l as the high and low system temperatures respectively. No communication of thermal forces is required and this subroutine can thus be used for both serial and parallel running.

Parameters

in	temph	High temperature value for system T_h
in	templ	Low temperature value for system T_l

fInteractionForceLishchuk()

```
int fInteractionForceLishchuk ()
```

Calculates the interaction forces acting on all fluids at all grid points away from processor subdomain boundaries using the original Lishchuk continuum-based interaction model. This subroutine avoids any lattice points that make up the boundary halo when running in parallel: the forces for these points are to be communicated from neighbouring processors prior to collisions.

fInteractionForceLishchukSpencer()

```
int fInteractionForceLishchukSpencer ()
```

Calculates the interaction forces acting on all fluids at all grid points away from processor subdomain boundaries using the Lishchuk-Spencer continuum-based interaction model. This subroutine avoids any lattice points that make up the boundary halo when running in parallel: the forces for these points are to be communicated from neighbouring processors prior to collisions.

fInteractionForceShanChen()

```
int fInteractionForceShanChen ()
```

Calculates the interaction forces acting on all fluids at all grid points away from processor subdomain boundaries using the Shan-Chen pseudopotential model. This subroutine avoids any lattice points that make up the boundary halo when running in parallel: the forces for these points are to be communicated from neighbouring processors prior to collisions.

fInteractionForceShanChenQuadratic()

```
int fInteractionForceShanChenQuadratic ()
```

Calculates the interaction forces acting on all fluids at all grid points away from processor subdomain boundaries using the Shan-Chen pseudopotential model with quadratic psuedopotential terms. This subroutine avoids any lattice points that make up the boundary halo when running in parallel: the forces for these points are to be communicated from neighbouring processors prior to collisions.

fInteractionForceZero()

```
int fInteractionForceZero ()
```

Sets arrays for interaction and heat forces to zero before any of these are calculated during the current timestep. This routine is only required if mesophase interactions and/or Boussinesq buoyancy forces are calculated.

fsCalcDensityConcentrationGradient_Swift()

```
int fsCalcDensityConcentrationGradient_Swift ()
```

Calculates the first-order and second-order derivatives of fluid density and concentration for the two-fluid Swift free-energy interaction model at all fluid lattice points - both away from and at the edges of the processor's subdomain - using either a stencil to reduce spurious microcurrents [102] or central difference and one-sided difference approximations for wet boundary nodes and grid points next to bounce-back boundary points. A quadratic surface wetting potential in density and concentration can also be applied, which modifies density and concentration derivatives at or near boundary points [13][103]. Modulo functions are used to find neighbouring grid points across periodic boundaries: this subroutine is therefore suitable for serial calculations.

fsCalcDensityGradient_Swift()

```
int fsCalcDensityGradient_Swift ()
```

Calculates the first-order and second-order derivatives of fluid density for the one-fluid Swift free-energy interaction model at all fluid lattice points - both away from and at the edges of the processor's subdomain - using either a stencil to reduce spurious microcurrents [102] or central difference and one-sided difference approximations for wet boundary nodes and grid points next to bounce-back boundary points. A quadratic surface wetting potential in density can also be applied, which modifies density derivatives at or near boundary points [13][103]. Modulo functions are used to find neighbouring grid points across periodic boundaries: this subroutine is therefore suitable for serial calculations.

fsCalcGradient_Swift()

```
int fsCalcGradient_Swift ()
```

Calculates first-order and second-order density (and concentration) derivatives at all lattice points - including boundary halo grid points - for Swift free-energy interactions [136][135]. The gradient calculations at subdomain boundaries use modulo functions to find neighbouring points at opposite sides of the lattice, which are required for serial running when no boundary halo is in use.

fsCalcPhaseIndex_Lishchuk()

```
int fsCalcPhaseIndex_Lishchuk ()
```

Calculates the interfacial normal vectors between pairs of fluid species required for Lishchuk interactions [81] by determing phase indices at every available grid point:

$$\rho_{ab}^{N} = \frac{\rho^a - \rho^b}{\rho^a + \rho^b},$$

using stencils to approximate first-order gradients at each grid point [75] , ordinarily one that uses values at nearest neighbouring grid points:

$$\nabla \rho_{ab}^{N}\left(\vec{x}\right) \approx \frac{1}{c_s^2 \Delta t} \sum_i w_i \rho_{ab}^{N}\left(\vec{x} + \hat{e}_i\right) \hat{e}_i,$$

and normalising these gradients to obtain the interfacial normals:

$$\hat{n}_{ab} = \frac{\nabla \rho_{ab}^{N}}{\left|\nabla \rho_{ab}^{N}\right|}.$$

This subroutine includes calculations of phase index gradients and interfacial normals at grid points close to the subdomain boundaries, which requires modulo functions to find neighbouring grid points: as such, this subroutine can be used for serial calculations where no boundary halos are in use.

fsInteractionForceLishchuk()

```
int fsInteractionForceLishchuk ()
```

Calculates the interaction forces acting on all fluids at all grid points - both at and away from processor subdomain boundaries using the original Lishchuk continuum-based interaction model. The force calculations at subdomain boundaries use modulo functions to find neighbouring points at opposite sides of the lattice, which are required for serial running when no boundary halo is in use.

fsInteractionForceLishchukSpencer()

```
int fsInteractionForceLishchukSpencer ()
```

Calculates the interaction forces acting on all fluids at all grid points - both at and away from processor subdomain boundaries using the Lishchuk-Spencer continuum-based interaction model. The force calculations at subdomain boundaries use modulo functions to find neighbouring points at opposite sides of the lattice, which are required for serial running when no boundary halo is in use.

fsInteractionForceShanChen()

```
int fsInteractionForceShanChen ()
```

Calculates the interaction forces acting on all fluids at all grid points - both at and away from processor subdomain boundaries using the Shan-Chen pseudopotential model. The force calculations at subdomain boundaries use modulo functions to find neighbouring points at opposite sides of the lattice, which are required for serial running when no boundary halo is in use.

fsInteractionForceShanChenQuadratic()

```
int fsInteractionForceShanChenQuadratic ()
```

Calculates the interaction forces acting on all fluids at all grid points - both at and away from processor subdomain boundaries using the Shan-Chen pseudopotential model with quadratic psuedopotential terms. The force calculations at subdomain boundaries use modulo functions to find neighbouring points at opposite sides of the lattice, which are required for serial running when no boundary halo is in use.

fWallInteractionForceLishchukLocal()

```
int fWallInteractionForceLishchukLocal ()
```

Calculates and applies forces for surface wetting effects by assuming fluid 0 is the background fluid and wets the walls, which applies an uncompensated Young stress on each fluid [28]:

$$\vec{F}_{wet}^{0a} = -\frac{1}{2} g_{wall,a} \nabla_{S,wall} \rho_{0a}^{N} = \frac{2 g_{wall,a} \beta^{0a} \rho^0 \rho^a}{\rho^3} \left(\hat{n}_w \left(\hat{n}_{0a} \cdot \hat{n}_w \right) - \hat{n}_{0a} \right)$$

where $\hat{n}_w$ is the normal vector to the solid surface. This subroutine is required for variants of the Lishchuk algorithm that do not calculate interaction forces but apply direct forcing terms in collisions, i.e. Lishchuk 'Spencer tensor' [129] and the fully local Lishchuk [130] algorithms. Since these forces do not require gradients of the interfacial normals, they can be calculated for all lattice points including boundary halos and thus no communication between neighbouring processors is required.

5.18 lbpRHEOLOGY.cpp

Module with routines to apply various rheological models for LBE calculations. (Header file available as lbpRHEOLOGY.hpp.)

Functions and subroutines to calculate shear rates at each lattice site and calculate relaxation frequencies based on the shear rates to apply specific rheological models to the fluids in a LBE calculation. Rate-of-strain tensors can be determined using momentum flux tensors [12], which can be calculated locally at each lattice site using its distribution functions but depend upon the relaxation frequencies used for the previous timestep:

$$S_{\alpha\beta} = -\frac{3}{2\rho\Delta t} \sum_i e_{i,\alpha} e_{i,\beta} \sum_j \Lambda_{ij} \left(f_j - f_j^{eq} \right) \equiv \frac{1}{2} \left(\frac{\partial u_\beta}{\partial x_\alpha} + \frac{\partial u_\alpha}{\partial x_\beta} \right).$$

The shear rate is obtained from the rate-of-strain tensor:

$$\dot{\gamma} = \sqrt{2 \sum_{\alpha,\beta} S_{\alpha\beta} S_{\alpha\beta}}$$

and the shear rate can then be used in various rheological models to find the gradient of shear stress with respect to shear rate (i.e. viscosity) for each lattice site.

5.18.1 Functions

- `double` *fGetRelaxationFrequency()*

  Calculates relaxation frequency for a fluid using specified rheological models.

- `int` *fGetShearRateBGK()*

  Calculates shear rates at specified lattice site when using BGK collisions for compressible fluids.

- `int` *fGetShearRateBGKIncom()*

  Calculates shear rates at specified lattice site when using BGK collisions for incompressible fluids.

- `int` *fGetShearRateTRT()*

  Calculates shear rates at specified lattice site when using TRT collisions for compressible fluids.

- `int` *fGetShearRateTRTIncom()*

  Calculates shear rates at specified lattice site when using TRT collisions for incompressible fluids.

- `int` *fGetShearRateMRT()*

  Calculates shear rates at specified lattice site when using MRT collisions for compressible fluids.

- `int` *fGetShearRateMRTIncom()*

  Calculates shear rates at specified lattice site when using MRT collisions for incompressible fluids.

- `int` *fGetShearRateCLBED2Q9()*

  Calculates shear rates at specified lattice site when using CLBE collisions for compressible fluids with D2Q9 lattice.

- `int` *fGetShearRateCLBED3Q19()*

  Calculates shear rates at specified lattice site when using CLBE collisions for compressible fluids with D3Q19 lattice.

- `int` *fGetShearRateCLBED3Q27()*

  Calculates shear rates at specified lattice site when using CLBE collisions for compressible fluids with D3Q27 lattice.

- `int` *fGetShearRateBGKSwift()*

  Calculates shear rates at specified lattice site when using BGK collisions for compressible fluids with Swift free-energy interactions.

- `int` *fGetShearRateTRTSwift()*

  Calculates shear rates at specified lattice site when using TRT collisions for compressible fluids with Swift free-energy interactions.

- `int` *fGetShearRateMRTSwift()*

  Calculates shear rates at specified lattice site when using MRT collisions for compressible fluids with Swift free-energy interactions.

- `int` *fGetSystemOmega()*

  Calculates relaxation frequencies for all lattice sites based on shear rates and rheological models.

- int *fGetSystemOmegaSimple()*

  Calculates relaxation frequencies for all lattice sites based on rheological models without calculating shear rates.

5.18.2 Function Documentation

fGetRelaxationFrequency()

```
double fGetRelaxationFrequency (int typ,
                                double gamma,
                                double rho,
                                double a,
                                double b,
                                double c,
                                double d,
                                double n)
```

Calculates and returns the relaxation frequency of a fluid at a given lattice point from its local shear rate, using a rheological model that relates the shear stress on the fluid to the shear rate. The following rheological models are available:

- Simple Newtonian fluid (constant kinematic viscosity):

$$\mu = \rho\nu_0$$

- (Density-dependent) Newtonian fluid (constant dynamic viscosity):

$$\mu = \mu_0$$

- Power-law fluid:

$$\mu = K\dot{\gamma}^{n-1}$$

- Bingham plastic [11]:

$$\mu = \mu_0 + \frac{\sigma_y}{\dot{\gamma}}$$

- Herschel-Bulkley plastic [55]:

$$\mu = K\dot{\gamma}^{n-1} + \frac{\sigma_y}{\dot{\gamma}}$$

- Casson fluid [19]:

$$\mu = \left(\sqrt{\mu_0} + \sqrt{\frac{\sigma_y}{\dot{\gamma}}}\right)^2$$

- Carreau-Yasuda fluid [152]:

$$\mu = \mu_\infty + (\mu_0 - \mu_\infty)\left(1 + (\lambda\dot{\gamma})^d\right)^{\frac{n-1}{d}}$$

with an additional exponential decay function added to models with reciprocals of shear rates (Bingham plastic, Herschel-Bulkley plastic and Casson fluid) to avoid discontinuities when shear stresses are close to yield values [98].

Parameters

in	typ	Type of rheological model for specified fluid
in	gamma	Shear stress for fluid at lattice site $\dot{\gamma}$
in	rho	Density of fluid at lattice site
in	a	Parameter a for rheological model (ν_0, μ_0 or μ_∞)
in	b	Parameter b for rheological model (σ_y, μ_0 or $(\mu_0 - \mu_\infty)$)
in	c	Parameter c for rheological model (λ or exponential decay parameter for models with yield stresses)
in	d	Parameter d for rheological model ($2\sqrt{\mu_0 \sigma_y}$ or d)
in	n	Parameter n (power-law index) for rheological model (n)

fGetShearRateBGK()

```
int fGetShearRateBGK (double * shearrate, long tpos)
```

Calculates the shear rates of all fluids at a given lattice site by determining rate-of-strain tensors from locally-calculated momentum flux tensors when using BGK single relaxation time collisions for mildly compressible fluids:

$$S_{\alpha\beta} = \frac{3\omega}{2\rho\Delta t} \sum_i \left(f_i - f_i^{eq}\right) e_{i,\alpha} e_{i,\beta}$$

where the relaxation frequency for the previous timestep ω is used to convert the momentum flux tensor to a rate-of-strain tensor. (This is an iterative calculation since the resulting shear rates are used to calculate new relaxation frequencies, although convergence is normally obtained within a few timesteps.)

Parameters

out	shearrate	Shear rates for all fluids at specified grid point
in	tpos	Position of current boundary lattice site (in one-dimensional form)

fGetShearRateBGKIncom()

```
int fGetShearRateBGKIncom (double * shearrate, long tpos)
```

Calculates the shear rates of all fluids at a given lattice site by determining rate-of-strain tensors from locally-calculated momentum flux tensors when using BGK single relaxation time collisions for fully incompressible fluids:

$$S_{\alpha\beta} = \frac{3\omega}{2\rho\Delta t} \sum_i \left(f_i - f_i^{eq}\right) e_{i,\alpha} e_{i,\beta}$$

where the relaxation frequency for the previous timestep ω is used to convert the momentum flux tensor to a rate-of-strain tensor. (This is an iterative calculation since the resulting shear rates are used to calculate new relaxation frequencies, although convergence is normally obtained within a few timesteps.)

Parameters

out	shearrate	Shear rates for all fluids at specified grid point
in	tpos	Position of current boundary lattice site (in one-dimensional form)

fGetShearRateBGKSwift()

```
int fGetShearRateBGKSwift (double * shearrate, long tpos)
```

Calculates the shear rates of all fluids at a given lattice site by determining rate-of-strain tensors from locally-calculated momentum flux tensors when using BGK single relaxation time collisions for mildly compressible fluids with Swift free-energy interactions:

$$S_{\alpha\beta} = -\frac{3\omega}{2\rho\Delta t}\sum_i \left(f_i - f_i^{eq}\right) e_{i,\alpha} e_{i,\beta}$$

where the relaxation frequency for the previous timestep ω is used to convert the momentum flux tensor to a rate-of-strain tensor. (This is an iterative calculation since the resulting shear rates are used to calculate new relaxation frequencies, although convergence is normally obtained within a few timesteps.)

Parameters

out	shearrate	Shear rates for all fluids at specified grid point
in	tpos	Position of current boundary lattice site (in one-dimensional form)

fGetShearRateCLBED2Q9()

```
int fGetShearRateCLBED2Q9 (double * shearrate, long tpos)
```

Calculates the shear rates of all fluids at a given lattice site by determining rate-of-strain tensors from locally-calculated momentum flux tensors when using cascaded LBE (CLBE) collisions for mildly compressible fluids with the two-dimensional D2Q9 lattice:

$$S_{\alpha\beta} = -\frac{3}{2\rho\Delta t}\sum_i e_{i,\alpha} e_{i,\beta} \sum_j \left(\mathbf{T}^{-1}\mathbf{N}^{-1}\mathbf{\Lambda}\mathbf{N}\mathbf{T}\right)_{ij} \left(f_j - f_j^{eq}\right)$$

where the relaxation frequency for the previous timestep ω (along with other relaxation frequencies, including the values for bulk viscosity, third-order and fourth-order central moments) is used to convert the momentum flux tensor to a rate-of-strain tensor. (This is an iterative calculation since the resulting shear rates are used to calculate new relaxation frequencies, although convergence is normally obtained within a few timesteps.)

Parameters

out	shearrate	Shear rates for all fluids at specified grid point
in	tpos	Position of current boundary lattice site (in one-dimensional form)

fGetShearRateCLBED3Q19()

```
int fGetShearRateCLBED3Q19 (double * shearrate, long tpos)
```

Calculates the shear rates of all fluids at a given lattice site by determining rate-of-strain tensors from locally-calculated momentum flux tensors when using cascaded LBE (CLBE) collisions for mildly compressible fluids with the three-dimensional D3Q19 lattice:

$$S_{\alpha\beta} = -\frac{3}{2\rho\Delta t}\sum_i e_{i,\alpha} e_{i,\beta} \sum_j \left(\mathbf{T}^{-1}\mathbf{N}^{-1}\mathbf{\Lambda}\mathbf{N}\mathbf{T}\right)_{ij} \left(f_j - f_j^{eq}\right)$$

where the relaxation frequency for the previous timestep ω (along with other relaxation frequencies, including the values for bulk viscosity, third-order and fourth-order central moments) is used to convert the momentum flux tensor to a rate-of-strain tensor. (This is an iterative calculation since the resulting shear rates are used to calculate new relaxation frequencies, although convergence is normally obtained within a few timesteps.)

Parameters

out	shearrate	Shear rates for all fluids at specified grid point
in	tpos	Position of current boundary lattice site (in one-dimensional form)

fGetShearRateCLBED3Q27()

```
int fGetShearRateCLBED3Q27 (double * shearrate, long tpos)
```

Calculates the shear rates of all fluids at a given lattice site by determining rate-of-strain tensors from locally-calculated momentum flux tensors when using cascaded LBE (CLBE) collisions for mildly compressible fluids with the three-dimensional D3Q27 lattice:

$$S_{\alpha\beta} = -\frac{3}{2\rho\Delta t}\sum_i e_{i,\alpha}e_{i,\beta}\sum_j \left(\mathbf{T}^{-1}\mathbf{N}^{-1}\mathbf{\Lambda}\mathbf{N}\mathbf{T}\right)_{ij}\left(f_j - f_j^{eq}\right)$$

where the relaxation frequency for the previous timestep ω (along with other relaxation frequencies, including the values for bulk viscosity, third-order and fourth-order central moments) is used to convert the momentum flux tensor to a rate-of-strain tensor. (This is an iterative calculation since the resulting shear rates are used to calculate new relaxation frequencies, although convergence is normally obtained within a few timesteps.)

Parameters

out	shearrate	Shear rates for all fluids at specified grid point
in	tpos	Position of current boundary lattice site (in one-dimensional form)

fGetShearRateMRT()

```
int fGetShearRateMRT (double * shearrate, long tpos)
```

Calculates the shear rates of all fluids at a given lattice site by determining rate-of-strain tensors from locally-calculated momentum flux tensors when using multiple relaxation time (MRT) collisions [20] for mildly compressible fluids:

$$S_{\alpha\beta} = -\frac{3}{2\rho\Delta t}\sum_i e_{i,\alpha}e_{i,\beta}\sum_j \left(\mathbf{T}^{-1}\mathbf{\Lambda}\mathbf{T}\right)_{ij}\left(f_j - f_j^{eq}\right)$$

where the relaxation frequency for the previous timestep ω (along with other relaxation frequencies, including the value for bulk viscosity) is used to convert the momentum flux tensor to a rate-of-strain tensor. (This is an iterative calculation since the resulting shear rates are used to calculate new relaxation frequencies, although convergence is normally obtained within a few timesteps.)

Parameters

out	shearrate	Shear rates for all fluids at specified grid point
in	tpos	Position of current boundary lattice site (in one-dimensional form)

fGetShearRateMRTIncom()

```
int fGetShearRateMRTIncom (double * shearrate, long tpos)
```

Calculates the shear rates of all fluids at a given lattice site by determining rate-of-strain tensors from locally-calculated momentum flux tensors when using multiple relaxation time (MRT) collisions [20] for fully incompressible fluids:

$$S_{\alpha\beta} = -\frac{3}{2\rho\Delta t}\sum_i e_{i,\alpha}e_{i,\beta}\sum_j \left(\mathbf{T}^{-1}\mathbf{\Lambda}\mathbf{T}\right)_{ij}\left(f_j - f_j^{eq}\right)$$

where the relaxation frequency for the previous timestep ω (along with other relaxation frequencies, including the value for bulk viscosity) is used to convert the momentum flux tensor to a rate-of-strain tensor. (This is an iterative calculation since the resulting shear rates are used to calculate new relaxation frequencies, although convergence is normally obtained within a few timesteps.)

Parameters

out	shearrate	Shear rates for all fluids at specified grid point
in	tpos	Position of current boundary lattice site (in one-dimensional form)

fGetShearRateMRTSwift()

```
int fGetShearRateMRTSwift (double * shearrate, long tpos)
```

Calculates the shear rates of all fluids at a given lattice site by determining rate-of-strain tensors from locally-calculated momentum flux tensors when using multiple relaxation time (MRT) collisions [20] for mildly compressible fluids with Swift free-energy interactions:

$$S_{\alpha\beta} = -\frac{3}{2\rho\Delta t}\sum_i e_{i,\alpha}e_{i,\beta}\sum_j \left(\mathbf{T}^{-1}\mathbf{\Lambda}\mathbf{T}\right)_{ij}\left(f_j - f_j^{eq}\right)$$

where the relaxation frequency for the previous timestep ω (along with other relaxation frequencies, including the value for bulk viscosity) is used to convert the momentum flux tensor to a rate-of-strain tensor. (This is an iterative calculation since the resulting shear rates are used to calculate new relaxation frequencies, although convergence is normally obtained within a few timesteps.)

Parameters

out	shearrate	Shear rates for all fluids at specified grid point
in	tpos	Position of current boundary lattice site (in one-dimensional form)

fGetShearRateTRT()

```
int fGetShearRateTRT (double * shearrate, long tpos)
```

Calculates the shear rates of all fluids at a given lattice site by determining rate-of-strain tensors from locally-calculated momentum flux tensors when using two relaxation time (TRT) collisions for mildly compressible fluids:

$$S_{\alpha\beta} = -\frac{3}{2\rho\Delta t}\sum_i e_{i,\alpha}e_{i,\beta}\left[\omega_p\left(f_i - f_i^{eq}\right) + \omega_m\left(f_j - f_j^{eq}\right)\right]$$

where the symmetric relaxation frequency for the previous timestep ω^+ (along with the antisymmetric relaxation frequency calculated using the 'magic number') is used to convert the momentum flux tensor to a rate-of-strain tensor. (This is an iterative calculation since the resulting shear rates are used to calculate new relaxation frequencies, although convergence is normally obtained within a few timesteps.)

Parameters

out	shearrate	Shear rates for all fluids at specified grid point
in	tpos	Position of current boundary lattice site (in one-dimensional form)

fGetShearRateTRTIncom()

```
int fGetShearRateTRTIncom (double * shearrate, long tpos)
```

Calculates the shear rates of all fluids at a given lattice site by determining rate-of-strain tensors from locally-calculated momentum flux tensors when using two relaxation time (TRT) collisions for fully incompressible fluids:

$$S_{\alpha\beta} = -\frac{3}{2\rho\Delta t}\sum_i e_{i,\alpha}e_{i,\beta}\left[\omega_p\left(f_i - f_i^{eq}\right) + \omega_m\left(f_j - f_j^{eq}\right)\right]$$

where the symmetric relaxation frequency for the previous timestep ω^+ (along with the antisymmetric relaxation frequency calculated using the 'magic number') is used to convert the momentum flux tensor to a rate-of-strain tensor. (This is an iterative calculation since the resulting shear rates are used to calculate new relaxation frequencies, although convergence is normally obtained within a few timesteps.)

Parameters

out	shearrate	Shear rates for all fluids at specified grid point
in	tpos	Position of current boundary lattice site (in one-dimensional form)

fGetShearRateTRTSwift()

```
int fGetShearRateTRTSwift (double * shearrate, long tpos)
```

Calculates the shear rates of all fluids at a given lattice site by determining rate-of-strain tensors from locally-calculated momentum flux tensors when using two relaxation time (TRT) collisions for mildly compressible fluids with Swift free-energy interactions:

$$S_{\alpha\beta} = -\frac{3}{2\rho\Delta t}\sum_i e_{i,\alpha}e_{i,\beta}\left[\omega_p\left(f_i - f_i^{eq}\right) + \omega_m\left(f_j - f_j^{eq}\right)\right]$$

where the symmetric relaxation frequency for the previous timestep ω^+ (along with the antisymmetric relaxation frequency calculated using the 'magic number') is used to convert the momentum flux tensor to a rate-of-strain tensor. (This is an iterative calculation since the resulting shear rates are used to calculate new relaxation frequencies, although convergence is normally obtained within a few timesteps.)

Parameters

out	shearrate	Shear rates for all fluids at specified grid point
in	tpos	Position of current boundary lattice site (in one-dimensional form)

fGetSystemOmega()

```
int fGetSystemOmega ()
```

Calculates the shear rates of all fluids at all lattice sites and the relaxation frequencies for the fluids using the shear rates with rheological models. This subroutine selects which shear rate calculation routine to use and how to calculate the relaxation frequencies based on collision type, whether or not the fluids are compressible and whether or not Swift free-energy interactions are in use. Since shear rates are calculated locally at each lattice site, this routine can be applied to all lattice sites (including boundary halo points).

fGetSystemOmegaSimple()

```
int fGetSystemOmegaSimple ()
```

Calculates the relaxation frequencies of all fluids at all lattice sites using rheological models that do not depend on shear rate (i.e. Newtonian fluid models for constant kinematic or dynamic viscosity). This subroutine selects how to calculate the relaxation frequencies based on whether or not the fluids are compressible and whether or not Swift free-energy interactions are in use. This routine can be applied to all lattice sites (including boundary halo points).

5.19 lbpIO.cpp

Module with routines to read input files, print simulation information to standard output and write simulation snapshots. (Header file available as lbpIO.hpp.)

Subroutines to read input files and set up LBE simulations based on user specifications, print summary of simulation input, fluid masses and system momentum periodically, and write snapshots of simulation to output files.

5.19.1 Functions

- `int` *fDefineSystem()*

  Reads in essential LBE calculation parameters from a system data file.

- `int` *fInputParameters()*

  Reads system parameters from system data file.

- `int` *fReadSpace2D()*

  Reads space parameters (boundary conditions) from a data file for a two-dimensional system.

- `int` *fReadSpace3D()*

  Reads space parameters (boundary conditions) from a data file for a three-dimensional system.

- `int` *fReadSpaceParameter()*

  Reads space parameters (boundary conditions) from a data file.

- `int` *fReadInitialState2D()*

  Reads initial simulation state from a data file for a two-dimensional system.

- `int` *fReadInitialState3D()*

  Reads initial simulation state from a data file for a three-dimensional system.

- `int` *fReadInitialState()*

  Reads initial simulation state from a data file.

- `int` *fSetoffSteer()*

  Creates file to prevent reading in input files when using computational steering (in serial).

- `int` *fCheckSteer()*

  Checks for file indicating steering is occurring and reads in input files if it does not exist (in serial).

- `int` *fPrintSystemInfo()*

  Prints system information to standard output prior to commencing LBE simulation.

- `int` *fPrintParameters()*

  Prints parameters for LBE simulation to standard output.

- `int` *fPrintEndEquilibration()*

  Prints message indicating end of system equilibration.

- `int` *fPrintEarlyTermination()*

  Prints message indicating simulation has been terminated early.

- `int` *fPrintDomainMass()*

  Calculates and prints total and individual fluid masses in subdomain.

- `int` *fPrintDomainMomentum()*

  Calculates and prints total fluid momentum in subdomain.

- `int` *fsCreateIOGroups()*

  Creates I/O group to gather together output data for writing to files during serial calculations.

- `int` *fOutput()*

  Outputs all system data in the required format at user-specified intervals.

5.19.2 Function Documentation

fCheckSteer()

```
int fCheckSteer ()
```

Checks for the existence of a file called notsteer, which was created to prevent DL_MESO_LBE from starting a new simulation when computaional steering is applied. If the files does not exist, read in system and space property files. This routine is for serial calculations: an atlnerative routine exists for parallel running - *fMPICheckSteer()* - but neither routine is currently in use in the main DL_MESO_LBE code.

fDefineSystem()

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

Reads calculation parameters (lattice scheme, types of collision and forcing, mesophase interaction algorithms, numbers of fluids, solutes, temperature scalars and phase field order parameters, the size of the grid, if fluids are fully incompressible, boundary halo size, whether a simulation is being restarted, output file type) from an input system file. Checks are carried out to ensure the selected combinations of lattice scheme, collisions, mesophase interactio algorithms etc. are viable for calculations. The lattice scheme, numbers of fluids, solutes, temperature fields and phase fields, grid size and boundary halo size must be specified and read by this subroutine.

Parameters

in	filename	Name of input system file (default: lbin.sys)

fInputParameters()

```
int fInputParameters (const char * filename = "lbin.sys")
```

Reads additional parameters for LBE simulation (e.g. numbers of timesteps, relaxation times for fluids, initial and boundary conditions for system, options to combine output files among processors) from an input system file. Some basic checks are made to ensure the simulation can proceed, and any missing parameters (e.g. critical temperature and pressure for equations of state) are either calculated using available user inputs or revert to default values (e.g. tuneable collision parameters for MRT collisions).

Parameters

in	filename	Name of input system file (default: lbin.sys)

fOutput()

```
int fOutput (const char * filename = "lbout")
```

Writes macroscopic system data (fluid densities, mass fractions, velocities, solute concentrations, temperatures, boundary conditions) to output files in the required format (XML-based VTK, Legacy VTK or Plot3D in either big endian binary or ANSI text) at user-specified intervals. The filenames for these files start with a given string and include 6-digit numbers to indicate snapshot number: if more than one file is produced per snapshot (by multiple processor cores), additional numbers are used in the filename to indicate which processor or group of processors has written the data in preparation for post-simulation gathering. The solution files for Plot3D can each hold only one main property (density, mass fraction etc.), so at least one file will also be produced for each property.

Parameters

in	filename	Beginning of name for output files (default: lbout)

fPrintDomainMass()

```
int fPrintDomainMass ()
```

Calculates both the total mass and the individual masses of all fluids in the simulation subdomain for the current processor and prints the results to the standard output. This subroutine will only produce the total and individual fluid masses for the entire system if running on a single processor (in serial): an alternative routine - *fPrintSystemMass()* - is available for printing system-wide total and individual fluid masses in parallel.

fPrintDomainMomentum()

```
int fPrintDomainMomentum ()
```

Calculates the total momentum of all fluids in the simulation subdomain for the current processor and prints the result to the standard output. This subroutine will only produce the total fluid momentume for the entire system if running on a single processor (in serial): an alternative routine - *fPrintSystemMomentum()* - is available for printing the system-wide total momentum in parallel.

fPrintEarlyTermination()

```
int fPrintEarlyTermination ()
```

Prints a message to the standard output (often the screen), indicating that the simulation has been terminated after the specified runtime (in seconds) has come to an end and the number of timesteps that have been completed up to this point. This message is not printed if no simulation runtime has been supplied or the simulation stops due to a runtime error.

fPrintEndEquilibration()

```
int fPrintEndEquilibration ()
```

Prints a message to the standard output (often the screen), indicating that all equilibration timesteps have been completed. This message is not printed if no equilibration timesteps are requested for the simulation.

fPrintParameters()

```
int fPrintParameters ()
```

Prints parameters for fluids, solutes, temperatures, interactions and rheological models - all obtained from input system file - to the standard output (often the screen). For simulations using equations of state and constant system temperatures, the attractive terms for certain equations of state (Redlich-Kwong, Carnahan-Starling-Redlich-Kwong, Soave-Redlich-Kwong and Peng-Robinson) are adjusted to incorporate constant temperatures and avoid recalculating this dependence for each lattice point and timestep.

fPrintSystemInfo()

```
int fPrintSystemInfo ()
```

Prints information about the LBE simulation about to start - including the system size, lattice scheme, numbers of fluids, solutes and temperature scalars, types of collision, forcing and mesoscopic interactions - to the standard output (often the screen).

fReadInitialState()

```
int fReadInitialState (const char * filename = "lbin.init")
```

Reads a initial state input file and replaces default values of initial velocities, fluid densities, solute concentrations and temperature with values supplied for specified grid points.

Parameters

in	filename	Name of initial state input file (default: lbin.init)

fReadInitialState2D()

```
int fReadInitialState2D (const char * filename = "lbin.init")
```

Reads a initial state input file and replaces default values of initial velocities, fluid densities, solute concentrations and temperature with values supplied for specified grid points in a two-dimensional system: only grid points with $z = 0$ are accepted in this case. Local equilibrium distribution functions for the required properties at each specified grid point are calculated to replace values previously calculated using default initial properties given in the system data file.

Parameters

in	filename	Name of initial state input file (default: lbin.init)

fReadInitialState3D()

```
int fReadInitialState3D (const char * filename = "lbin.init")
```

Reads a initial state input file and replaces default values of initial velocities, fluid densities, solute concentrations and temperature with values supplied for specified grid points in a three-dimensional system. Local equilibrium distribution functions for the required properties at each specified grid point are calculated to replace values previously calculated using default initial properties given in the system data file.

Parameters

in	filename	Name of initial state input file (default: lbin.init)

fReadSpace2D()

```
int fReadSpace2D (const char * filename = "lbin.spa")
```

Reads a space parameter input file and assigns boundary conditions to specified grid points (supplied as numerical codes indicating type and direction) for a two-dimensional system: only grid points with $z = 0$ are accepted in this case. If the system is being equilibrated, no boundary conditions other than blank sites, bounce-back and outflows are applied until the equilibration period has come to an end, which allows immiscible fluid drops to settle before flow fields are applied.

Parameters

in	filename	Name of space parameter input file (default: lbin.spa)

fReadSpace3D()

```
int fReadSpace3D (const char * filename = "lbin.spa")
```

Reads a space parameter input file and assigns boundary conditions to specified grid points (supplied as numerical codes indicating type and direction) for a three-dimensional system. If the system is being equilibrated, no boundary conditions other than blank sites, bounce-back and outflows are applied until the equilibration period has come to an end, which allows immiscible fluid drops to settle before flow fields are applied.

Parameters

in	filename	Name of space parameter input file (default: lbin.spa)

fReadSpaceParameter()

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

Reads a space parameter input file and assigns boundary conditions to specified grid points (supplied as numerical codes indicating type and direction) for a three-dimensional system.

Parameters

in	filename	Name of space parameter input file (default: lbin.spa)

fsCreateIOGroups()

```
int fsCreateIOGroups ()
```

Sets up the various properties normally required for a group of processors sharing and writing data to output files for a single processor (running in serial). This subroutine is a serial equivalent of *fCreateIOGroups()* and populates the required properties in a structure for storing information on data gathering prior to writing to files.

fSetoffSteer()

```
int fSetoffSteer ()
```

Creates a file called notsteer to prevent DL_MESO_LBE from starting a new simulation by reading in system and space property files when a LBE simulation is computationally steered. This routine is for serial calculations: an alternative routine exists for parallel running (*fMPISetoffSteer()*), but neither routine is currently in use in the main DL_MESO_LBE code.

5.20 lbpIOAGGPAR.cpp and lbpIOAGGSER.cpp

Modules with routines to gather together data for writing output files and reading/writing restart files when running in parallel (lbpIOAGGPAR.cpp) or serial (lbpIOAGGSER.cpp). (Header files available at lbpIOAGGPAR.hpp and lbpIOAGGSER.hpp).

Subroutines to gather together system properties (fluid densities, mass fractions, velocities, solute concentrations, temperatures, boundary conditions) to write to output files, to write gathered data in the required forms for the available file formats, to read and write files to enable simulation restarts. lbpIOAGGPAR.cpp includes subroutines for simulations carried out on multiple processors (in parallel), lbpIOAGGSER.cpp has subroutines for simulations carried out on a single processor (in serial).

5.20.1 Functions

- `float` *fGetTemperatureWrap()*

  Obtains temperature at given lattice site.

- `float` *fGetOneMassSiteWrap()*

  Obtains density of specific fluid at given lattice site.

- `float` *fGetOneMassSwiftSiteWrap()*

  Obtains density of specific fluid at given lattice site when using Swift free-energy interactions.

- `float` *fGetFracSiteWrap()*

  Obtains mass fraction of specific fluid at given lattice site.

- `float` *fGetFracSwiftSiteWrap()*

  Obtains mass fraction of specific fluid at given lattice site when using Swift free-energy interactions.

- `float` *fGetOneConcSiteWrap()*

  Obtains concentration of specific solute at given lattice site.

- `void` *fPieceRangeLocal()*

  Determines extent of lattice covered by current processor in local coordinates.

- `void` *fPieceRangeGlobal()*

  Determines extent of lattice covered by current processor in global coordinates.

- `int` *fPieceDataPoints()*

  Determines number of data points for subdomain in each dimension.

- `void` *fFillBuffer()*

  Fills floating-point buffer with data for grid points.

- `void` *fPieceDensities()*

  Fills floating-point buffer with densities of specified fluid at grid points in current processor.

- `void` *fPieceDensitiesSwift()*

  Fills floating-point buffer with densities of specified fluid at grid points in current processor when using Swift free-energy interactions.

- `void` *fPieceMassFractions()*

  Fills floating-point buffer with mass fractions of specified fluid at grid points in current processor.

- `void`*fPieceMassFractionsSwift()*

  Fills floating-point buffer with densities of specified fluid at grid points in current processor when using Swift free-energy interactions.

- `void` *fPieceSoluteConcentrations()*

  Fills floating-point buffer with concentrations of specified solute at grid points in current processor.

- `void` *fPieceTemperatures()*

  Fills floating-point buffer with temperatures at grid points in current processor.

- `void` *fPieceVelocity()*

  Fills floating-point buffer with a single component of fluid velocities at grid points in current processor.

- `void` *fPieceVelocities()*

  Fills floating-point buffer with fluid velocities at grid points in current processor.

- `void` *fPiecePhaseField()*

  Fills integer buffer with boundary conditions (phase fields) at grid points in current processor.

- `void` *fPiecePhaseFieldFloat()*

  Fills single-precision floating-point buffer with boundary conditions (phase fields) at grid points in current processor.

- `void` *fPieceGridPoints()*

  Fills floating-point buffer with coordinates of grid points in current processor.

- `void` *fPieceGridPointComponent()*

  Fills floating-point buffer with a single component of coordinates of grid points in current processor.

- `void` *fGroupRangeGlobal()*

  Finds minimum and maximum grid coordinates for I/O group of processors.

- `void` *fGroupPieceRangeLocal()*

  Returns range of grid points covered by an I/O group on a local basis.

- `void` *fGroupPieceRangeGlobal()*

  Returns range of grid points covered by an I/O group on a global basis.

- `int` *fGroupPieceDataPoints()*

  Determines number of data points for I/O group in each dimension.

- `int` *fGroupPieceGatherInfo()*

  Gathers information about data being gathered together by I/O group among its processors.

- `void` *fGroupGatherFloatData()*

  Fills single-precision floating-point buffer with data from across entire I/O group.

- `void` *fGroupGatherIntData()*

  Fills integer buffer with data from across entire I/O group.

- `void` *fGroupGatherRestartData()*

  Fills double-precision floating-point buffer with data from across entire I/O group required for simulation restart.

- `void` *fGroupDensities()*

  Pulls together densities for specific fluid over lattice section covered by I/O group.

- `void` *fGroupMassFractions()*

  Pulls together mass fractions for specific fluid over lattice section covered by I/O group.

- `void` *fGroupSoluteConcentrations()*

  Pulls together concentrations for specific solute over lattice section covered by I/O group.

- `void` *fGroupTemperatures()*

  Pulls together temperatures over lattice section covered by I/O group.

- `void` *fGroupPhaseField()*

  Pulls together boundary conditions (phase fields) over lattice section covered by I/O group (as integers).

- `void` *fGroupPhaseFieldFloat()*

  Pulls together boundary conditions (phase fields) over lattice section covered by I/O group (as single-precision floating-point numbers).

- `void` *fGroupVelocityComponent()*

  Pulls together values of a single velocity component over lattice section covered by I/O group.

- `void` *fGroupVelocities()*

  Pulls together fluid velocities over lattice section covered by I/O group.

- `void` *fGroupGridPoints()*

  Pulls together coordinates over lattice section covered by I/O group.

- `void` *fGroupGridPointComponent()*

  Pulls together single component of coordinates over lattice section covered by I/O group.

- `int` *fOpenOutputBinaryFile()*

  Opens binary file for simulation output.

- `int` *fCloseOutputANSIFile()*

  Closes ANSI text file for simulation output.

- `int` *fOpenOutputANSIFile()*

  Opens ANSI text file for simulation output.

- `int` *fCloseOutputBinaryFile()*

  Closes binary file for simulation output.

- `int` *fWriteVTKFloatBinaryData()*

  Writes a block of single-precision floating-point data to a binary XML-based structured-grid VTK file.

- `int` *fWriteVTKIntegerBinaryData()*

  Writes a block of integer data to a binary XML-based structured-grid VTK file.

- `int` *fWriteVTKFloatANSIData()*

  Writes a block of single-precision floating-point data to an ANSI text XML-based structured-grid VTK file.

- `int` *fWriteVTKIntegerANSIData()*

  Writes a block of integer data to an ANSI text XML-based structured-grid VTK file.

- `int` *fWriteLegacyVTKFloatBinaryData()*

  Writes a block of single-precision floating-point data to a binary structured-grid Legacy VTK file.

- `int` *fWriteLegacyVTKIntegerBinaryData()*

  Writes a block of integer data to a binary structured-grid Legacy VTK file.

- `int` *fWriteLegacyVTKFloatANSIData()*

  Writes a block of single-precision floating-point data to an ANSI text structured-grid Legacy VTK file.

- `int` *fWriteLegacyVTKIntegerANSIData()*

  Writes a block of integer data to an ANSI text structured-grid Legacy VTK file.

- `int` *fWritePlot3DGridFloatBinaryData()*

  Writes a block of single-precision floating-point data to a binary Plot3D file.

- `int` *fWritePlot3DGridIntegerBinaryData()*

  Writes a block of integer data to a binary Plot3D file.

- `int` *fWritePlot3DGridFloatANSIData()*

  Writes a block of single-precision floating-point data to an ANSI text Plot3D file.

- `int` *fWritePlot3DGridIntegerANSIData()*

  Writes a block of integer data to an ANSI text Plot3D file.

- `int` *fReadRestart()*

  Reads simulation state from restart file to resume a previous LBE simulation.

- `int` *fWriteRestart()*

  Writes current simulation state to a restart file.

- `void` *fGroupCartesianCommRange()*

  Determines extent of current I/O group among available processors to create its communicator.

- `int` *fCreateIOGroups()*

  Creates I/O groups to gather together output data for writing to files during parallel calculations.

- `int` *fOutputInfo()*

  Writes information required to process separate output files after simulations.

5.20.2 Function Documentation

fCloseOutputANSIFile()

```
int fCloseOutputANSIFile (ofstream & file)
```

Closes an ANSI/text file used to write simulation outputs. If running in serial or not gathering data in all dimensions, a standard text filestream is closed: if combining data in all dimensions, MPI-IO is used to close the file based on the previously-generated file handle (used to write data to the file). Only the root processor for an I/O group will close the file.

Parameters

in	file	Output filestream when not using MPI-IO

fCloseOutputBinaryFile()

```
int fCloseOutputBinaryFile (ofstream & file)
```

Closes a binary file used to write simulation outputs. If running in serial or not gathering data in all dimensions, a standard binary filestream is closed: if combining data in all dimensions, MPI-IO is used to close the file based on the previously-generated file handle (used to write data to the file). Only the root processor for an I/O group will close the file.

Parameters

in	file	Output filestream when not using MPI-IO

fCreateIOGroups()

```
int fCreateIOGroups ()
```

Sets up the I/O groups among processors needed to gather together and write data to output files during parallel LBE simulations. (A serial version of this subroutine - *fsCreateIOGroups()* - is available in *lbpIO.cpp* to do the same for single processor simulations.) This subroutine creates a Cartesian communicator among the available processors (based on the total number of processors for each direction) and checks the coordinates of each processor match their location within the entire lattice. I/O groups based on data being shared in selected directions are then created, identifying the number of processors and root processors for each group and the rank (number) for the current processor in its I/O group, as well as the extent of the lattice for each I/O group, before creating MPI communicators to gather data among each group and for all I/O group root processors to write their data to output files. As data is normally written with the x-coordinate incrementing most quickly (followed by the y-coordinate and then the z-coordinate) in the output files but may not be gathered together in this order, an array is also created to specify where each received data value is to be written in the output file, which facilitates reordering the data values after gathering and before writing.

fFillBuffer()

```
void fFillBuffer (float * buffer,
                  int * start,
                  int * end,
                  int iprop,
                  bool swapbit,
                  float(*)(int, int) GetFunction)
```

Fills a single-precision floating-point data buffer with values at a range of lattice points representing a piece of the entire lattice, specifying the lattice extent, property and whether or not to swap bits (to obtain required endianness when writing to files).

Parameters

out	buffer	Single-precision floating-point array with data values for specified lattice points
in	start	Starting (local) coordinates of the lattice section (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section (front-right-front corner)
in	iprop	Number of fluid/solute/temperature scalar for required property to fill array
in	swapbit	Flag for swapping bits of each data point
in	GetFunction	Wrapped function to obtain data value for each lattice point

fGetFracSiteWrap()

```
float fGetFracSiteWrap (int ilen, int iprop)
```

Returns mass fraction of a selected fluid at a given lattice site as a single-precision float. This function is a wrapper for the actual function that calculates the mass fraction, and is used when gathering values together for writing to output files.

Parameters

in	ilen	Position of lattice site in one-dimensional form
in	iprop	Number of fluid for required mass fraction

fGetFracSwiftSiteWrap()

```
float fGetFracSwiftSiteWrap (int ilen, int iprop)
```

Returns mass fraction of a selected fluid at a given lattice site when using Swift free-energy interactions as a single-precision float. This function is a wrapper for the actual function that calculates the mass fraction, and is used when gathering values together for writing to output files.

Parameters

in	ilen	Position of lattice site in one-dimensional form
in	iprop	Number of fluid for required mass fraction

fGetOneConcSiteWrap()

```
float fGetOneConcSiteWrap (int ilen, int iprop)
```

Returns concentration of a selected solute at a given lattice site as a single-precision float. This function is a wrapper for the actual function that calculates the concentration, and is used when gathering values together for writing to output files.

Parameters

in	ilen	Position of lattice site in one-dimensional form
in	iprop	Number of solute for required concentration

fGetOneMassSiteWrap()

```
float fGetOneMassSiteWrap (int ilen, int iprop)
```

Returns density of a selected fluid at a given lattice site as a single-precision float. This function is a wrapper for the actual function that calculates the density, and is used when gathering values together for writing to output files.

Parameters

in	ilen	Position of lattice site in one-dimensional form
in	iprop	Number of fluid for required density

fGetOneMassSwiftSiteWrap()

```
float fGetOneMassSwiftSiteWrap (int ilen, int iprop)
```

Returns density of a selected fluid at a given lattice site when using Swift free-energy interactions as a single-precision float. This function is a wrapper for the actual functions that calculate the density, and is used when gathering values together for writing to output files.

Parameters

in	ilen	Position of lattice site in one-dimensional form
in	iprop	Number of fluid for required density

fGetTemperatureWrap()

```
float fGetTemperatureWrap (int ilen, int iprop)
```

Returns temperature for a given lattice site as a single-precision float. This function is a wrapper for the actual function that calculates the temperature, and is used when gathering values together for writing to output files.

Parameters

in	ilen	Position of lattice site in one-dimensional form
in	iprop	Number of temperature scalar (unused dummy value for this function)

fGroupCartesianCommRange()

```
void fGroupCartesianCommRange (int * start, int * end, sIOGroup * info)
```

Finds the range of the Cartesian communicator grid covered by the current I/O group, i.e. find the extent of processors in each direction for the group in terms of the total numbers of processors in each direction. (Only needed for parallel calculations.)

Parameters

out	start	Starting coordinates of the I/O group in terms of numbers of processors (bottom-left-back corner)
out	end	Ending coordinates of the I/O group in terms of numbers of processors (top-right-front corner)
in	info	I/O group information (communicator for I/O group)

fGroupDensities()

```
void fGroupDensities (float * buffer,
                      int * start,
                      int * end,
                      int iprop,
                      bool swapbit)
```

Fills a single-precision floating-point data buffer array for each I/O group with densities for a specified fluid.

Parameters

out	buffer	Array of fluid densities gathered among I/O group (only on group's root processor)
in	start	Starting (local) coordinates of the lattice section for the I/O group (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section for the I/O group (front-right-front corner)
in	iprop	Number of fluid for required density
in	swapbit	Flag for swapping bits of each data point

fGroupGatherFloatData()

```
void fGroupGatherFloatData (sIOGroup * ioGroup,
                            float * buffer,
                            int * start,
                            int * end,
                            int dataPerPoint,
                            int param,
                            bool swapbit,
                            void(*)(float *, int *, int *, int, bool) GetLocalData)
```

Fills a single-precision floating-point data buffer array for each I/O group with the values of a required property by filling arrays for each processor with values and gathering the data together into the group's buffer (stored on the root processor for the group).

Parameters

in	ioGroup	I/O group information (flags indicating data gather across processors in each dimension, number of processors in group, MPI communicator for group, ordering of lattice points for file writing)
out	buffer	Array of data gathered among I/O group (only on group's root processor)
in	start	Starting (local) coordinates of the lattice section for the I/O group (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section for the I/O group (front-right-front corner)
in	data-Per-Point	Number of data values per lattice point
in	param	Parameter denoting property number (for fluid density, mass fraction or solute concentration, dummy value for other data)
in	swap-bit	Flag for swapping bits of each data point
in	Get-Local-Data	Function to fill array for each processor in the I/O group (i.e. a piece of the I/O group's lattice) with values for the required property

fGroupGatherIntData()

```
void fGroupGatherIntData (sIOGroup * ioGroup,
                          int * buffer,
                          int * start,
                          int * end,
                          int dataPerPoint,
                          int param,
                          bool swapbit,
                          void(*)(int *, int *, int *, int, bool) GetLocalData)
```

Fills an integer data buffer array for each I/O group with the values of a required property by filling arrays for each processor with values and gathering the data together into the group's buffer (stored on the root processor for the group).

Parameters

in	ioGroup	I/O group information (flags indicating data gather across processors in each dimension, number of processors in group, MPI communicator for group, ordering of lattice points for file writing)
out	buffer	Array of data gathered among I/O group (only on group's root processor)
in	start	Starting (local) coordinates of the lattice section for the I/O group (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section for the I/O group (front-right-front corner)
in	data-Per-Point	Number of data values per lattice point
in	param	Parameter denoting property number (dummy value)
in	swap-bit	Flag for swapping bits of each data point
in	Get-Local-Data	Function to fill array for each processor in the I/O group (i.e. a piece of the I/O group's lattice) with values for the required property

fGroupGatherRestartData()

```
void fGroupGatherRestartData (sIOGroup * ioGroup,
                              double * buffer,
                              int * posbuffer)
```

Fills a double-precision floating-point data buffer array for each I/O group with distribution functions and local relaxation frequencies for all lattice points, as well as an integer array indicating the one-dimensional grid points (in global terms) for each point in the group's section of lattice. by filling arrays for each processor with the required values and gathering the data together into the group's buffers (stored on the root processor for the group), which will be written to a file to enable simulation restarts. (This subroutine is only required when running in parallel.)

Parameters

in	ioGroup	I/O group information (MPI communicator for group)
out	buffer	Array of distributions functions and local relaxation frequencies gathered among I/O group (only on group's root processor)
out	pos-buffer	Array of one-dimensional lattice positions gathered among I/O group (only on group's root processor)

fGroupGridPointComponent()

```
void fGroupGridPointComponent (float * buffer,
                               int * start,
                               int * end,
                               int comp,
                               bool swapbit)
```

Fills a single-precision floating-point data buffer for each I/O with a specified component of coordinates (at 'real-world' scale).

Parameters

out	buffer	Array of specified lattice point coordinates gathered among I/O group (only on group's root processor)
in	start	Starting (local) coordinates of the lattice section (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section (front-right-front corner)
in	comp	Required coordinate components (0 = x, 1 = y, 2 = z)
in	swap-bit	Flag for swapping bits of each data point

fGroupGridPoints()

```
void fGroupGridPoints (float * buffer,
                       int * start,
                       int * end,
                       bool swapbit)
```

Fills a single-precision floating-point data buffer for each I/O group with all three components of coordinates (at 'real-world' scale).

Parameters

out	buffer	Array of lattice point coordinates gathered among I/O group (only on group's root processor)
in	start	Starting (local) coordinates of the lattice section for the I/O group (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section for the I/O group (front-right-front corner)
in	swap-bit	Flag for swapping bits of each data point

fGroupMassFractions()

```
void fGroupMassFractions (float * buffer,
                          int * start,
                          int * end,
                          int iprop,
                          bool swapbit)
```

Fills a single-precision floating-point data buffer array for each I/O group with densities for a specified fluid.

Parameters

out	buffer	Array of fluid mass fractions gathered among I/O group (only on group's root processor)
in	start	Starting (local) coordinates of the lattice section for the I/O group (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section for the I/O group (front-right-front corner)
in	iprop	Number of fluid for required mass fraction
in	swapbit	Flag for swapping bits of each data point

fGroupPhaseField()

```
void fGroupPhaseField (int * buffer,
                       int * start,
                       int * end,
                       bool swapbit)
```

Fills an integer data buffer array for each I/O group with boundary conditions (phase fields).

Parameters

out	buffer	Array of boundary conditions (phase fields) gathered among I/O group (only on group's root processor)
in	start	Starting (local) coordinates of the lattice section for the I/O group (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section for the I/O group (front-right-front corner)
in	swap-bit	Flag for swapping bits of each data point

fGroupPhaseFieldFloat()

```
void fGroupPhaseFieldFloat (float * buffer,
                            int * start,
                            int * end,
                            bool swapbit)
```

Fills a single-precision floating-point data buffer array for each I/O group with boundary conditions (phase fields).

Parameters

out	buffer	Array of boundary conditions (phase fields) gathered among I/O group (only on group's root processor)
in	start	Starting (local) coordinates of the lattice section for the I/O group (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section for the I/O group (front-right-front corner)
in	swap-bit	Flag for swapping bits of each data point

fGroupPieceDataPoints()

```
int fGroupPieceDataPoints (int * length)
```

Calculates the number of lattice points (and therefore number of data points) in each dimension of the piece of the simulation box held by the I/O group, based on the provided extents, and returns the total number.

Parameters

out	length	Number of lattice points in section for each direction

fGroupPieceGatherInfo()

```
int fGroupPieceGatherInfo (sIOGroup * ioGroup,
                           int numberOfPoints,
                           int dataPerPoint,
                           int * data,
                           int * displacements)
```

Pulls together information required for gathering a data set among processors within an I/O group: the number of data points in the group and the displacements required to position each processor's data when gathering together (only on group's root processor, i.e. the processor that will receive and write the data to output files). This function returns the total number of data points for the group only on the I/O group's root processor. (This subroutine is only required when running in parallel.)

Parameters

in	ioGroup	I/O group information (number of processors in group, processor numbers within group, MPI communicator for group)
in	numberOf-Points	Number of lattice points held by processor
in	dataPer-Point	Number of values required per lattice point
out	data	Numbers of data values for each processor in the I/O group
out	displace-ments	Displacements for positioning received data (locations for each processor's data in the root processor's array with gathered data)

fGroupPieceRangeGlobal()

```
void fGroupPieceRangeGlobal (int * start, int * end)
```

Determines the range of an I/O group's section of the lattice, outputting the global coordinates of this range.

Parameters

out	start	Starting (global) coordinates of the lattice section (bottom-left-back corner)
out	end	Ending (global) coordinates of the lattice section (front-right-front corner)

fGroupPieceRangeLocal()

```
void fGroupPieceRangeLocal (int * start, int * end)
```

Determines the range of an I/O group's section of the lattice, outputting the local coordinates of this range with adjustments for boundary halos if at the edge of entire lattice and/or using MPI-IO.

Parameters

out	start	Starting (local) coordinates of the lattice section (bottom-left-back corner)
out	end	Ending (local) coordinates of the lattice section (front-right-front corner)

fGroupRangeGlobal()

```
void fGroupRangeGlobal (sIOGroup * ioGroup)
```

Finds the minimum (bottom-left-back corner) and maximum (top-right-front corner) grid global coordinates for the section of lattice covered by the processors in an I/O group (which will share data and write an output file per simulation snapshot). This determination only needs to be carried out once per simulation: an internal switch in the subroutine will skip over this if it is called after the first time. The serial version of this subroutine uses the extent determined for the only processor, while the parallel version reduces the minimum and maximum grid coordinates over all processors in the I/O group.

Parameters

out	ioGroup	I/O group information (start and end global coordinates for group)

fGroupSoluteConcentrations()

```
void fGroupSoluteConcentrations (float * buffer,
                                  int * start,
                                  int * end,
                                  int iprop,
                                  bool swapbit)
```

Fills a single-precision floating-point data buffer array for each I/O group with concetrations for a specified solute.

Parameters

out	buffer	Array of solute concentrations gathered among I/O group (only on group's root processor)
in	start	Starting (local) coordinates of the lattice section for the I/O group (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section for the I/O group (front-right-front corner)
in	iprop	Number of solute for required concentration
in	swapbit	Flag for swapping bits of each data point

fGroupTemperatures()

```
void fGroupTemperatures (float * buffer,
                         int * start,
                         int * end,
                         bool swapbit)
```

Fills a single-precision floating-point data buffer array for each I/O group with temperatures.

Parameters

out	buffer	Array of temperatures gathered among I/O group (only on group's root processor)
in	start	Starting (local) coordinates of the lattice section for the I/O group (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section for the I/O group (front-right-front corner)
in	swapbit	Flag for swapping bits of each data point

fGroupVelocities()

```
void fGroupVelocities (float * buffer,
                       int * start,
                       int * end,
                       bool swapbit)
```

Fills a single-precision floating-point data buffer array for each I/O group with all three components of fluid velocity.

Parameters

out	buffer	Array of velocities gathered among I/O group (only on group's root processor)
in	start	Starting (local) coordinates of the lattice section for the I/O group (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section for the I/O group (front-right-front corner)
in	swapbit	Flag for swapping bits of each data point

fGroupVelocityComponent()

```
void fGroupVelocityComponent (float * buffer,
                              int * start,
                              int * end,
                              int comp,
                              bool swapbit)
```

Fills a single-precision floating-point data buffer array for each I/O group with values for a single component of fluid velocity.

Parameters

out	buffer	Array of velocity components gathered among I/O group (only on group's root processor)
in	start	Starting (local) coordinates of the lattice section for the I/O group (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section for the I/O group (front-right-front corner)
in	comp	Required velocity components (0 = x, 1 = y, 2 = z)
in	swapbit	Flag for swapping bits of each data point

fOpenOutputANSIFile()

```
int fOpenOutputANSIFile (const char * filename, ofstream & file)
```

Opens an ANSI/text file to write simulation outputs. If running in serial or not gathering data in all dimensions, a standard text filestream is opened: if combining data in all dimensions, MPI-IO is used to open the file and generate the required output file handle (used to write data to the file). Only the root processor for an I/O group will open the file.

Parameters

in	filename	Name of output file to be opened
out	file	Output filestream when not using MPI-IO

fOpenOutputBinaryFile()

```
int fOpenOutputBinaryFile (const char * filename, ofstream & file)
```

Opens a binary file to write simulation outputs. If running in serial or not gathering data in all dimensions, a standard binary filestream is opened: if combining data in all dimensions, MPI-IO is used to open the file and generate the required output file handle (used to write data to the file). Only the root processor for an I/O group will open the file.

Parameters

in	filename	Name of output file to be opened
out	file	Output filestream when not using MPI-IO

fOutputInfo()

```
int fOutputInfo ()
```

Creates a file (lbout.info) to specify the numbers of dimensions, fluids, solutes, temperatures and I/O groups, as well as sizes of integers and single-precision floating-point numbers in bytes. If MPI-IO is not used to create single output files per simulation snapshot, an additional file (lbout.ext) is created to specify the lattice extents for each I/O group. Both files are only written when running DL_MESO_LBE in parallel and are intended for gathering together simulation data written to several files per snapshot after the simulation has finished, using some of the utilities supplied with DL_MESO (e.g. combining solution and grid files in Plot3D, writing linking files for XML-based structured-grid VTK files).

fPieceDataPoints()

```
int fPieceDataPoints (int * start,
                      int * end,
                      int * length)
```

Calculates the number of lattice points (and therefore number of data points) in each dimension of the piece of the simulation box held by the processor, based on the provided extents, and returns the total number.

Parameters

in	start	Starting coordinates of the lattice section (bottom-left-back corner)
in	end	Ending coordinates of the lattice section (front-right-front corner)
out	length	Number of lattice points in section for each direction

fPieceDensities()

```
void fPieceDensities (float * buffer,
                      int * start,
                      int * end,
                      int iprop,
                      bool swapbit)
```

Fills a single-precision floating-point data buffer with densities of a specific fluid for a range of lattice points held by the current processor.

Parameters

out	buffer	Single-precision floating-point array with fluid densities at specified lattice points
in	start	Starting (local) coordinates of the lattice section (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section (front-right-front corner)
in	iprop	Number of fluid for required densities
in	swapbit	Flag for swapping bits of each data point

fPieceDensitiesSwift()

```
void fPieceDensitiesSwift (float * buffer,
                           int * start,
                           int * end,
                           int iprop,
                           bool swapbit)
```

Fills a single-precision floating-point data buffer with densities of a specific fluid for a range of lattice points held by the current processor when using Swift free-energy interactions.

Parameters

out	buffer	Single-precision floating-point array with fluid densities at specified lattice points
in	start	Starting (local) coordinates of the lattice section (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section (front-right-front corner)
in	iprop	Number of fluid for required densities
in	swapbit	Flag for swapping bits of each data point

fPieceGridPointComponent()

```
void fPieceGridPointComponent (float * buffer,
                               int * start,
                               int * end,
                               int comp,
                               bool swapbit)
```

Fills a single-precision floating-point data buffer with a specified component of coordinates (at ‘real-world’ scale) for a range of lattice points held by the current processor.

Parameters

out	buffer	Single-precision floating-point array with specified component of coordinates of specified lattice points
in	start	Starting (local) coordinates of the lattice section (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section (front-right-front corner)
in	comp	Required coordinate components (0 = x, 1 = y, 2 = z)
in	swapbit	Flag for swapping bits of each data point

fPieceGridPoints()

```
void fPieceGridPoints (float * buffer,
                       int * start,
                       int * end,
                       int param,
                       bool swapbit)
```

Fills a single-precision floating-point data buffer with all three components of coordinates (at 'real-world' scale) for a range of lattice points held by the current processor.

Parameters

out	buffer	Single-precision floating-point array with coordinates of specified lattice points
in	start	Starting (local) coordinates of the lattice section (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section (front-right-front corner)
in	param	Dummy parameter (used for calls to data gathering routines)
in	swapbit	Flag for swapping bits of each data point

fPieceMassFractions()

```
void fPieceMassFractions (float * buffer,
                          int * start,
                          int * end,
                          int iprop,
                          bool swapbit)
```

Fills a single-precision floating-point data buffer with mass fractions of a specific fluid for a range of lattice points held by the current processor.

Parameters

out	buffer	Single-precision floating-point array with fluid mass fractions at specified lattice points
in	start	Starting (local) coordinates of the lattice section (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section (front-right-front corner)
in	iprop	Number of fluid for required mass fractions
in	swapbit	Flag for swapping bits of each data point

fPieceMassFractionsSwift()

```
void fPieceMassFractionsSwift (float * buffer,
                               int * start,
                               int * end,
                               int iprop,
                               bool swapbit)
```

Fills a single-precision floating-point data buffer with mass fractions of a specific fluid for a range of lattice points held by the current processor when using Swift free-energy interactions.

Parameters

out	buffer	Single-precision floating-point array with fluid mass fractions at specified lattice points
in	start	Starting (local) coordinates of the lattice section (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section (front-right-front corner)
in	iprop	Number of fluid for required mass fractions
in	swapbit	Flag for swapping bits of each data point

fPiecePhaseField()

```
void fPiecePhaseField (int * buffer,
                       int * start,
                       int * end,
                       int param,
                       bool swapbit)
```

Fills an integer data buffer with boundary condition values (phase fields) for a range of lattice points held by the current processor.

Parameters

out	buffer	Integer array with boundary conditions (phase fields) at specified lattice points
in	start	Starting (local) coordinates of the lattice section (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section (front-right-front corner)
in	param	Dummy parameter (used for calls to data gathering routines)
in	swapbit	Flag for swapping bits of each data point

fPiecePhaseFieldFloat()

```
void fPiecePhaseFieldFloat (float * buffer,
                            int * start,
                            int * end,
                            int param,
                            bool swapbit)
```

Fills a single-precision floating-point data buffer with boundary condition values (phase fields) for a range of lattice points held by the current processor.

Parameters

out	buffer	Single-precision floating-point array with boundary conditions (phase fields) at specified lattice points
in	start	Starting (local) coordinates of the lattice section (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section (front-right-front corner)
in	param	Dummy parameter (used for calls to data gathering routines)
in	swap-bit	Flag for swapping bits of each data point

fPieceRangeGlobal()

```
void fPieceRangeGlobal (int * myStart, int * myEnd)
```

Works out the extent of the subdomain (section of lattice) covered by the current processor in global coordinates, considering any surrounding boundary halos as input parameters for the subroutine. The inner boundary halos between processors in a gathered group can either be switched on or off: these are usually switched off for inner processors.

Parameters

out	myStart	Starting global coordinates of the lattice section (bottom-left-back corner)
out	myEnd	Ending global coordinates of the lattice section (top-right-front corner)

fPieceRangeLocal()

```
void fPieceRangeLocal (int * start,
                       int * end,
                       bool * includeInnerHalos)
```

Works out the extent of the subdomain (section of lattice) covered by the current processor in local coordinates, considering any surrounding boundary halos as input parameters for the subroutine. The inner boundary halos between processors in a gathered group can either be switched on or off: these are usually switched off for inner processors.

Parameters

out	start	Starting local coordinates of the lattice section (bottom-left-back corner)
out	end	Ending local coordinates of the lattice section (top-right-front corner)
in	includeInner-Halos	Switch for each dimension to include boundary halos in processors not at the edges of the simulation box

fPieceSoluteConcentrations()

```
void fPieceSoluteConcentrations (float * buffer,
                                 int * start,
                                 int * end,
                                 int iprop,
                                 bool swapbit)
```

Fills a single-precision floating-point data buffer with concentrations of a specific solute for a range of lattice points held by the current processor.

Parameters

out	buffer	Single-precision floating-point array with solute concentrations at specified lattice points
in	start	Starting (local) coordinates of the lattice section (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section (front-right-front corner)
in	iprop	Number of solute for required concentrations
in	swapbit	Flag for swapping bits of each data point

fPieceTemperatures()

```
void fPieceTemperatures (float * buffer,
                         int * start,
                         int * end,
                         int param,
                         bool swapbit)
```

Fills a single-precision floating-point data buffer with temperatures for a range of lattice points held by the current processor.

Parameters

out	buffer	Single-precision floating-point array with temperatures at specified lattice points
in	start	Starting (local) coordinates of the lattice section (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section (front-right-front corner)
in	param	Dummy parameter (used for calls to data gathering routines)
in	swapbit	Flag for swapping bits of each data point

fPieceVelocities()

```
void fPieceVelocities (float * buffer,
                       int * start,
                       int * end,
                       int param,
                       bool swapbit)
```

Fills a single-precision floating-point data buffer with all three components of fluid velocities for a range of lattice points held by the currentl processor.

Parameters

out	buffer	Single-precision floating-point array with fluid velocities at specified lattice points
in	start	Starting (local) coordinates of the lattice section (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section (front-right-front corner)
in	param	Dummy parameter (used for calls to data gathering routines)
in	swapbit	Flag for swapping bits of each data point

fPieceVelocity()

```
void fPieceVelocity (float * buffer,
                     int * start,
                     int * end,
                     int comp,
                     bool swapbit)
```

Fills a single-precision floating-point data buffer with a specified component of fluid velocities for a range of lattice points held by the current processor.

Parameters

out	buffer	Single-precision floating-point array with specified component of fluid velocities at specified lattice points
in	start	Starting (local) coordinates of the lattice section (bottom-left-back corner)
in	end	Ending (local) coordinates of the lattice section (front-right-front corner)
in	comp	Required velocity components (0 = x, 1 = y, 2 = z)
in	swap-bit	Flag for swapping bits of each data point

fReadRestart()

```
int fReadRestart (const char * filename)
```

Opens a restart file created during a previous DL_MESO_LBE calculation, reads in the positions for each lattice point and the distribution functions and local relaxation frequencies at that point, and assigns the data to the appropriate arrays in the processor that includes that lattice point. Some important simulation properties (lattice scheme, grid size, numbers of fluids, solutes, temperature scalars and phase fields, flag for incompressible fluids) are checked prior to reading in simulation data.

Parameters

in	filename	Name of restart file to be read

fWriteLegacyVTKFloatANSIData()

```
int fWriteLegacyVTKFloatANSIData (float * buffer,
                                  int bufflen,
                                  bool vec,
                                  unsigned long & startpos,
                                  const char * header,
                                  const char * footer,
                                  unsigned long headersize,
                                  unsigned long footersize,
                                  ofstream & file)
```

Writes a single-precision floating-point array of data (gathered among an I/O group) to an ANSI/text Legacy VTK structured grid file, including any required text before and/or after the data stream as a header and/or a footer. If running in serial or writing multiple files per snapshot (i.e. not combining data in all dimensions), the data are written using the previously-opened text filestream for the I/O group by its root processor; if combining data in all dimensions, all root processors in all I/O groups write their data concurrently (at the same time) to the file at a specific position using MPI-IO. The specified position for writing data to the file is advanced by how much data (in total, including headers and footers) is written to the file.

Parameters

in	buffer	Single-precision floating-point array of data to be written to structured-grid Legacy VTK file
in	bufflen	Number of values in data array to write to output file
in	vec	Flag indicating whether or not data is supplied as vectors (if so, writes three values per line instead of one for scalar properties)
in,out	start-pos	Position to start writing data to output file (as number of bytes from beginning)
in	header	Text header to include in output file before writing data array
in	footer	Text footer to include in output file after writing data array
in	header-size	Size of text header in bytes
in	footer-size	Size of text footer in bytes
in	file	Output filestream when not using MPI-IO

fWriteLegacyVTKFloatBinaryData()

```
int fWriteLegacyVTKFloatBinaryData (float * buffer,
                                    int bufflen,
                                    unsigned long & startpos,
                                    const char * header,
                                    const char * footer,
                                    unsigned long headersize,
                                    unsigned long footersize,
                                    ofstream & file)
```

Writes a single-precision floating-point array of data (gathered among an I/O group) to a big endian binary Legacy VTK structured grid file, including any required text before and/or after the data stream as a header and/or a footer. If running in serial or writing multiple files per snapshot (i.e. not combining data in all dimensions), the data are written using the previously-opened binary filestream for the I/O group by its root processor; if combining data in all dimensions, all root processors in all I/O groups write their data concurrently (at the same time) to the file at a specific position using MPI-IO. The specified position for writing data to the file is advanced by how much data (in total, including headers and footers) is written to the file.

Parameters

in	buffer	Single-precision floating-point array of data to be written to structured-grid Legacy VTK file
in	bufflen	Number of values in data array to write to output file
in,out	startpos	Position to start writing data to output file (as number of bytes from beginning)
in	header	Text header to include in output file before writing data array
in	footer	Text footer to include in output file after writing data array
in	header-size	Size of text header in bytes
in	footersize	Size of text footer in bytes
in	file	Output filestream when not using MPI-IO

fWriteLegacyVTKIntegerANSIData()

```
int fWriteLegacyVTKIntegerANSIData (int * buffer,
                                    int bufflen,
                                    bool vec,
                                    unsigned long & startpos,
                                    const char * header,
                                    const char * footer,
                                    unsigned long headersize,
                                    unsigned long footersize,
                                    ofstream & file)
```

Writes an integer array of data (gathered among an I/O group) to an ANSI/text Legacy VTK structured grid file, including any required text before and/or after the data stream as a header and/or a footer. If running in serial or writing multiple files per snapshot (i.e. not combining data in all dimensions), the data are written using the previously-opened text filestream for the I/O group by its root processor; if combining data in all dimensions, all root processors in all I/O groups write their data concurrently (at the same time) to the file at a specific position using MPI-IO. The specified position for writing data to the file is advanced by how much data (in total, including headers and footers) is written to the file.

Parameters

in	buffer	Integer array of data to be written to structured-grid Legacy VTK file
in	bufflen	Number of values in data array to write to output file
in	vec	Flag indicating whether or not data is supplied as vectors (if so, writes three values per line instead of one for scalar properties)
in,out	start-pos	Position to start writing data to output file (as number of bytes from beginning)
in	header	Text header to include in output file before writing data array
in	footer	Text footer to include in output file after writing data array
in	header-size	Size of text header in bytes
in	footer-size	Size of text footer in bytes
in	file	Output filestream when not using MPI-IO

fWriteLegacyVTKIntegerBinaryData()

```
int fWriteLegacyVTKIntegerBinaryData (int * buffer,
                                      int bufflen,
                                      unsigned long & startpos,
                                      const char * header,
                                      const char * footer,
                                      unsigned long headersize,
                                      unsigned long footersize,
                                      ofstream & file)
```

Writes an integer array of data (gathered among an I/O group) to a big endian binary Legacy VTK structured grid file, including any required text before and/or after the data stream as a header and/or a footer. If running in serial or writing multiple files per snapshot (i.e. not combining data in all dimensions), the data are written using the previously-opened binary filestream for the I/O group by its root processor; if combining data in all dimensions, all root processors in all I/O groups write their data concurrently (at the same time) to the file at a specific position using MPI-IO. The specified position for writing data to the file is advanced by how much data (in total, including headers and footers) is written to the file.

Parameters

in	buffer	Integer array of data to be written to structured-grid Legacy VTK file
in	bufflen	Number of values in data array to write to output file
in,out	startpos	Position to start writing data to output file (as number of bytes from beginning)
in	header	Text header to include in output file before writing data array
in	footer	Text footer to include in output file after writing data array
in	headersize	Size of text header in bytes
in	footersize	Size of text footer in bytes
in	file	Output filestream when not using MPI-IO

fWritePlot3DGridFloatANSIData()

```
int fWritePlot3DGridFloatANSIData (float * buffer,
                                   int bufflen,
                                   unsigned long & startpos,
                                   ofstream & file)
```

Writes a single-precision floating-point array of data (gathered among an I/O group) to an ANSI/text Plot3D solution file. If running in serial or writing multiple files per snapshot (i.e. not combining data in all dimensions), the data are written using the previously-opened text filestream for the I/O group by its root processor; if combining data in all dimensions, all root processors in all I/O groups write their data concurrently at the same time) to the file at a specific position using MPI-IO. The specified position for writing data to the file is advanced by how much data (in total) is written to the file.

Parameters

in	buffer	Single-precision floating-point array of data to be written to structured-grid Legacy VTK file
in	bufflen	Number of values in data array to write to output file
in,out	start-pos	Position to start writing data to output file (as number of bytes from beginning)
in	file	Output filestream when not using MPI-IO

fWritePlot3DGridFloatBinaryData()

```
int fWritePlot3DGridFloatBinaryData (float * buffer,
                                     int bufflen,
                                     unsigned long & startpos,
                                     ofstream & file)
```

Writes a single-precision floating-point array of data (gathered among an I/O group) to a binary Plot3D solution file (using the native endianness of the computer running the simulation). If running in serial or writing multiple files per snapshot (i.e. not combining data in all dimensions), the data are written using the previously-opened binary filestream for the I/O group by its root processor; if combining data in all dimensions, all root processors in all I/O groups write their data concurrently at the same time) to the file at a specific position using MPI-IO. The specified position for writing data to the file is advanced by how much data (in total) is written to the file.

Parameters

in	buffer	Single-precision floating-point array of data to be written to structured-grid Legacy VTK file
in	bufflen	Number of values in data array to write to output file
in,out	start-pos	Position to start writing data to output file (as number of bytes from beginning)
in	file	Output filestream when not using MPI-IO

fWritePlot3DGridIntegerANSIData()

```
int fWritePlot3DGridIntegerANSIData (int * buffer,
                                     int bufflen,
                                     unsigned long & startpos,
                                     ofstream & file)
```

Writes an integer array of data (gathered among an I/O group) to an ANSI/text Plot3D solution file. If running in serial or writing multiple files per snapshot (i.e. not combining data in all dimensions), the data are written using the previously-opened text filestream for the I/O group by its root processor; if combining data in all dimensions, all root processors in all I/O groups write their data concurrently at the same time) to the file at a specific position using MPI-IO. The specified position for writing data to the file is advanced by how much data (in total) is written to the file.

Parameters

in	buffer	Integer array of data to be written to structured-grid Legacy VTK file
in	bufflen	Number of values in data array to write to output file
in,out	startpos	Position to start writing data to output file (as number of bytes from beginning)
in	file	Output filestream when not using MPI-IO

fWritePlot3DGridIntegerBinaryData()

```
int fWritePlot3DGridIntegerBinaryData (int * buffer,
                                       int bufflen,
                                       unsigned long & startpos,
                                       ofstream & file)
```

Writes an integer array of data (gathered among an I/O group) to a binary Plot3D solution file (using the native endianness of the computer running the simulation). If running in serial or writing multiple files per snapshot (i.e. not combining data in all dimensions), the data are written using the previously-opened binary filestream for the I/O group by its root processor; if combining data in all dimensions, all root processors in all I/O groups write their data concurrently at the same time) to the file at a specific position using MPI-IO. The specified position for writing data to the file is advanced by how much data (in total) is written to the file.

Parameters

in	buffer	Integer array of data to be written to structured-grid Legacy VTK file
in	bufflen	Number of values in data array to write to output file
in,out	startpos	Position to start writing data to output file (as number of bytes from beginning)
in	file	Output filestream when not using MPI-IO

fWriteRestart()

```
int fWriteRestart (const char * filename = "lbout")
```

Opens a binary file and writes basic simulation properties (lattice scheme, grid size, numbers of fluids, solutes, temperature scalars and phase fields, flag for incompressible fluids), constant fluid densities (if using fully incompressible fluids), one-dimensional grid positions (on global basis), distribution functions and local relaxation frequencies for every lattice point. The parallel version of this routine exploits the I/O groups created for simulation output files, with each group's root processor writing data concurrently (at the same time) to the file using MPI-IO with displacements to avoid overlap (even if the simulation output files do not use MPI-IO). The serial version of this routine directly writes the processor's data to the file. The filename used as a parameter for this subroutine automatically has '.dump' appended to the end.

Parameters

in	filename	Name of restart file to be written (default: lbout.dump)

fWriteVTKFloatANSIData()

```
int fWriteVTKFloatANSIData (float * buffer,
                            int bufflen,
                            unsigned long & startpos,
                            const char * header,
                            const char * footer,
                            unsigned long headersize,
                            unsigned long footersize,
                            ofstream & file)
```

Writes a single-precision floating-point array of data (gathered among an I/O group) to an ANSI/text XML-based VTK structured grid file, including any required XML tags as a header and/or a footer to the data stream. If running in serial or writing multiple files per snapshot (i.e. not combining data in all dimensions), the data are written using the previously-opened text filestream for the I/O group by its root processor; if combining data in all dimensions, all root processors in all I/O groups write their data concurrently (at the same time) to the file at a specific position using MPI-IO. The specified position for writing data to the file is advanced by how much data (in total, including headers and footers) is written to the file.

Parameters

in	buffer	Single-precision floating-point array of data to be written to XML-based structured-grid VTK file
in	bufflen	Number of values in data array to write to output file
in,out	startpos	Position to start writing data to output file (as number of bytes from beginning)
in	header	Text header to include in output file before writing data array
in	footer	Text footer to include in output file after writing data array
in	header-size	Size of text header in bytes
in	footer-size	Size of text footer in bytes
in	file	Output filestream when not using MPI-IO

fWriteVTKFloatBinaryData()

```
int fWriteVTKFloatBinaryData (float * buffer,
                              int bufflen,
                              unsigned long & startpos,
                              const char * header,
                              const char * footer,
                              unsigned long headersize,
                              unsigned long footersize,
                              ofstream & file)
```

Writes a single-precision floating-point array of data (gathered among an I/O group) to a big endian binary XML-based VTK structured grid file, including any required XML tags as a header and/or a footer to the data stream. If running in serial or writing multiple files per snapshot (i.e. not combining data in all dimensions), the data are written using the previously-opened binary filestream for the I/O group by its root processor; if combining data in all dimensions, all root processors in all I/O groups write their data concurrently (at the same time) to the file at a specific position using MPI-IO. The specified position for writing data to the file is advanced by how much data (in total, including headers and footers) is written to the file.

Parameters

in	buffer	Single-precision floating-point array of data to be written to XML-based structured-grid VTK file
in	bufflen	Number of values in data array to write to output file
in,out	startpos	Position to start writing data to output file (as number of bytes from beginning)
in	header	Text header to include in output file before writing data array
in	footer	Text footer to include in output file after writing data array
in	header-size	Size of text header in bytes
in	footer-size	Size of text footer in bytes
in	file	Output filestream when not using MPI-IO

fWriteVTKIntegerANSIData()

```
int fWriteVTKIntegerANSIData (int * buffer,
                              int bufflen,
                              unsigned long & startpos,
                              const char * header,
                              const char * footer,
                              unsigned long headersize,
                              unsigned long footersize,
                              ofstream & file)
```

Writes an integer array of data (gathered among an I/O group) to an ANSI/text XML-based VTK structured grid file, including any required XML tags as a header and/or a footer to the data stream. If running in serial or writing multiple files per snapshot (i.e. not combining data in all dimensions), the data are written using the previously-opened text filestream for the I/O group by its root processor; if combining data in all dimensions, all root processors in all I/O groups write their data concurrently (at the same time) to the file at a specific position using MPI-IO. The specified position for writing data to the file is advanced by how much data (in total, including headers and footers) is written to the file.

Parameters

in	buffer	Integer array of data to be written to XML-based structured-grid VTK file
in	bufflen	Number of values in data array to write to output file
in,out	startpos	Position to start writing data to output file (as number of bytes from beginning)
in	header	Text header to include in output file before writing data array
in	footer	Text footer to include in output file after writing data array
in	headersize	Size of text header in bytes
in	footersize	Size of text footer in bytes
in	file	Output filestream when not using MPI-IO

fWriteVTKIntegerBinaryData()

```
int fWriteVTKIntegerBinaryData (int * buffer,
                                int bufflen,
                                unsigned long & startpos,
                                const char * header,
                                const char * footer,
                                unsigned long headersize,
                                unsigned long footersize,
                                ofstream & file)
```

Writes an integer array of data (gathered among an I/O group) to a big endian binary XML-based VTK structured grid file, including any required XML tags as a header and/or a footer to the data stream. If running in serial or writing multiple files per snapshot (i.e. not combining data in all dimensions), the data are written using the previously-opened binary filestream for the I/O group by its root processor; if combining data in all dimensions, all root processors in all I/O groups write their data concurrently (at the same time) to the file at a specific position using MPI-IO. The specified position for writing data to the file is advanced by how much data (in total, including headers and footers) is written to the file.

Parameters

in	buffer	Integer array of data to be written to XML-based structured-grid VTK file
in	bufflen	Number of values in data array to write to output file
in,out	startpos	Position to start writing data to output file (as number of bytes from beginning)
in	header	Text header to include in output file before writing data array
in	footer	Text footer to include in output file after writing data array
in	headersize	Size of text header in bytes
in	footersize	Size of text footer in bytes
in	file	Output filestream when not using MPI-IO

5.21 lbpIOVTK.cpp

Module with routines to write calculation output files in structured grid XML-based VTK format. (Header file available as lbpIOVTK.hpp.)

Subroutines to write big endian binary and ANSI/text XML-based VTK files in structured grid format for LBE outputs, including fluid velocities. densities, mass fractions, solute concentrations, temperatures and boundary conditions (phase fields), as simulation snapshots. The binary form of these files uses XML tags to identify the datasets and indicates where each set starts in a raw stream of binary data appended near the end of the file. The ANSI/text form of these files places the values of properties inside their XML tags.

5.21.1 Functions

- `int` *fOutputVTK()*

  Writes binary simulation output file with all system data in XML-based VTK structured grid format.
- `int` *fsOutputVTK()*

  Writes ANSI text simulation output file with all system data in XML-based VTK structured grid format.
- `int` *fOutputVTKP()*

  Writes binary simulation output file with density of specified fluid in XML-based VTK structured grid format.
- `int` *fsOutputVTKP()*

  Writes ANSI text simulation output file with density of specified fluid in XML-based VTK structured grid format.
- `int` *fOutputVTKCA()*

  Writes binary simulation output file with mass fraction of specified fluid in XML-based VTK structured grid format.
- `int` *fsOutputVTKCA()*

  Writes ANSI text simulation output file with mass fraction of specified fluid in XML-based VTK structured grid format.
- `int` *fOutputVTKCB()*

  Writes binary simulation output file with concentration of specified solute in XML-based VTK structured grid format.
- `int` *fsOutputVTKCB()*

  Writes ANSI text simulation output file with concentration of specified solute in XML-based VTK structured grid format.
- `int` *fOutputVTKT()*

  Writes binary simulation output file with temperatures in XML-based VTK structured grid format.
- `int` *fsOutputVTKT()*

  Writes ANSI text simulation output file with temperatures in XML-based VTK structured grid format.

5.21.2 Function Documentation

fOutputVTK()

```
int fOutputVTK (const char * filename = "lbout")
```

Writes a structured grid XML-based VTK output file (*.vts) in big endian binary that includes all possible properties at each lattice point: fluid densities and mass fractions, solute concentrations, temperatures, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)

fOutputVTKCA()

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

Writes a structured grid XML-based VTK output file (*.vts) in big endian binary that includes the mass fraction of a specified fluid, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of fluid for required mass fraction (default: 0)

fOutputVTKCB()

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

Writes a structured grid XML-based VTK output file (*.vts) in big endian binary that includes the concentration of a specified solute, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of solute for required concentration (default: 0)

fOutputVTKP()

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

Writes a structured grid XML-based VTK output file (*.vts) in big endian binary that includes the density of a specified fluid, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of fluid for required density (default: 0)

fOutputVTKT()

```
int fOutputVTKT (const char * filename = "lbout")
```

Writes a structured grid XML-based VTK output file (*.vts) in big endian binary that includes the temperatures, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)

fsOutputVTK()

```
int fsOutputVTK (const char * filename = "lbout")
```

Writes a structured grid XML-based VTK output file (*.vts) in ANSI/text that includes all possible properties at each lattice point: fluid densities and mass fractions, solute concentrations, temperatures, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)

fsOutputVTKCA()

```
int fsOutputVTKCA (const char * filename = "lbout", int iprop = 0)
```

Writes a structured grid XML-based VTK output file (*.vts) in ANSI/text that includes the mass fraction of a specified fluid, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of fluid for required mass fraction (default: 0)

fsOutputVTKCB()

```
int fsOutputVTKCB (const char * filename = "lbout", int iprop = 0)
```

Writes a structured grid XML-based VTK output file (*.vts) in ANSI/text that includes the concentration of a specified solute, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of solute for required concentration (default: 0)

fsOutputVTKP()

```
int fsOutputVTKP (const char * filename = "lbout", int iprop = 0)
```

Writes a structured grid XML-based VTK output file (*.vts) in ANSI/text that includes the density of a specified fluid, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of fluid for required density (default: 0)

fsOutputVTKT()

```
int fsOutputVTKT (const char * filename = "lbout")
```

Writes a structured grid XML-based VTK output file (*.vts) in ANSI/text that includes the temperatures, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)

5.22 lbpIOLegacyVTK.cpp

Module with routines to write calculation output files in structured grid Legacy VTK format. (Header file available as lbpIOLegacyVTK.cpp.)

Subroutines to write big endian binary and ANSI/text Legacy VTK files in structured grid format for LBE outputs, including fluid velocities. densities, mass fractions, solute concentrations, temperatures and boundary conditions (phase fields), as simulation snapshots.

5.22.1 Functions

- `int` *fOutputLegacyVTK()*

  Writes binary simulation output file with all system data in Legacy VTK structured grid format.

- `int` *fsOutputLegacyVTK()*

  Writes ANSI text simulation output file with all system data in Legacy VTK structured grid format.

- `int` *fOutputLegacyVTKP()*

  Writes binary simulation output file with density of specified fluid in Legacy VTK structured grid format.

- `int` *fsOutputLegacyVTKP()*

  Writes ANSI text simulation output file with density of specified fluid in Legacy VTK structured grid format.
- `int` *fOutputLegacyVTKCA()*

  Writes binary simulation output file with mass fraction of specified fluid in Legacy VTK structured grid format.
- `int` *fsOutputLegacyVTKCA()*

  Writes ANSI text simulation output file with mass fraction of specified fluid in Legacy VTK structured grid format.
- `int` *fOutputLegacyVTKCB()*

  Writes binary simulation output file with concentration of specified solute in Legacy VTK structured grid format.
- `int` *fsOutputLegacyVTKCB()*

  Writes ANSI text simulation output file with concentration of specified solute in Legacy VTK structured grid format.
- `int` *fOutputLegacyVTKT()*

  Writes binary simulation output file with temperatures in Legacy VTK structured grid format.
- `int` *fsOutputLegacyVTKT3D()*

  Writes ANSI text simulation output file with temperatures in Legacy VTK structured grid format.

5.22.2 Function Documentation

fOutputLegacyVTK()

```
int fOutputLegacyVTK (const char * filename = "lbout")
```

Writes a structured grid Legacy VTK output file (*.vtk) in big endian binary that includes all possible properties at each lattice point: fluid densities and mass fractions, solute concentrations, temperatures, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)

fOutputLegacyVTKCA()

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

Writes a structured grid Legacy VTK output file (*.vtk) in big endian binary that includes the mass fraction of a specified fluid, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of fluid for required mass fraction (default: 0)

fOutputLegacyVTKCB()

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

Writes a structured grid Legacy VTK output file (*.vtk) in big endian binary that includes the concentration of a specified solute, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of solute for required concentration (default: 0)

fOutputLegacyVTKP()

```
int fOutputLegacyVTKP (const char * filename = "lbout", int iprop = 0)
```

Writes a structured grid Legacy VTK output file (*.vtk) in big endian binary that includes the density of a specified fluid, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of fluid for required density (default: 0)

fOutputLegacyVTKT()

```
int fOutputLegacyVTKT (const char * filename = "lbout")
```

Writes a structured grid Legacy VTK output file (*.vtk) in big endian binary that includes the temperatures, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)

fsOutputLegacyVTK()

```
int fsOutputLegacyVTK (const char * filename = "lbout")
```

Writes a structured grid Legacy VTK output file (*.vtk) in ANSI/text that includes all possible properties at each lattice point: fluid densities and mass fractions, solute concentrations, temperatures, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)

fsOutputLegacyVTKCA()

```
int fsOutputLegacyVTKCA (const char * filename = "lbout", int iprop = 0)
```

Writes a structured grid Legacy VTK output file (*.vtk) in ANSI/text that includes the mass fraction of a specified fluid, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of fluid for required mass fraction (default: 0)

fsOutputLegacyVTKCB()

```
int fsOutputLegacyVTKCB (const char * filename = "lbout", int iprop = 0)
```

Writes a structured grid Legacy VTK output file (*.vtk) in ANSI/text that includes the concentration of a specified solute, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of solute for required concentration (default: 0)

fsOutputLegacyVTKP()

```
int fsOutputLegacyVTKP (const char * filename = "lbout", int iprop = 0)
```

Writes a structured grid Legacy VTK output file (*.vtk) in ANSI/text that includes the density of a specified fluid, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of fluid for required density (default: 0)

fsOutputLegacyVTKT3D()

```
int fsOutputLegacyVTKT3D (const char * filename = "lbout")
```

Writes a structured grid Legacy VTK output file (*.vtk) in ANSI/text that includes the temperatures, velocities, boundary conditions (phase fields) and grid points in 'real-life' units. This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)

5.23 lbpIOPlot3D.cpp

Module with routines to write calculation output files in Plot3D format. (Header file available as lbpIO-Plot3D.hpp.)

Subroutines to write binary and ANSI/text grid and solution files in Plot3D format for LBE outputs, including fluid velocities. densities, mass fractions, solute concentrations, temperatures and boundary conditions (phase fields), as simulation snapshots.

5.23.1 Functions

- `int` *fOutputQ()*

  Writes binary simulation output files with all system data in Plot3D format.

- `int` *fsOutputQ()*

  Writes ANSI text simulation output files with all system data in Plot3D format.

- `int` *fOutputGrid()*

  Writes binary simulation grid files with all system data in Plot3D format.

- `int` *fsOutputGrid()*

  Writes ANSI text simulation grid files with all system data in Plot3D format.

- `int` *fOutputQP()*

  Writes binary simulation output file with density of specified fluid in Plot3D format.

- `int` *fsOutputQP()*

  Writes ANSI text simulation output file with density of specified fluid in Plot3D format.

- `int` *fOutputQCA()*

  Writes binary simulation output file with mass fraction of specified fluid in Plot3D format.

- `int` *fsOutputQCA()*

  Writes ANSI text simulation output file with mass fraction of specified fluid in Plot3D format.

- `int` *fOutputQCB()*

  Writes binary simulation output file with concentration of specified solute in Plot3D format.

- `int` *fsOutputQCB()*

  Writes ANSI text simulation output file with concentration of specified solute in Plot3D format.

- `int` *fOutputQT()*

  Writes binary simulation output file with temperature in Plot3D format.

- `int` *fsOutputQT()*

  Writes ANSI text simulation output file with temperature in Plot3D format.

5.23.2 Function Documentation

fOutputGrid()

```
int fOutputGrid (const char * filename = "lbout")
```

Writes a Plot3D grid file (*.xyz or *.xy) in binary that includes lattice points in 'real-life' units, with each component in a data block. These files can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and, if more than one file is written, are numbered according to the lattice section for combining together after the end of the simulation. This subroutine only needs to be called once per simulation as the lattice grid does not change.

Parameters

in	filename	Beginning of filename for output grid file(s) (default: lbout)

fOutputQ()

```
int fOutputQ (const char * filename = "lbout")
```

Writes Plot3D solution files (*.q) in binary that include all possible properties at each lattice point: fluid densities and mass fractions, solute concentrations, temperatures, velocities and boundary conditions (phase fields). These files can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and are numbered according to the number of frames, (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation, and the property (fluid density, mass fraction, solute concentration, temperature) included in the file.

Parameters

in	filename	Beginning of filename for output files (default: lbout)

fOutputQCA()

```
int fOutputQCA (const char * filename = "lbout", int iprop = 0)
```

Writes a Plot3D solution file (*.q) in binary that includes the mass fractions of a specified fluid, velocities and boundary conditions (phase fields). This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of fluid for required density (default: 0)

fOutputQCB()

```
int fOutputQCB (const char * filename = "lbout", int iprop = 0)
```

Writes a Plot3D solution file (*.q) in binary that includes the concentrations of a specified solute, velocities and boundary conditions (phase fields). This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of fluid for required density (default: 0)

fOutputQP()

```
int fOutputQP (const char * filename = "lbout", int iprop = 0)
```

Writes a Plot3D solution file (*.q) in binary that includes the densities of a specified fluid, velocities and boundary conditions (phase fields). This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of fluid for required density (default: 0)

fOutputQT()

```
int fOutputQT (const char * filename = "lbout")
```

Writes a Plot3D solution file (*.q) in binary that includes the temperatures, velocities and boundary conditions (phase fields). This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)

fsOutputGrid()

```
int fsOutputGrid (const char * filename = "lbout")
```

Writes a Plot3D grid file (*.xyz or *.xy) in ANSI/text that includes lattice points in ‘real-life’ units, with each component in a data block. These files can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and, if more than one file is written, are numbered according to the lattice section for combining together after the end of the simulation. This subroutine only needs to be called once per simulation as the lattice grid does not change.

Parameters

in	filename	Beginning of filename for output grid file(s) (default: lbout)

fsOutputQ()

```
int fsOutputQ (const char * filename = "lbout")
```

Writes Plot3D solution files (*.q) in ANSI/text that include all possible properties at each lattice point: fluid densities and mass fractions, solute concentrations, temperatures, velocities and boundary conditions (phase fields). These files can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and are numbered according to the number of frames, (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation, and the property (fluid density, mass fraction, solute concentration, temperature) included in the file.

Parameters

in	filename	Beginning of filename for output solution files (default: lbout)

fsOutputQCA()

```
int fsOutputQCA (const char * filename = "lbout", int iprop = 0)
```

Writes a Plot3D solution file (*.q) in ANSI/text that includes the mass fractions of a specified fluid, velocities and boundary conditions (phase fields). This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of fluid for required density (default: 0)

fsOutputQCB()

```
int fsOutputQCB (const char * filename = "lbout", int iprop = 0)
```

Writes a Plot3D solution file (*.q) in ANSI/text that includes the concentrations of a specified solute, velocities and boundary conditions (phase fields). This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of fluid for required density (default: 0)

fsOutputQP()

```
int fsOutputQP (const char * filename = "lbout", int iprop = 0)
```

Writes a Plot3D solution file (*.q) in ANSI/text that includes the densities of a specified fluid, velocities and boundary conditions (phase fields). This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)
in	iprop	Number of fluid for required density (default: 0)

fsOutputQT()

```
int fsOutputQT (const char * filename = "lbout")
```

Writes a Plot3D solution file (*.q) in ANSI/text that includes the temperatures, velocities and boundary conditions (phase fields). This file can either represent the entire lattice (if running in serial or using MPI-IO in parallel) or a section of the lattice, and is numbered according to the number of frames and (if more than one file is written per snapshot) the lattice section for combining together after the end of the simulation.

Parameters

in	filename	Beginning of filename for output file (default: lbout)

5.24 lbpBOUND.cpp

Module for applying boundary conditions. (Header file available as lbpBOUND.hpp.)

5.24.1 Functions

- `long` *fNextStep()*

  Find position at next lattice site along given lattice link.

- `int` *fMoveNonzeroAway()*

  Moves non-zero distribution functions along all lattice links.

- `int` *fBounceBackF()*

  Performs on-grid bounce-back on fluid distribution functions.

- `int` *fBounceBackC()*

  Performs on-grid bounce-back on solute distribution functions.

- `int` *fBounceBackT()*

  Performs on-grid bounce-back on temperature field distribution functions.

- `int` *fMidBounceBackF()*

  Performs mid-grid bounce-back on fluid distribution functions.

- `int` *fMidBounceBackC()*

  Performs mid-grid bounce-back on solute distribution functions.

- `int` *fMidBounceBackT()*

  Performs mid-grid bounce-back on temperature field distribution functions.

- `int` *fSiteBlankF()*

  Zeros distribution functions for fluids at given lattice site.

- `int` *fSiteBlankC()*

  Zeros distribution functions for solutes at given lattice site.

- `int` *fSiteBlankT()*

  Zeros distribution functions for temperature field at given lattice site.

- `int` *fD2CCFracSite()*

  Calculates mass fractions of fluids in concave corner of two-dimensional system.

- `double` *fD2CCSwiftPhi()*

  Calculates fluid concentration in concave corner of two-dimensional system with Swift free-energy interactions.

- `int` *fD3CECCFracSite()*

  Calculates mass fractions of fluids in concave edge or corner of three-dimensional system.

- `double` *fD3CECCSwiftPhi()*

  Calculates fluid concentration in concave edge or corner of three-dimensional system with Swift free-energy interactions.

- `int` *fD2Q9CEFracSite()*

  Calculates mass fractions of fluids at concave edge of two-dimensional system with D2Q9 lattice.

- `double` *fD2Q9CESwiftPhi()*

  Calculates fluid concentration at concave edge of two-dimensional system with Swift free-energy interactions and D2Q9 lattice.

- `int` *fD2Q9BoundaryForceVelocity()*

  Calculates forces required at a constant velocity boundary point for a D2Q9 lattice.

- `int` *fD2Q9BoundaryForceDensity()*

  Calculates forces required at a constant density boundary point for a D2Q9 lattice.

- `int` *fD2Q9OF1()*

  Applies first-order outflow boundary condition for a D2Q9 lattice.

- `int` *fD2Q9OF2()*

  Applies second-order outflow boundary condition for a D2Q9 lattice.

- `int` *fD3Q15PSFracSite()*

  Calculates mass fractions of fluids at planar surface of three-dimensional system with D3Q15 lattice.

- `double` *fD3Q15PSSwiftPhi()*

  Calculates fluid concentration at planar surface of three-dimensional system with Swift free-energy interactions and D3Q15 lattice.

- `int` *fD3Q15BoundaryForceVelocity()*

  Calculates forces required at a constant velocity boundary point for a D3Q15 lattice.

- `int` *fD3Q15BoundaryForceDensity()*

  Calculates forces required at a constant density boundary point for a D3Q15 lattice.

- `int` *fD3Q15OF1()*

  Applies first-order outflow boundary condition for a D3Q15 lattice.

- `int` *fD3Q15OF2()*

  Applies second-order outflow boundary condition for a D3Q15 lattice.

- `int` *fD3Q19PSFracSite()*

  Calculates mass fractions of fluids at planar surface of three-dimensional system with D3Q19 lattice.

- `double` *fD3Q19PSSwiftPhi()*

  Calculates fluid concentration at planar surface of three-dimensional system with Swift free-energy interactions and D3Q19 lattice.

- `int` *fD3Q19BoundaryForceVelocity()*

  Calculates forces required at a constant velocity boundary point for a D3Q19 lattice.

- `int` *fD3Q19BoundaryForceDensity()*

  Calculates forces required at a constant density boundary point for a D3Q19 lattice.
- `int` *fD3Q19OF1()*

  Applies first-order outflow boundary condition for a D3Q19 lattice.
- `int` *fD3Q19OF2()*

  Applies second-order outflow boundary condition for a D3Q19 lattice.
- `int` *fD3Q27PSFracSite()*

  Calculates mass fractions of fluids at planar surface of three-dimensional system with D3Q27 lattice.
- `int` *fD3Q27BoundaryForceVelocity()*

  Calculates forces required at a constant velocity boundary point for a D3Q27 lattice.
- `int` *fD3Q27BoundaryForceDensity()*

  Calculates forces required at a constant density boundary point for a D3Q27 lattice.
- `int` *fD3Q27OF1()*

  Applies first-order outflow boundary condition for a D3Q27 lattice.
- `int` *fD3Q27OF2()*

  Applies second-order outflow boundary condition for a D3Q27 lattice.
- `int` *fFixedSpeedFluid()*

  Calculates distribution functions for a fixed-speed boundary grid point.
- `int` *fFixedDensityFluid()*

  Calculates distribution functions for a fixed fluid density boundary grid point.
- `int` *fFixedSoluteConcen()*

  Calculates distribution functions for a fixed solute concentration boundary grid point.
- `int` *fFixedTemperature()*

  Calculates distribution functions for a fixed temperature boundary grid point.
- `int` *fOutFlow()*

  Calculates distribution functions for an outflow boundary grid point.
- `int` *fPostCollBoundary()*

  Calculates distribution functions at boundary lattice points after collsions (before propagation).
- `int` *fPostPropBoundary()*

  Calculates distribution functions at boundary lattice points after propagation.
- `int` *fNeighbourBoundary()*

  Determines which neighbouring lattice points are boundary points, assigns flags for gradient calculations and calculates boundary normals.
- `int` *fsPeriodic()*

  Applies periodic boundary conditions on distribution functions for serial simulations with non-zero boundary halos.
- `int` *fsBoundPeriodic()*

  Applies periodic boundary conditions on boundary information for serial simulations with non-zero boundary halos.

- `int` *fsForcePeriodic()*

  Applies periodic boundary conditions on interfacial forces for serial simulations with non-zero boundary halos.

- `int` *fsIndexPeriodic()*

  Applies periodic boundary conditions on phase indices for serial simulations with non-zero boundary halos.

- `int` *fsPeriodic2D()*

  Applies periodic boundary conditions on distribution functions for two-dimensional serial simulations with non-zero boundary halos.

- `int` *fsPeriodic3D()*

  Applies periodic boundary conditions on distribution functions for three-dimensional serial simulations with non-zero boundary halos.

- `int` *fsBoundPeriodic2D()*

  Applies periodic boundary conditions on boundary information for two-dimensional serial simulations with non-zero boundary halos.

- `int` *fsBoundPeriodic3D()*

  Applies periodic boundary conditions on boundary information for three-dimensional serial simulations with non-zero boundary halos.

- `int` *fsForcePeriodic2D()*

  Applies periodic boundary conditions on interfacial forces for two-dimensional serial simulations with non-zero boundary halos.

- `int` *fsForcePeriodic3D()*

  Applies periodic boundary conditions on interfacial forces for three-dimensional serial simulations with non-zero boundary halos.

- `int` *fsIndexPeriodic2D()*

  Applies periodic boundary conditions on phase indices for two-dimensional serial simulations with non-zero boundary halos.

- `int` *fsIndexPeriodic3D()*

  Applies periodic boundary conditions on phase indices for three-dimensional serial simulations with non-zero boundary halos.

5.24.2 Function Documentation

fBounceBackC()

```
int fBounceBackC (long tpos)
```

At the current lattice site, apply an on-grid bounce-back boundary condition on solute distribution functions, i.e. swap distribution functions between conjugate lattice links.

Parameters

in	tpos	Position of current lattice site (in one-dimensional form)

fBounceBackF()

```
int fBounceBackF (long tpos)
```

At the current lattice site, apply an on-grid bounce-back boundary condition on fluid distribution functions, i.e. swap distribution functions between conjugate lattice links.

Parameters

in	tpos	Position of current lattice site (in one-dimensional form)

fBounceBackT()

```
int fBounceBackT (long tpos)
```

At the current lattice site, apply an on-grid bounce-back boundary condition on temperature distribution functions, i.e. swap distribution functions between conjugate lattice links.

Parameters

in	tpos	Position of current lattice site (in one-dimensional form)

fD2CCFracSite()

```
int fD2CCFracSite (double * frac, long tpos)
```

Determines mass fractions of all fluids at a concave corner of a two-dimensional system from the fluid densities of a neighbouring lattice site. This can be applied to any corner if the neighbouring lattice site is specified (normally one site away diagonally from the corner).

Parameters

out	frac	Mass fractions of all fluid at corner lattice site
in	tpos	Position of neighbouring lattice site (in one-dimensional form)

fD2CCSwiftPhi()

```
double fD2CCSwiftPhi (long tpos,
                      double dx,
                      double dy,
                      double dpdx,
                      double dpdy)
```

Determines the fluid concentration of two fluids at a concave corner of a two-dimensional system using the value of a neighbouring lattice site when Swift free-energy interactions are in use and its concentration gradients. This is specified for a bottom-left concave corner site (CCTRF), but can be used for any corner by specifying positive or negative values for changes in position and concentration gradients.

Parameters

in	tpos	Position of neighbouring lattice site (in one-dimensional form)
in	dx	Change in x-position between corner site and neighbouring site
in	dy	Change in y-position between corner site and neighbouring site
in	dpdx	Concentration gradient in x-direction
in	dpdy	Concentration gradient in y-direction

fD2Q9BoundaryForceDensity()

```
int fD2Q9BoundaryForceDensity (double * force,
                               long tpos,
                               double * p0,
                               int prop)
```

Based on the type of boundary (concave edge or corner) and its direction, calculates the forces (interaction, buoyancy-driven thermal and constant/oscillating) acting on a given boundary grid point for application of a constant density boundary condition for a two-dimensional D2Q9 lattice. If the simulation is equilibrating, the constant/oscilliating body forces are not applied.

Parameters

out	force	Overall forces acting on fluids at boundary point
in	tpos	Position of boundary lattice site (in one-dimensional form)
in	p0	Fixed fluid densities at boundary site
in	prop	Boundary condition code indicating type and direction

fD2Q9BoundaryForceVelocity()

```
int fD2Q9BoundaryForceVelocity (double * force,
                                long tpos,
                                long tpos1,
                                double dx,
                                double dy,
                                double * uwall,
                                int prop)
```

Based on the type of boundary (concave edge or corner) and its direction, calculates the forces (interaction, buoyancy-driven thermal and constant/oscillating) acting on a given boundary grid point for application of a constant velocity boundary condition for a two-dimensional D2Q9 lattice. If the simulation is equilibrating, the constant/oscilliating body forces are not applied.

Parameters

out	force	Overall forces acting on fluids at boundary point
in	tpos	Position of boundary lattice site (in one-dimensional form)
in	tpos1	Position of neighbouring lattice site (in one-dimensional form)
in	dx	Change in x-position between boundary site and neighbouring site for Swift free-energy interactions
in	dy	Change in y-position between boundary site and neighbouring site for Swift free-energy interactions
in	uwall	Fixed velocity at boundary site
in	prop	Boundary condition code indicating type and direction

fD2Q9CEFracSite()

```
int fD2Q9CEFracSite (double * frac,
                     double vy,
                     double * f0, double * f1, double * f2,
                     double * f3, double * f4, double * f5,
                     double * f6, double * f7, double * f8)
```

Determines mass fractions of all fluids at a concave edge of a two-dimensional system using known distribution functions at the lattice site for a D2Q9 lattice. The fluid densities are calculated from mass and momentum conservation in the orthogonal direction to the boundary, e.g.

$$\rho\left(1-u_y\right)=f_0+f_2+f_6+2\left(f_3+f_4+f_5\right)$$

The expression in this subroutine is for bottom concave edges (CETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for a (different) velocity component.

Parameters

out	frac	Mass fractions of all fluid at edge lattice site
in	vy	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site

fD2Q9CESwiftPhi()

```
double fD2Q9CESwiftPhi (double vy,
                        double g0, double g1, double g2,
                        double g3, double g4, double g5,
                        double g6, double g7, double g8)
```

Determines concentration of two fluids at a concave edge of a two-dimensional system using known concentration distribution functions at the lattice site for a D2Q9 lattice. The fluid concentrations are calculated from concentration and momentum conservation in the orthogonal direction to the boundary, e.g.

$$\phi\left(1-u_y\right)=g_0+g_2+g_6+2\left(g_3+g_4+g_5\right)$$

The expression in this subroutine is for bottom concave edges (CETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for a (different) velocity component.

Parameters

in	vy	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	g0	Concentration distribution function for link 0 at edge lattice site
in	g1	Concentration distribution function for link 1 at edge lattice site
in	g2	Concentration distribution function for link 2 at edge lattice site
in	g3	Concentration distribution function for link 3 at edge lattice site
in	g4	Concentration distribution function for link 4 at edge lattice site
in	g5	Concentration distribution function for link 5 at edge lattice site
in	g6	Concentration distribution function for link 6 at edge lattice site
in	g7	Concentration distribution function for link 7 at edge lattice site
in	g8	Concentration distribution function for link 8 at edge lattice site

fD2Q9OF1()

```
int fD2Q9OF1 (long tpos, int prop)
```

Based on the direction of the boundary (only edges), applies a first-order outflow (zero-gradient) condition to a boundary lattice point for a two-dimensional D2Q9 lattice. This subroutine applies the following to lattice links pointing back into the lattice:

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

using distribution function values at the nearest neighbouring lattice point orthogonal to the boundary.

Parameters

tpos	Position of boundary lattice site (in one-dimensional form)
prop	Boundary condition code indicating type and direction

fD2Q9OF2()

```
int fD2Q9OF2 (long tpos, int prop)
```

Based on the direction of the boundary (only edges), applies a second-order outflow (zero-gradient) condition to a boundary lattice point for a two-dimensional D2Q9 lattice. This subroutine applies the following to lattice links pointing back into the lattice:

$$f_i\left(\vec{x}_w, t^+\right) = 2f_i\left(\vec{x}_w + \Delta x, t^+\right) - f_i\left(\vec{x}_w + 2\Delta x, t^+\right)$$

using distribution function values at the nearest and next-nearest neighbouring lattice points orthogonal to the boundary.

Parameters

tpos	Position of boundary lattice site (in one-dimensional form)
prop	Boundary condition code indicating type and direction

fD3CECCFracSite()

```
int fD3CECCFracSite (double * frac, long tpos)
```

Determines mass fractions of all fluids at a concave edge or corner of a three-dimensional system from the fluid densities of a neighbouring lattice site. This can be applied to any edge or corner if the neighbouring lattice site is specified (normally one site away diagonally from the edge or corner).

Parameters

out	frac	Mass fractions of all fluid at corner lattice site
in	tpos	Position of neighbouring lattice site (in one-dimensional form)

fD3CECCSwiftPhi()

```
double fD3CECCSwiftPhi (long tpos,
                        double dx,
                        double dy,
                        double dz,
                        double dpdx,
                        double dpdy,
                        double dpdz)
```

Determines the fluid concentration of two fluids at a concave edge or corner of a three-dimensional system using the value of a neighbouring lattice site when Swift free-energy interactions are in use and its concentration gradients. This is specified for a bottom-left-back concave corner site (CCTRF), but can be used for any edge or corner by specifying zero, positive or negative values for changes in position and concentration gradients.

Parameters

in	tpos	Position of neighbouring lattice site (in one-dimensional form)
in	dx	Change in x-position between corner site and neighbouring site
in	dy	Change in y-position between corner site and neighbouring site
in	dz	Change in z-position between corner site and neighbouring site
in	dpdx	Concentration gradient in x-direction
in	dpdy	Concentration gradient in y-direction
in	dpdz	Concentration gradient in z-direction

fD3Q15BoundaryForceDensity()

```
int fD3Q15BoundaryForceDensity (double * force,
                                long tpos,
                                double * p0,
                                int prop)
```

Based on the type of boundary (planar surface, concave edge or corner) and its direction, calculates the forces interaction, buoyancy-driven thermal and constant/oscillating) acting on a given boundary grid point for application of a constant density boundary condition for a three-dimensional D3Q15 lattice. If the simulation is equilibrating, the constant/oscilliating body forces are not applied.

Parameters

out	force	Overall forces acting on fluids at boundary point
in	tpos	Position of boundary lattice site (in one-dimensional form)
in	p0	Fixed fluid densities at boundary site
in	prop	Boundary condition code indicating type and direction

fD3Q15BoundaryForceVelocity()

```
int fD3Q15BoundaryForceVelocity (double * force,
                                 long tpos,
                                 long tpos1,
                                 double dx,
                                 double dy,
                                 double dz,
                                 double * uwall,
                                 int prop)
```

Based on the type of boundary (planar surface, concave edge or corner) and its direction, calculates the forces (interaction, buoyancy-driven thermal and constant/oscillating) acting on a given boundary grid point for application of a constant velocity boundary condition for a three-dimensional D3Q15 lattice. If the simulation is equilibrating, the constant/oscilliating body forces are not applied.

Parameters

out	force	Overall forces acting on fluids at boundary point
in	tpos	Position of boundary lattice site (in one-dimensional form)
in	tpos1	Position of neighbouring lattice site (in one-dimensional form)
in	dx	Change in x-position between boundary site and neighbouring site for Swift free-energy interactions
in	dy	Change in y-position between boundary site and neighbouring site for Swift free-energy interactions
in	dz	Change in z-position between boundary site and neighbouring site for Swift free-energy interactions
in	uwall	Fixed velocity at boundary site
in	prop	Boundary condition code indicating type and direction

fD3Q15OF1()

```
int fD3Q15OF1 (long tpos, int prop)
```

Based on the direction of the boundary (only edges), applies a first-order outflow (zero-gradient) condition to a boundary lattice point for a three-dimensional D3Q15 lattice. This subroutine applies the following to lattice links pointing back into the lattice:

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

using distribution function values at the nearest neighbouring lattice point orthogonal to the boundary.

Parameters

tpos	Position of boundary lattice site (in one-dimensional form)
prop	Boundary condition code indicating type and direction

fD3Q15OF2()

```
int fD3Q15OF2 (long tpos, int prop)
```

Based on the direction of the boundary (only edges), applies a second-order outflow (zero-gradient) condition to a boundary lattice point for a three-dimensional D3Q15 lattice. This subroutine applies the following to lattice links pointing back into the lattice:

$$f_i\left(\vec{x}_w, t^+\right) = 2f_i\left(\vec{x}_w + \Delta x, t^+\right) - f_i\left(\vec{x}_w + 2\Delta x, t^+\right)$$

using distribution function values at the nearest and next-nearest neighbouring lattice points orthogonal to the boundary.

Parameters

tpos	Position of boundary lattice site (in one-dimensional form)
prop	Boundary condition code indicating type and direction

fD3Q15PSFracSite()

```
int fD3Q15PSFracSite (double * frac,
                      double vy,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14)
```

Determines mass fractions of all fluids at a planar surface of a three-dimensional system using known distribution functions at the lattice site for a D3Q15 lattice. The fluid densities are calculated from mass and momentum conservation in the orthogonal direction to the boundary, e.g.

$$\rho\left(1-u_y\right)=f_0+f_1+f_3+f_8+f_{10}+2\left(f_2+f_4+f_5+f_{13}+f_{14}\right)$$

The expression in this subroutine is for bottom planar surfaces (PST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for a (different) velocity component.

Parameters

out	frac	Mass fractions of all fluid at edge lattice site
in	vy	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site

fD3Q15PSSwiftPhi()

```
double fD3Q15PSSwiftPhi (double vy,
                         double g0, double g1, double g2,
                         double g3, double g4, double g5,
                         double g6, double g7, double g8,
                         double g9, double g10, double g11,
                         double g12, double g13, double g14)
```

Determines concentration of two fluids at a planar surfae of a three-dimensional system using known concentration distribution functions at the lattice site for a D3Q15 lattice. The fluid concentrations are calculated from concentration and momentum conservation in the orthogonal direction to the boundary, e.g.

$$\phi\left(1-u_y\right)=g_0+g_1+g_3+g_8+g_{10}+2\left(g_2+g_4+g_5+g_{13}+g_{14}\right)$$

The expression in this subroutine is for bottom planar surfaces (PST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for a (different) velocity component.

Parameters

in	vy	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	g0	Concentration distribution function for link 0 at edge lattice site
in	g1	Concentration distribution function for link 1 at edge lattice site
in	g2	Concentration distribution function for link 2 at edge lattice site
in	g3	Concentration distribution function for link 3 at edge lattice site
in	g4	Concentration distribution function for link 4 at edge lattice site
in	g5	Concentration distribution function for link 5 at edge lattice site
in	g6	Concentration distribution function for link 6 at edge lattice site
in	g7	Concentration distribution function for link 7 at edge lattice site
in	g8	Concentration distribution function for link 8 at edge lattice site
in	g9	Concentration distribution function for link 9 at edge lattice site
in	g10	Concentration distribution function for link 10 at edge lattice site
in	g11	Concentration distribution function for link 11 at edge lattice site
in	g12	Concentration distribution function for link 12 at edge lattice site
in	g13	Concentration distribution function for link 13 at edge lattice site
in	g14	Concentration distribution function for link 14 at edge lattice site

fD3Q19BoundaryForceDensity()

```
int fD3Q19BoundaryForceDensity (double * force,
                                long tpos,
                                double * p0,
                                int prop)
```

Based on the type of boundary (planar surface, concave edge or corner) and its direction, calculates the forces interaction, buoyancy-driven thermal and constant/oscillating) acting on a given boundary grid point for application of a constant density boundary condition for a three-dimensional D3Q19 lattice. If the simulation is equilibrating, the constant/oscilliating body forces are not applied.

Parameters

out	force	Overall forces acting on fluids at boundary point
in	tpos	Position of boundary lattice site (in one-dimensional form)
in	p0	Fixed fluid densities at boundary site
in	prop	Boundary condition code indicating type and direction

fD3Q19BoundaryForceVelocity()

```
int fD3Q19BoundaryForceVelocity (double * force,
                                 long tpos,
                                 long tpos1,
                                 double dx,
                                 double dy,
                                 double dz,
                                 double * uwall,
                                 int prop)
```

Based on the type of boundary (planar surface, concave edge or corner) and its direction, calculates the forces (interaction, buoyancy-driven thermal and constant/oscillating) acting on a given boundary grid point for application of a constant velocity boundary condition for a three-dimensional D3Q19 lattice. If the simulation is equilibrating, the constant/oscilliating body forces are not applied.

Parameters

out	force	Overall forces acting on fluids at boundary point
in	tpos	Position of boundary lattice site (in one-dimensional form)
in	tpos1	Position of neighbouring lattice site (in one-dimensional form)
in	dx	Change in x-position between boundary site and neighbouring site for Swift free-energy interactions
in	dy	Change in y-position between boundary site and neighbouring site for Swift free-energy interactions
in	dz	Change in z-position between boundary site and neighbouring site for Swift free-energy interactions
in	uwall	Fixed velocity at boundary site
in	prop	Boundary condition code indicating type and direction

fD3Q19OF1()

```
int fD3Q19OF1 (long tpos, int prop)
```

Based on the direction of the boundary (only edges), applies a first-order outflow (zero-gradient) condition to a boundary lattice point for a three-dimensional D3Q19 lattice. This subroutine applies the following to lattice links pointing back into the lattice:

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

using distribution function values at the nearest neighbouring lattice point orthogonal to the boundary.

Parameters

tpos	Position of boundary lattice site (in one-dimensional form)
prop	Boundary condition code indicating type and direction

fD3Q19OF2()

```
int fD3Q19OF2 (long tpos, int prop)
```

Based on the direction of the boundary (only edges), applies a second-order outflow (zero-gradient) condition to a boundary lattice point for a three-dimensional D3Q19 lattice. This subroutine applies the following to lattice links pointing back into the lattice:

$$f_i\left(\vec{x}_w, t^+\right) = 2f_i\left(\vec{x}_w + \Delta x, t^+\right) - f_i\left(\vec{x}_w + 2\Delta x, t^+\right)$$

using distribution function values at the nearest and next-nearest neighbouring lattice points orthogonal to the boundary.

Parameters

tpos	Position of boundary lattice site (in one-dimensional form)
prop	Boundary condition code indicating type and direction

fD3Q19PSFracSite()

```
int fD3Q19PSFracSite (double * frac,
                      double vy,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18)
```

Determines mass fractions of all fluids at a planar surface of a three-dimensional system using known distribution functions at the lattice site for a D3Q19 lattice. The fluid densities are calculated from mass and momentum conservation in the orthogonal direction to the boundary, e.g.

$$\rho\left(1-u_y\right)=f_0+f_1+f_3+f_6+f_7+f_{10}+f_{12}+f_{15}+f_{16}+2\left(f_2+f_4+f_8+f_9+f_{14}\right)$$

The expression in this subroutine is for bottom planar surfaces (PST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for a (different) velocity component.

Parameters

out	frac	Mass fractions of all fluid at edge lattice site
in	vy	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
in	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site

fD3Q19PSSwiftPhi()

```
double fD3Q19PSSwiftPhi (double vy,
                         double g0, double g1, double g2,
                         double g3, double g4, double g5,
                         double g6, double g7, double g8,
                         double g9, double g10, double g11,
                         double g12, double g13, double g14,
                         double g15, double g16, double g17,
                         double g18)
```

Determines concentration of two fluids at a planar surfae of a three-dimensional system using known concentration distribution functions at the lattice site for a D3Q19 lattice. The fluid concentrations are calculated from concentration and momentum conservation in the orthogonal direction to the boundary, e.g.

$$\phi\left(1-u_y\right)=g_0+g_1+g_3+g_6+g_7+g_{12}+g_{15}+g_{16}+2\left(g_2+g_4+g_8+g_9+g_{14}\right)$$

The expression in this subroutine is for bottom planar surfaces (PST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for a (different) velocity component.

Parameters

in	vy	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	g0	Concentration distribution function for link 0 at edge lattice site
in	g1	Concentration distribution function for link 1 at edge lattice site
in	g2	Concentration distribution function for link 2 at edge lattice site
in	g3	Concentration distribution function for link 3 at edge lattice site
in	g4	Concentration distribution function for link 4 at edge lattice site
in	g5	Concentration distribution function for link 5 at edge lattice site
in	g6	Concentration distribution function for link 6 at edge lattice site
in	g7	Concentration distribution function for link 7 at edge lattice site
in	g8	Concentration distribution function for link 8 at edge lattice site
in	g9	Concentration distribution function for link 9 at edge lattice site
in	g10	Concentration distribution function for link 10 at edge lattice site
in	g11	Concentration distribution function for link 11 at edge lattice site
in	g12	Concentration distribution function for link 12 at edge lattice site
in	g13	Concentration distribution function for link 13 at edge lattice site
in	g14	Concentration distribution function for link 14 at edge lattice site
in	g15	Concentration distribution function for link 15 at edge lattice site
in	g16	Concentration distribution function for link 16 at edge lattice site
in	g17	Concentration distribution function for link 17 at edge lattice site
in	g18	Concentration distribution function for link 18 at edge lattice site

fD3Q27BoundaryForceDensity()

```
int fD3Q27BoundaryForceDensity (double * force,
                                long tpos,
                                double * p0,
                                int prop)
```

Based on the type of boundary (planar surface, concave edge or corner) and its direction, calculates the forces interaction, buoyancy-driven thermal and constant/oscillating) acting on a given boundary grid point for application of a constant density boundary condition for a three-dimensional D3Q27 lattice. If the simulation is equilibrating, the constant/oscilliating body forces are not applied.

Parameters

out	force	Overall forces acting on fluids at boundary point
in	tpos	Position of boundary lattice site (in one-dimensional form)
in	p0	Fixed fluid densities at boundary site
in	prop	Boundary condition code indicating type and direction

fD3Q27BoundaryForceVelocity()

```
int fD3Q27BoundaryForceVelocity (double * force,
                                 long tpos,
                                 long tpos1,
                                 double * uwall,
                                 int prop)
```

Based on the type of boundary (planar surface, concave edge or corner) and its direction, calculates the forces (interaction, buoyancy-driven thermal and constant/oscillating) acting on a given boundary grid point for application of a constant velocity boundary condition for a three-dimensional D3Q27 lattice. If the simulation is equilibrating, the constant/oscilliating body forces are not applied.

Parameters

out	force	Overall forces acting on fluids at boundary point
in	tpos	Position of boundary lattice site (in one-dimensional form)
in	tpos1	Position of neighbouring lattice site (in one-dimensional form)
in	uwall	Fixed velocity at boundary site
in	prop	Boundary condition code indicating type and direction

fD3Q27OF1()

```
int fD3Q27OF1 (long tpos, int prop)
```

Based on the direction of the boundary (only edges), applies a first-order outflow (zero-gradient) condition to a boundary lattice point for a three-dimensional D3Q27 lattice. This subroutine applies the following to lattice links pointing back into the lattice:

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

using distribution function values at the nearest neighbouring lattice point orthogonal to the boundary.

Parameters

tpos	Position of boundary lattice site (in one-dimensional form)
prop	Boundary condition code indicating type and direction

fD3Q27OF2()

```
int fD3Q27OF2 (long tpos, int prop)
```

Based on the direction of the boundary (only edges), applies a second-order outflow (zero-gradient) condition to a boundary lattice point for a three-dimensional D3Q27 lattice. This subroutine applies the following to lattice links pointing back into the lattice:

$$f_i\left(\vec{x}_w, t^+\right) = 2f_i\left(\vec{x}_w + \Delta x, t^+\right) - f_i\left(\vec{x}_w + 2\Delta x, t^+\right)$$

using distribution function values at the nearest and next-nearest neighbouring lattice points orthogonal to the boundary.

Parameters

tpos	Position of boundary lattice site (in one-dimensional form)
prop	Boundary condition code indicating type and direction

fD3Q27PSFracSite()

```
int fD3Q27PSFracSite (double * frac,
                      double vy,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26)
```

Determines mass fractions of all fluids at a planar surface of a three-dimensional system using known distribution functions at the lattice site for a D3Q27 lattice. The fluid densities are calculated from mass and momentum conservation in the orthogonal direction to the boundary, e.g.

$$\rho\left(1-u_y\right) = f_0 + f_1 + f_3 + f_6 + f_7 + f_{14} + f_{16} + f_{19} + f_{20} + 2\left(f_2 + f_4 + f_8 + f_9 + f_{10} + f_{11} + f_{18} + f_{25} + f_{26}\right)$$

The expression in this subroutine is for bottom planar surfaces (PST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for a (different) velocity component.

Parameters

out	frac	Mass fractions of all fluid at edge lattice site
in	vy	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
in	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
in	f21	Distribution functions for link 21 at surface lattice site
in	f22	Distribution functions for link 22 at surface lattice site
in	f23	Distribution functions for link 23 at surface lattice site
in	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site

fFixedDensityFluid()

```
int fFixedDensityFluid (long tpos,
                        int prop,
                        int type,
                        double * uwall)
```

Apply a boundary condition at a given lattice point to give constant fluid densities. Four boundary condition schemes are available - Zou-He [158], Inamuro [63], regularised [74] and kinetic [5] - and can be applied at planar surfaces, concave edges and concave corners. The fluid velocity output by this routine is used for solute concentration and temperature boundary conditions, set according to values obtained from applying constant density conditions. The boundary condition type is used to determine the boundary temperature required for Swift free-energy interactions (either a fixed system-wide temperature, the value at a neighbouring lattice point or the specified value for the boundary).

Parameters

in	tpos	Position of boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type (planar surface, concave edge, concave corner) and direction
in	type	Boundary condition type indicating forms of solute and temperature conditions
out	uwall	Resulting fluid velocity at boundary site

fFixedSoluteConcen()

```
int fFixedSoluteConcen (long tpos,
                        int prop,
                        double * uwall)
```

Apply a boundary condition at a given lattice point to give constant solute concentrations. Two boundary condition schemes are available - Zou-He [158] and Inamuro [63] - and can be applied at planar surfaces, concave edges and concave corners. The fluid velocity input into this routine originates from fixed velocity or fixed fluid density boundaries.

Parameters

in	tpos	Position of boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type (planar surface, concave edge, concave corner) and direction
in	uwall	Fluid velocity at boundary site

fFixedSpeedFluid()

```
int fFixedSpeedFluid (long tpos,
                      int prop,
                      int type,
                      double * uwall)
```

Apply a boundary condition at a given lattice point to give a constant fluid velocity. Four boundary condition schemes are available - Zou-He [158], Inamuro [63], regularised [74] and kinetic [5] - and can be applied at planar surfaces, concave edges and concave corners. The constant fluid velocity output by this routine is used for solute concentration and temperature boundary conditions, set according to required constant or oscillating values. The boundary condition type is used to determine the boundary temperature required for Swift free-energy interactions (either a fixed system-wide temperature, the value at a neighbouring lattice point or the specified value for the boundary).

Parameters

in	tpos	Position of boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type (planar surface, concave edge, concave corner) and direction
in	type	Boundary condition type indicating forms of solute and temperature conditions
out	uwall	Fixed fluid velocity at boundary site

fFixedTemperature()

```
int fFixedTemperature (long tpos,
                       int prop,
                       double * uwall)
```

Apply a boundary condition at a given lattice point to give a constant temperature. Two boundary condition schemes are available - Zou-He [158] and Inamuro [63] - and can be applied at planar surfaces, concave edges and concave corners. The fluid velocity input into this routine originates from fixed velocity or fixed fluid density boundaries.

Parameters

in	tpos	Position of boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type (planar surface, concave edge, concave corner) and direction
in	uwall	Fluid velocity at boundary site

fMidBounceBackC()

```
int fMidBounceBackC (long tpos)
```

At the current lattice site, apply a mid-grid bounce-back boundary condition on solute distribution functions, i.e. assign conjugate post-collisional distribution functions from neighbouring lattice sites.

Parameters

in	tpos	Position of current lattice site (in one-dimensional form)

fMidBounceBackF()

```
int fMidBounceBackF (long tpos)
```

At the current lattice site, apply a mid-grid bounce-back boundary condition on fluid distribution functions, i.e. assign conjugate post-collisional distribution functions from neighbouring lattice sites.

Parameters

in	tpos	Position of current lattice site (in one-dimensional form)

fMidBounceBackT()

```
int fMidBounceBackT (long tpos)
```

At the current lattice site, apply a mid-grid bounce-back boundary condition on temperature distribution functions, i.e. assign conjugate post-collisional distribution functions from neighbouring lattice sites.

Parameters

in	tpos	Position of current lattice site (in one-dimensional form)

fMoveNonzeroAway()

```
int fMoveNonzeroAway (long tpos)
```

Move any non-zero distribution functions at the current grid point along each and every lattice link, before zeroing values at that grid point. This subroutine is currently not used but may be useful for certain boundary conditions.

Parameters

tpos	Position of current lattice site (in one-dimensional form)

fNeighbourBoundary()

```
int fNeighbourBoundary ()
```

At each lattice point, neighbouring lattice points are checked to see if any are some kind of boundary point: if so, a number is assigned to an array to indicate how derivatives of fluid properties (e.g. to calculate interaction forces) can be calculated for each lattice point. This value is an integer with the hundreds indicating x-dimension, tens indicating y-dimension and units indicating z-dimension:

- 0 = no restriction in any direction,
- 1 = no restriction in given dimension
- 2 = only one fluid point in positive direction for dimension
- 3 = two fluid points in positive direction for dimension
- 4 = only one fluid point in negative direction for dimension
- 5 = two fluid points in positive direction for dimension
- 6 = no fluid points available in either direction for given dimension

Depending on the number of fluid points available in each direction and dimension, all-neighbour stencils, central difference schemes or one-sided difference schemes may be used to calculate derivatives. The boundary normals are also determined to find directions to apply surface tension forces on multiple fluids. This routine only needs to be called once if boundary conditions do not change during calculations.

fNextStep()

```
long fNextStep (int dx, int dy, int dz, long tpos)
long fNextStep (int q, int xpos, int ypos)
long fNextStep (int q, int xpos, int ypos, int zpos)
```

Starting from a given lattice site, find the next lattice site along a given vector or lattice link, taking any periodic boundaries into account.

Parameters

in	dx	Vector to move from current lattice site (x-component)
in	dy	Vector to move from current lattice site (y-component)
in	dz	Vector to move from current lattice site (z-component)
in	tpos	Position of current lattice site (in one-dimensional form)
in	q	Lattice link to move along
in	xpos	Position of current lattice site (x-coordinate)
in	ypos	Position of current lattice site (y-coordinate)
in	zpos	Position of current lattice site (z-coordinate)

fOutFlow()

```
int fOutFlow (long tpos, int prop)
```

Apply a boundary condition at a given lattice point to give an outflow condition (a Neumann boundary condition with zero gradients in velocity, density, concentration and/or temperature [82]. This type of condition can only be applied along concave edges in two dimensions or planar surfaces in three dimensions, and can either be first-order or second-order based on the number of lattice points used to obtain the required distribution functions at the boundary lattice point.

Parameters

in	tpos	Position of boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type (planar surface, concave edge, concave corner) and direction

fPostCollBoundary()

```
int fPostCollBoundary ()
```

Applies any boundary conditions that need to be applied immediately after collisions but before propagation: the only type of boundary condition that currently needs to be applied at this stage is mid-grid bounce-back.

fPostPropBoundary()

```
int fPostPropBoundary ()
```

Applies any boundary conditions that need to be applied after propagation: most boundary conditions (blank sites, on-grid bounce-back, constant fluid velocity or density, solute concentration and temperature) are applied at this stage.

fsBoundPeriodic()

```
int fsBoundPeriodic ()
```

Applies periodic boundary conditions for customised serial LBE calculations that use boundary halos by copying boundary type values from the edges of fluid points. This is the serial equivalent of an MPI communication used in parallel calculations, and is not generally used in the standard serial version of DL_MESO_LBE (which does not use a boundary halo).

fsBoundPeriodic2D()

```
int fsBoundPeriodic2D ()
```

Applies periodic boundary conditions for customised two-dimensional serial LBE calculations that use boundary halos by copying boundary type values from the edges of fluid points. This is the serial equivalent of an MPI communication used in parallel calculations, and is not generally used in the standard serial version of DL_MESO_LBE.

fsBoundPeriodic3D()

```
int fsBoundPeriodic3D ()
```

Applies periodic boundary conditions for customised three-dimensional serial LBE calculations that use boundary halos by copying boundary type values from the edges of fluid points. This is the serial equivalent of an MPI communication used in parallel calculations, and is not generally used in the standard serial version of DL_MESO_LBE.

fsForcePeriodic()

```
int fsForcePeriodic ()
```

Applies periodic boundary conditions for customised serial LBE calculations that use boundary halos by copying interfacial forces from the edges of fluid points. This is the serial equivalent of an MPI communication used in parallel calculations, and is not generally used in the standard serial version of DL_MESO_LBE (which does not use a boundary halo).

fsForcePeriodic2D()

```
int fsForcePeriodic2D ()
```

Applies periodic boundary conditions for customised two-dimensional serial LBE calculations that use boundary halos by copying interfacial forces from the edges of fluid points. This is the serial equivalent of an MPI communication used in parallel calculations, and is not generally used in the standard serial version of DL_MESO_LBE.

fsForcePeriodic3D()

```
int fsForcePeriodic3D ()
```

Applies periodic boundary conditions for customised three-dimensional serial LBE calculations that use boundary halos by copying interfacial forces from the edges of fluid points. This is the serial equivalent of an MPI communication used in parallel calculations, and is not generally used in the standard serial version of DL_MESO_LBE.

fsIndexPeriodic()

```
int fsIndexPeriodic ()
```

Applies periodic boundary conditions for customised serial LBE calculations that use boundary halos by copying phase indices (for Lishchuk interactions) from the edges of fluid points. This is the serial equivalent of an MPI communication used in parallel calculations, and is not generally used in the standard serial version of DL_MESO_LBE (which does not use a boundary halo).

fsIndexPeriodic2D()

```
int fsIndexPeriodic2D ()
```

Applies periodic boundary conditions for customised two-dimensional serial LBE calculations that use boundary halos by copying phase indices from the edges of fluid points. This is the serial equivalent of an MPI communication used in parallel calculations, and is not generally used in the standard serial version of DL_MESO_LBE.

fsIndexPeriodic3D()

```
int fsIndexPeriodic3D ()
```

Applies periodic boundary conditions for customised three-dimensional serial LBE calculations that use boundary halos by copying phase indices from the edges of fluid points. This is the serial equivalent of an MPI communication used in parallel calculations, and is not generally used in the standard serial version of DL_MESO_LBE.

fSiteBlankC()

```
int fSiteBlankC (long tpos)
```

At the current lattice site, set all distribution function values (along all lattice links) for all solutes to zero. This will ensure e.g. solute diffusion inside a bulk solid is negligible.

Parameters

in	tpos	Position of current lattice site (in one-dimensional form)

fSiteBlankF()

```
int fSiteBlankF (long tpos)
```

At the current lattice site, set all distribution function values (along all lattice links) for all fluids to zero. This will ensure e.g. flows inside solid boundaries are negligible.

Parameters

in	tpos	Position of current lattice site (in one-dimensional form)

fSiteBlankT()

```
int fSiteBlankT (long tpos)
```

At the current lattice site, set all distribution function values (along all lattice links) for the temperature field to zero. This will ensure e.g. negligible heat transfer through an insulator.

Parameters

in	tpos	Position of current lattice site (in one-dimensional form)

fsPeriodic()

```
int fsPeriodic ()
```

Applies periodic boundary conditions for customised serial LBE calculations that use boundary halos by copying distribution functions from the edges of fluid points. This is the serial equivalent of an MPI communication used in parallel calculations, and is not generally used in the standard serial version of DL_MESO_LBE (which does not use a boundary halo).

fsPeriodic2D()

```
int fsPeriodic2D ()
```

Applies periodic boundary conditions for customised two-dimensional serial LBE calculations that use boundary halos by copying distribution functions from the edges of fluid points. This is the serial equivalent of an MPI communication used in parallel calculations, and is not generally used in the standard serial version of DL_MESO_LBE.

fsPeriodic3D()

```
int fsPeriodic3D ()
```

Applies periodic boundary conditions for customised three-dimensional serial LBE calculations that use boundary halos by copying distribution functions from the edges of fluid points. This is the serial equivalent of an MPI communication used in parallel calculations, and is not generally used in the standard serial version of DL_MESO_LBE.

5.25 lbpBOUNDZouHe.cpp

Module for applying Zou-He boundary conditions. (Header file available as lbpBOUNDZouHe.hpp.)

Applies Zou-He [158] boundary conditions at specified lattice points to give fixed fluid velocities or densities, solute concentrations and temperatures. This scheme uses non-equilibrium bounce-back reflection for 'missing' distribution functions re-entering the simulation box, i.e.

$$f_i - f_i^{eq} = f_j - f_j^{eq}$$

where i and j are conjugate links (i.e. $\hat{e}_i = -\hat{e}_j$). The standard Zou-He boundary condition applies this non-equilibrium bounce-back condition unmodified to the orthogonal lattice link only for velocity and density conditions, while the other missing distribution functions are obtained to ensure the overall density and momentum are correct to satisfy the following summations:

$$\rho = \sum_i f_i,$$

$$\rho u_\alpha = \sum_i f_i e_{i,\alpha}.$$

In two dimensions, these conditions are sufficient to find the remaining missing distribution functions. For three-dimensional systems, the non-equilibrium bounce-back condition is applied to the other missing distribution functions along with transverse momentum corrections for tangential directions to provide sufficient unknown quantities for available equations.

The simplified implementation of the Zou-He boundary conditions (for velocity and density conditions only) uses non-equilibrium bounce-back for all 'active' missing distribution functions: as such, the above summations are not necessarily achieved for two-dimensional concave edges and three-dimensional planar surfaces.

For solute concentration and temperature conditions, an 'inverse' approach is taken by using the non-equilibrium bounce-back condition on all missing distribution functions other than the orthogonal lattice link, whose distribution function is determined to ensure correct solute concentration or temperature (in a similar manner to fluid density).

For boundaries other than concave edges in two dimensions or planar surfaces in three dimensions where 'buried' links that neither enter nor leave the simulation box are included, the non-buried 'active' links make use of non-equilibrium bounce-back, while the buried links re obtained by ensuring the overall density and momentum are correct. In cases where there are more unknown buried links than density and momentum equations, each buried link is expressed as a combination of terms for these three or four components, which are solved to give the correct density and momentum at the lattice point.

5.25.1 Functions

- `int` *fD2Q9VCEZouHe()*

  Applies Zou-He constant velocity boundary condition to concave edge for D2Q9 lattice.

- `int` *fD2Q9VCCZouHe()*

  Applies Zou-He constant velocity or density boundary condition to concave corner for D2Q9 lattice.

- `int` *fD2Q9VCECLBEZouHe()*

  Applies Zou-He constant velocity boundary condition to concave edge for D2Q9 lattice with cascaded LBE collisions.

- `int` *fD2Q9VCCCLBEZouHe()*

  Applies Zou-He constant velocity or density boundary condition to concave corner for D2Q9 lattice with cascaded LBE collisions.

- `int` *fD2Q9VCESimpleZouHe()*

  Applies simple Zou-He constant velocity boundary condition to concave edge for D2Q9 lattice.

- `int` *fD2Q9VCCSimpleZouHe()*

  Applies simple Zou-He constant velocity or density boundary condition to concave corner for D2Q9 lattice.
- `int` *fD2Q9VCECLBESimpleZouHe()*

  Applies simple Zou-He constant velocity boundary condition to concave edge for D2Q9 lattice with cascaded LBE collisions.
- `int` *fD2Q9VCCCLBESimpleZouHe()*

  Applies simple Zou-He constant velocity or density boundary condition to concave corner for D2Q9 lattice with cascaded LBE collisions.
- `int` *fD2Q9VFZouHe()*

  Applies constant velocity Zou-He boundary condition to lattice point using D2Q9 lattice scheme.
- `int` *fD2Q9VFSimpleZouHe()*

  Applies constant velocity simple Zou-He boundary condition to lattice point using D2Q9 lattice scheme.
- `int` *fD2Q9PCEZouHe()*

  Applies Zou-He constant density boundary condition to concave edge for D2Q9 lattice.
- `int` *fD2Q9PCESwiftZouHe()*

  Applies Zou-He constant density boundary condition to concave edge for D2Q9 lattice with Swift free-energy interactions.
- `int` *fD2Q9PFZouHe()*

  Applies constant density Zou-He boundary condition to lattice point using D2Q9 lattice scheme.
- `int` *fD2Q9PFSimpleZouHe()*

  Applies constant density simple Zou-He boundary condition to lattice point using D2Q9 lattice scheme.
- `int` *fD2Q9CCEZouHe()*

  Applies Zou-He constant solute concentration boundary condition to concave edge for D2Q9 lattice.
- `int` *fD2Q9CCCZouHe()*

  Applies Zou-He constant solute concentration boundary condition to concave corner for D2Q9 lattice.
- `int` *fD2Q9PCZouHe()*

  Applies constant solute concentration Zou-He boundary condition to lattice point using D2Q9 lattice scheme.
- `int` *fD2Q9TCEZouHe()*

  Applies Zou-He constant temperature boundary condition to concave edge for D2Q9 lattice.
- `int` *fD2Q9TCCZouHe()*

  Applies Zou-He constant temperature boundary condition to concave corner for D2Q9 lattice.
- `int` *fD2Q9PTZouHe()*

  Applies constant temperature Zou-He boundary condition to lattice point using D2Q9 lattice scheme.
- `int` *fD3Q15VPSZouHe()*

  Applies Zou-He constant velocity boundary condition to planar surface for D3Q15 lattice.
- `int` *fD3Q15VCEZouHe()*

  Applies Zou-He constant velocity or density boundary condition to concave edge for D3Q15 lattice.
- `int` *fD3Q15VCCZouHe()*

  Applies Zou-He constant velocity or density boundary condition to concave corner for D3Q15 lattice.

- `int` *fD3Q15VPSSimpleZouHe()*

  Applies simple Zou-He constant velocity boundary condition to planar surface for D3Q15 lattice.
- `int` *fD3Q15VCESimpleZouHe()*

  Applies simple Zou-He constant velocity or density boundary condition to concave edge for D3Q15 lattice.
- `int` *fD3Q15VCCSimpleZouHe()*

  Applies simple Zou-He constant velocity or density boundary condition to concave corner for D3Q15 lattice.
- `int` *fD3Q15VFZouHe()*

  Applies constant velocity Zou-He boundary condition to lattice point using D3Q15 lattice scheme.
- `int` *fD3Q15VFSimpleZouHe()*

  Applies constant velocity simple Zou-He boundary condition to lattice point using D3Q15 lattice scheme.
- `int` *fD3Q15PPSZouHe()*

  Applies Zou-He constant density boundary condition to planar surface for D3Q15 lattice.
- `int` *fD3Q15PPSSwiftZouHe()*

  Applies Zou-He constant density boundary condition to planar surface for D3Q15 lattice with Swift free-energy interactions.
- `int` *fD3Q15PFZouHe()*

  Applies constant density Zou-He boundary condition to lattice point using D3Q15 lattice scheme.
- `int` *fD3Q15PFSimpleZouHe()*

  Applies constant density simple Zou-He boundary condition to lattice point using D3Q15 lattice scheme.
- `int` *fD3Q15CPSZouHe()*

  Applies Zou-He constant solute concentration boundary condition to planar surface for D3Q15 lattice.
- `int` *fD3Q15CCEZouHe()*

  Applies Zou-He constant solute concentration boundary condition to concave edge for D3Q15 lattice.
- `int` *fD3Q15CCCZouHe()*

  Applies Zou-He constant solute concentration boundary condition to concave corner for D3Q15 lattice.
- `int` *fD3Q15PCZouHe()*

  Applies constant solute concentration Zou-He boundary condition to lattice point using D3Q15 lattice scheme.
- `int` *fD3Q15TPSZouHe()*

  Applies Zou-He constant temperature boundary condition to planar surface for D3Q15 lattice.
- `int` *fD3Q15TCEZouHe()*

  Applies Zou-He constant temperature boundary condition to concave edge for D3Q15 lattice.
- `int` *fD3Q15TCCZouHe()*

  Applies Zou-He constant temperature boundary condition to concave corner for D3Q15 lattice.
- `int` *fD3Q15PTZouHe()*

  Applies constant temperature Zou-He boundary condition to lattice point using D3Q15 lattice scheme.
- `int` *fD3Q19VPSZouHe()*

  Applies Zou-He constant velocity boundary condition to planar surface for D3Q19 lattice.
- `int` *fD3Q19VCEZouHe()*

  Applies Zou-He constant velocity or density boundary condition to concave edge for D3Q19 lattice.

- `int` *fD3Q19VCCZouHe()*

  Applies Zou-He constant velocity or density boundary condition to concave corner for D3Q19 lattice.
- `int` *fD3Q19VPSCLBEZouHe()*

  Applies Zou-He constant velocity boundary condition to planar surface for D3Q19 lattice with cascaded LBE collisions.
- `int` *fD3Q19VCECLBEZouHe()*

  Applies Zou-He constant velocity or density boundary condition to concave edge for D3Q19 lattice with cascaded LBE collisions.
- `int` *fD3Q19VCCCLBEZouHe()*

  Applies Zou-He constant velocity or density boundary condition to concave corner for D3Q19 lattice with cascaded LBE collisions.
- `int` *fD3Q19VPSSimpleZouHe()*

  Applies simple Zou-He constant velocity boundary condition to planar surface for D3Q19 lattice.
- `int` *fD3Q19VCESimpleZouHe()*

  Applies simple Zou-He constant velocity or density boundary condition to concave edge for D3Q19 lattice.
- `int` *fD3Q19VCCSimpleZouHe()*

  Applies simple Zou-He constant velocity or density boundary condition to concave corner for D3Q19 lattice.
- `int` *fD3Q19VPSCLBESimpleZouHe()*

  Applies simple Zou-He constant velocity boundary condition to planar surface for D3Q19 lattice with cascaded LBE collisions.
- `int` *fD3Q19VCECLBESimpleZouHe()*

  Applies simple Zou-He constant velocity or density boundary condition to concave edge for D3Q19 lattice with cascaded LBE collisions.
- `int` *fD3Q19VCCCLBESimpleZouHe()*

  Applies simple Zou-He constant velocity or density boundary condition to concave corner for D3Q19 lattice with cascaded LBE collisions.
- `int` *fD3Q19VFZouHe()*

  Applies constant velocity Zou-He boundary condition to lattice point using D3Q19 lattice scheme.
- `int` *fD3Q19VFSimpleZouHe()*

  Applies constant velocity simple Zou-He boundary condition to lattice point using D3Q19 lattice scheme.
- `int` *fD3Q19PPSZouHe()*

  Applies Zou-He constant density boundary condition to planar surface for D3Q19 lattice.
- `int` *fD3Q19PPSSwiftZouHe()*

  Applies Zou-He constant density boundary condition to planar surface for D3Q19 lattice with Swift free-energy interactions.
- `int` *fD3Q19PFZouHe()*

  Applies constant density Zou-He boundary condition to lattice point using D3Q19 lattice scheme.
- `int` *fD3Q19PFSimpleZouHe()*

  Applies constant density simple Zou-He boundary condition to lattice point using D3Q19 lattice scheme.
- `int` *fD3Q19CPSZouHe()*

  Applies Zou-He constant solute concentration boundary condition to planar surface for D3Q19 lattice.

- `int` *fD3Q19CCEZouHe()*

  Applies Zou-He constant solute concentration boundary condition to concave edge for D3Q19 lattice.
- `int` *fD3Q19CCCZouHe()*

  Applies Zou-He constant solute concentration boundary condition to concave corner for D3Q15 lattice.
- `int` *fD3Q19PCZouHe()*

  Applies constant solute concentration Zou-He boundary condition to lattice point using D3Q19 lattice scheme.
- `int` *fD3Q19TPSZouHe()*

  Applies Zou-He constant temperature boundary condition to planar surface for D3Q19 lattice.
- `int` *fD3Q19TCEZouHe()*

  Applies Zou-He constant temperature boundary condition to concave edge for D3Q19 lattice.
- `int` *fD3Q19TCCZouHe()*

  Applies Zou-He constant temperature boundary condition to concave corner for D3Q19 lattice.
- `int` *fD3Q19PTZouHe()*

  Applies constant temperature Zou-He boundary condition to lattice point using D3Q19 lattice scheme.
- `int` *fD3Q27VPSZouHe()*

  Applies Zou-He constant velocity boundary condition to planar surface for D3Q27 lattice.
- `int` *fD3Q27VCEZouHe()*

  Applies Zou-He constant velocity or density boundary condition to concave edge for D3Q27 lattice.
- `int` *fD3Q27VCCZouHe()*

  Applies Zou-He constant velocity or density boundary condition to concave corner for D3Q27 lattice.
- `int` *fD3Q27VPSCLBEZouHe()*

  Applies Zou-He constant velocity boundary condition to planar surface for D3Q27 lattice with cascaded LBE collisions.
- `int` *fD3Q27VCECLBEZouHe()*

  Applies Zou-He constant velocity or density boundary condition to concave edge for D3Q27 lattice with cascaded LBE collisions.
- `int` *fD3Q27VCCCLBEZouHe()*

  Applies Zou-He constant velocity or density boundary condition to concave corner for D3Q27 lattice with cascaded LBE collisions.
- `int` *fD3Q27VPSSimpleZouHe()*

  Applies simple Zou-He constant velocity boundary condition to planar surface for D3Q27 lattice.
- `int` *fD3Q27VCESimpleZouHe()*

  Applies simple Zou-He constant velocity or density boundary condition to concave edge for D3Q27 lattice.
- `int` *fD3Q27VCCSimpleZouHe()*

  Applies simple Zou-He constant velocity or density boundary condition to concave corner for D3Q27 lattice.
- `int` *fD3Q27VPSCLBESimpleZouHe()*

  Applies simple Zou-He constant velocity boundary condition to planar surface for D3Q27 lattice with cascaded LBE collisions.

- `int` *fD3Q27VCECLBESimpleZouHe()*

  Applies simple Zou-He constant velocity or density boundary condition to concave edge for D3Q27 lattice with cascaded LBE collisions.

- `int` *fD3Q27VCCCLBESimpleZouHe()*

  Applies simple Zou-He constant velocity or density boundary condition to concave corner for D3Q27 lattice with cascaded LBE collisions.

- `int` *fD3Q27VFZouHe()*

  Applies constant velocity Zou-He boundary condition to lattice point using D3Q27 lattice scheme.

- `int` *fD3Q27VFSimpleZouHe()*

  Applies constant velocity simple Zou-He boundary condition to lattice point using D3Q27 lattice scheme.

- `int` *fD3Q27PPSZouHe()*

  Applies Zou-He constant density boundary condition to planar surface for D3Q27 lattice.

- `int` *fD3Q27PFZouHe()*

  Applies constant density Zou-He boundary condition to lattice point using D3Q27 lattice scheme.

- `int` *fD3Q27PFSimpleZouHe()*

  Applies constant density simple Zou-He boundary condition to lattice point using D3Q27 lattice scheme.

- `int` *fD3Q27CPSZouHe()*

  Applies Zou-He constant solute concentration boundary condition to planar surface for D3Q27 lattice.

- `int` *fD3Q27CCEZouHe()*

  Applies Zou-He constant solute concentration boundary condition to concave edge for D3Q27 lattice.

- `int` *fD3Q27CCCZouHe()*

  Applies Zou-He constant solute concentration boundary condition to concave corner for D3Q27 lattice.

- `int` *fD3Q27PCZouHe()*

  Applies constant solute concentration Zou-He boundary condition to lattice point using D3Q27 lattice scheme.

- `int` *fD3Q27TPSZouHe()*

  Applies Zou-He constant temperature boundary condition to planar surface for D3Q27 lattice.

- `int` *fD3Q27TCEZouHe()*

  Applies Zou-He constant temperature boundary condition to concave edge for D3Q27 lattice.

- `int` *fD3Q27TCCZouHe()*

  Applies Zou-He constant temperature boundary condition to concave corner for D3Q27 lattice.

- `int` *fD3Q27PTZouHe()*

  Applies constant temperature Zou-He boundary condition to lattice point using D3Q27 lattice scheme.

5.25.2 Function Documentation

fD2Q9CCCZouHe()

```
int fD2Q9CCCZouHe (double * p,
                   double v0, double v1,
                   double * f0, double * f1, double * f2,
                   double * f3, double * f4, double * f5,
                   double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed solute concentrations at a concave corner using the two-dimensional D2Q9 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for the bottom-left concave corner (CCCTRF) but can be used for any concave corner by selecting different distribution functions.

Parameters

in	p	Solute concentrations for boundary lattice point
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site

fD2Q9CCEZouHe()

```
int fD2Q9CCEZouHe (double * p,
                   double v0, double v1,
                   double * f0, double * f1, double * f2,
                   double * f3, double * f4, double * f5,
                   double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed solute concentrations at a concave edge using the two-dimensional D2Q9 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom concave edges (CCETF) but can be used for any concave edge by selecting different distribution functions.

Parameters

in	p	Solute concentrations for boundary lattice point
in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site

fD2Q9PCESwiftZouHe()

```
int fD2Q9PCESwiftZouHe (double * p,
                        double * force,
                        double * f0, double * f1, double * f2,
                        double * f3, double * f4, double * f5,
                        double * f6, double * f7, double * f8,
                        double & vel)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed fluid densities at a concave edge using the two-dimensional D2Q9 lattice and Swift free-energy interactions. The resulting orthogonal velocity component is calculated using the density distribution functions: this value is subsequently used for concentration distribution functions (if required for two-fluid systems) as well as solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom concave edges (PCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for density/concentration gradients (which may be swapped around).

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD2Q9PCEZouHe()

```
int fD2Q9PCEZouHe (double * p,
                   double * force,
                   double * f0, double * f1, double * f2,
                   double * f3, double * f4, double * f5,
                   double * f6, double * f7, double * f8,
                   double & vel)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed fluid densities at a concave edge using the two-dimensional D2Q9 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions: this routine can also be used for systems with cascaded LBE collisions as the local equilibrium distribution functions for these (with zero tangential velocity) result in the same expressions for missing distribution functions. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom concave edges (PCETF) but can be used for any concave edge by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD2Q9PCZouHe()

```
int fD2Q9PCZouHe (long tpos,
                  int prop,
                  double * p0,
                  double * uwall)
```

Applies the appropriate Zou-He boundary condition for constant solute concentrations based on direction (concave edges and corners) for a two-dimensional D2Q9 lattice.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Solute concentrations for boundary lattice point
in	uwall	Velocity at boundary site determined from applying constant velocity/density boundary condition

fD2Q9PFSimpleZouHe()

```
int fD2Q9PFSimpleZouHe (long tpos,
                        int prop,
                        double * p0,
                        double * uwall)
```

Applies the appropriate simple Zou-He boundary condition for constant fluid densities based on types of collisions, interactions and direction for a two-dimensional D2Q9 lattice. (In this case, there are boundary options for cascaded LBE collisions, Swift free-energy interactions, as well as concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying simple Zou-He boundary condition

fD2Q9PFZouHe()

```
int fD2Q9PFZouHe (long tpos,
                  int prop,
                  double * p0,
                  double * uwall)
```

Applies the appropriate Zou-He boundary condition for constant fluid densities based on types of collisions, interactions and direction for a two-dimensional D2Q9 lattice. (In this case, there are boundary options for cascaded LBE collisions, Swift free-energy interactions, as well as concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying Zou-He boundary condition

fD2Q9PTZouHe()

```
int fD2Q9PTZouHe (long tpos,
                  int prop,
                  double p0,
                  double * uwall)
```

Applies the appropriate Zou-He boundary condition for a constant temperature based on direction (concave edges and corners) for a two-dimensional D2Q9 lattice.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Solute concentrations for boundary lattice point
in	uwall	Velocity at boundary site determined from applying constant velocity/density boundary condition

fD2Q9TCCZouHe()

```
int fD2Q9TCCZouHe (double p,
                   double v0, double v1,
                   double * f0, double * f1, double * f2,
                   double * f3, double * f4, double * f5,
                   double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed temperature at a concave corner using the two-dimensional D2Q9 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for the bottom-left concave corner (TCCTRF) but can be used for any concave corner by selecting different distribution functions.

Parameters

in	p	Temperature for boundary lattice point
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site

fD2Q9TCEZouHe()

```
int fD2Q9TCEZouHe (double p,
                   double v0, double v1,
                   double * f0, double * f1, double * f2,
                   double * f3, double * f4, double * f5,
                   double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed temperature at a concave edge using the two-dimensional D2Q9 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom concave edges (TCETF) but can be used for any concave edge by selecting different distribution functions.

Parameters

in	p	Temperature for boundary lattice point
in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site

fD2Q9VCCCLBESimpleZouHe()

```
int fD2Q9VCCCLBESimpleZouHe (double * p,
                             double v0, double v1,
                             double * force,
                             double * f0, double * f1, double * f2,
                             double * f3, double * f4, double * f5,
                             double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity or density at a concave corner using the two-dimensional D2Q9 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site

fD2Q9VCCCLBEZouHe()

```
int fD2Q9VCCCLBEZouHe (double * p,
                       double v0, double v1,
                       double * force,
                       double * f0, double * f1, double * f2,
                       double * f3, double * f4, double * f5,
                       double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity or density at a concave corner using the two-dimensional D2Q9 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site

fD2Q9VCCSimpleZouHe()

```
int fD2Q9VCCSimpleZouHe (double * p,
                         double v0, double v1,
                         double * force,
                         double * f0, double * f1, double * f2,
                         double * f3, double * f4, double * f5,
                         double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity or density at a concave corner using the two-dimensional D2Q9 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions, including those with Swift free-energy interactions as the differences between local equiibrium distribution functions for conjugate links eliminate all density/concentration gradient and Galilean invariance terms. The expressions in this subroutine are for bottom-left concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site

fD2Q9VCCZouHe()

```
int fD2Q9VCCZouHe (double * p,
                   double v0, double v1,
                   double * force,
                   double * f0, double * f1, double * f2,
                   double * f3, double * f4, double * f5,
                   double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity or density at a concave corner using the two-dimensional D2Q9 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions, including those with Swift free-energy interactions as the differences between local equiibrium distribution functions for conjugate links eliminate all density/concentration gradient and Galilean invariance terms. The expressions in this subroutine are for bottom-left concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site

fD2Q9VCECLBESimpleZouHe()

```
int fD2Q9VCECLBESimpleZouHe (double v0, double v1,
                             double * force,
                             double * f0, double * f1, double * f2,
                             double * f3, double * f4, double * f5,
                             double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity at a concave edge using the two-dimensional D2Q9 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom concave edges (VCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site

fD2Q9VCECLBEZouHe()

```
int fD2Q9VCECLBEZouHe (double v0, double v1,
                       double * force,
                       double * f0, double * f1, double * f2,
                       double * f3, double * f4, double * f5,
                       double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity at a concave edge using the two-dimensional D2Q9 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom concave edges (VCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site

fD2Q9VCESimpleZouHe()

```
int fD2Q9VCESimpleZouHe (double v0, double v1,
                         double * force,
                         double * f0, double * f1, double * f2,
                         double * f3, double * f4, double * f5,
                         double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity at a concave edge using the two-dimensional D2Q9 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions, including those with Swift free-energy interactions as the differences between local equiibrium distribution functions for conjugate links eliminate all density/concentration gradient and Galilean invariance terms. The expressions in this subroutine are for bottom concave edges (VCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site

fD2Q9VCEZouHe()

```
int fD2Q9VCEZouHe (double v0, double v1,
                   double * force,
                   double * f0, double * f1, double * f2,
                   double * f3, double * f4, double * f5,
                   double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity at a concave edge using the two-dimensional D2Q9 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions, including those with Swift free-energy interactions as the differences between local equiibrium distribution functions for conjugate links eliminate all density/concentration gradient and Galilean invariance terms. The expressions in this subroutine are for bottom concave edges (VCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site

fD2Q9VFSimpleZouHe()

```
int fD2Q9VFSimpleZouHe (long tpos,
                        long tpos1,
                        int prop,
                        double * uwall,
                        double dx,
                        double dy)
```

Applies the appropriate simple Zou-He boundary condition for a constant velocity based on types of collisions and direction for a two-dimensional D2Q9 lattice. (In this case, there are boundary options for cascaded LBE collisions as well as concave edges and corners.) For corners with Swift free-energy interactions, the vector between the boundary lattice point and sampling point for densities can be specified to correct fluid density/concentration using gradients of those properties evaluated at the boundary point.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	tpos1	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site
in	dx	Vector to move from current lattice site (x-component)
in	dy	Vector to move from current lattice site (y-component)

fD2Q9VFZouHe()

```
int fD2Q9VFZouHe (long tpos,
                  long tpos1,
                  int prop,
                  double * uwall,
                  double dx,
                  double dy)
```

Applies the appropriate Zou-He boundary condition for a constant velocity based on types of collisions and direction for a two-dimensional D2Q9 lattice. (In this case, there are boundary options for cascaded LBE collisions as well as concave edges and corners.) For corners with Swift free-energy interactions, the vector between the boundary lattice point and sampling point for densities can be specified to correct fluid density/concentration using gradients of those properties evaluated at the boundary point.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	tpos1	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site
in	dx	Vector to move from current lattice site (x-component)
in	dy	Vector to move from current lattice site (y-component)

fD3Q15CCCZouHe()

```
int fD3Q15CCCZouHe (double * p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed solute concentrations at a concave corner using the three-dimensional D3Q15 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom-left-back concave corners (CCCTRF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations at concave corner
in	v0	Velocity component at concave corner (x-component for bottom-left-back corner)
in	v1	Velocity component at concave corner (y-component for bottom-left-back corner)
in	v2	Velocity component at concave corner (z-component for bottom-left-back corner)
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site

fD3Q15CCEZouHe()

```
int fD3Q15CCEZouHe (double * p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed solute concentrations at a concave edge using the three-dimensional D3Q15 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom-left concave edges (CCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations at concave edge
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site

fD3Q15CPSZouHe()

```
int fD3Q15CPSZouHe (double * p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed solute concentrations at a planar surface using the three-dimensional D3Q15 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom planar surfaces (CPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations for boundary lattice point
in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site

fD3Q15PCZouHe()

```
int fD3Q15PCZouHe (long tpos,
                   int prop,
                   double * p0,
                   double * uwall)
```

Applies the appropriate Zou-He boundary condition for constant solute concentrations based on direction (planar surface, concave edges and corners) for a three-dimensional D3Q15 lattice.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Solute concentrations for boundary lattice point
in	uwall	Velocity at boundary site determined from applying constant velocity/density boundary condition

fD3Q15PFSimpleZouHe()

```
int fD3Q15PFSimpleZouHe (long tpos,
                         int prop,
                         double * p0,
                         double * uwall)
```

Applies the appropriate simple Zou-He boundary condition for constant fluid densities based on types of collisions, interactions and direction for a three-dimensional D3Q15 lattice. (In this case, there are boundary options for Swift free-energy interactions, as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying simple Zou-He boundary condition

fD3Q15PFZouHe()

```
int fD3Q15PFZouHe (long tpos,
                   int prop,
                   double * p0,
                   double * uwall)
```

Applies the appropriate Zou-He boundary condition for constant fluid densities based on types of collisions, interactions and direction for a three-dimensional D3Q15 lattice. (In this case, there are boundary options for Swift free-energy interactions, as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying Zou-He boundary condition

fD3Q15PPSSwiftZouHe()

```
int fD3Q15PPSSwiftZouHe (double * p,
                         double * force,
                         double * f0, double * f1, double * f2,
                         double * f3, double * f4, double * f5,
                         double * f6, double * f7, double * f8,
                         double * f9, double * f10, double * f11,
                         double * f12, double * f13, double * f14,
                         double & vel)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q15 lattice and Swift free-energy interactions. The resulting orthogonal velocity component is calculated using the density distribution functions: this value is subsequently used for concentration distribution functions (if required for two-fluid systems) as well as solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q15PPSZouHe()

```
int fD3Q15PPSZouHe (double * p,
                    double * force,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double & vel)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q15PTZouHe()

```
int fD3Q15PTZouHe (long tpos,
                   int prop,
                   double p0,
                   double * uwall)
```

Applies the appropriate Zou-He boundary condition for constant temperature based on direction (planar surface, concave edges and corners) for a three-dimensional D3Q15 lattice.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Temperature for boundary lattice point
in	uwall	Velocity at boundary site determined from applying constant velocity/density boundary condition

fD3Q15TCCZouHe()

```
int fD3Q15TCCZouHe (double p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed temperature at a concave corner using the three-dimensional D3Q15 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom-left-back concave corners (TCCTRF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature at concave corner
in	v0	Velocity component at concave corner (x-component for bottom-left-back corner)
in	v1	Velocity component at concave corner (y-component for bottom-left-back corner)
in	v2	Velocity component at concave corner (z-component for bottom-left-back corner)
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site

fD3Q15TCEZouHe()

```
int fD3Q15TCEZouHe (double p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed temperature at a concave edge using the three-dimensional D3Q15 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom-left concave edges (TCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature at concave edge
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site

fD3Q15TPSZouHe()

```
int fD3Q15TPSZouHe (double p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed temperature at a planar surface using the three-dimensional D3Q15 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom planar surfaces (TPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature for boundary lattice point
in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site

fD3Q15VCCSimpleZouHe()

```
int fD3Q15VCCSimpleZouHe (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions, including those with Swift free-energy interactions as the differences between local equiibrium distribution functions for conjugate links eliminate all density/concentration gradient and Galilean invariance terms. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site

fD3Q15VCCZouHe()

```
int fD3Q15VCCZouHe (double * p,
                    double v0, double v1, double v2,
                    double * force,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions, including those with Swift free-energy interactions as the differences between local equiibrium distribution functions for conjugate links eliminate all density/concentration gradient and Galilean invariance terms. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site

fD3Q15VCESimpleZouHe()

```
int fD3Q15VCESimpleZouHe (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions, including those with Swift free-energy interactions as the differences between local equiibrium distribution functions for conjugate links eliminate all density/concentration gradient and Galilean invariance terms. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site

fD3Q15VCEZouHe()

```
int fD3Q15VCEZouHe (double * p,
                    double v0, double v1, double v2,
                    double * force,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions, including those with Swift free-energy interactions as the differences between local equiibrium distribution functions for conjugate links eliminate all density/concentration gradient and Galilean invariance terms. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site

fD3Q15VFSimpleZouHe()

```
int fD3Q15VFSimpleZouHe (long tpos,
                         long rpos,
                         int prop,
                         double * uwall,
                         double dx,
                         double dy,
                         double dz)
```

Applies the appropriate simple Zou-He boundary condition for a constant velocity based on direction for a three-dimensional D3Q15 lattice. (In this case, there are boundary options for planar surfaces, concave edges and corners.) For edges and corners with Swift free-energy interactions, the vector between the boundary lattice point and sampling point for densities can be specified to correct fluid density/concentration using gradients of those properties evalulated at the boundary point.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	rpos	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site
in	dx	Vector to move from current lattice site (x-component)
in	dy	Vector to move from current lattice site (y-component)
in	dz	Vector to move from current lattice site (z-component)

fD3Q15VFZouHe()

```
int fD3Q15VFZouHe (long tpos,
                   long rpos,
                   int prop,
                   double * uwall,
                   double dx,
                   double dy,
                   double dz)
```

Applies the appropriate Zou-He boundary condition for a constant velocity based on direction for a three-dimensional D3Q15 lattice. (In this case, there are boundary options for planar surfaces, concave edges and corners.) For edges and corners with Swift free-energy interactions, the vector between the boundary lattice point and sampling point for densities can be specified to correct fluid density/concentration using gradients of those properties evaluated at the boundary point.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	rpos	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site
in	dx	Vector to move from current lattice site (x-component)
in	dy	Vector to move from current lattice site (y-component)
in	dz	Vector to move from current lattice site (z-component)

fD3Q15VPSSimpleZouHe()

```
int fD3Q15VPSSimpleZouHe (double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions, including those with Swift free-energy interactions as the differences between local equiibrium distribution functions for conjugate links eliminate all density/concentration gradient and Galilean invariance terms. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site

fD3Q15VPSZouHe()

```
int fD3Q15VPSZouHe (double v0, double v1, double v2,
                    double * force,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions, including those with Swift free-energy interactions as the differences between local equiibrium distribution functions for conjugate links eliminate all density/concentration gradient and Galilean invariance terms. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site

fD3Q19CCCZouHe()

```
int fD3Q19CCCZouHe (double * p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed solute concentrations at a concave corner using the three-dimensional D3Q15 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom-left-back concave corners (CCCTRF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations at concave corner
in	v0	Velocity component at concave corner (x-component for bottom-left-back corner)
in	v1	Velocity component at concave corner (y-component for bottom-left-back corner)
in	v2	Velocity component at concave corner (z-component for bottom-left-back corner)
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site

fD3Q19CCEZouHe()

```
int fD3Q19CCEZouHe (double * p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed solute concentrations at a concave edge using the three-dimensional D3Q19 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom-left concave edges (CCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations at concave edge
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site

fD3Q19CPSZouHe()

```
int fD3Q19CPSZouHe (double * p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed solute concentrations at a planar surface using the three-dimensional D3Q19 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom planar surfaces (CPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations for boundary lattice point
in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site

fD3Q19PCZouHe()

```
int fD3Q19PCZouHe (long tpos,
                   int prop,
                   double * p0,
                   double * uwall)
```

Applies the appropriate Zou-He boundary condition for constant solute concentrations based on direction (planar surface, concave edges and corners) for a three-dimensional D3Q19 lattice.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Solute concentrations for boundary lattice point
in	uwall	Velocity at boundary site determined from applying constant velocity/density boundary condition

fD3Q19PFSimpleZouHe()

```
int fD3Q19PFSimpleZouHe (long tpos,
                         int prop,
                         double * p0,
                         double * uwall)
```

Applies the appropriate simple Zou-He boundary condition for constant fluid densities based on types of collisions, interactions and direction for a three-dimensional D3Q19 lattice. (In this case, there are boundary options for cascaded LBE collisions, Swift free-energy interactions, as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying simple Zou-He boundary condition

fD3Q19PFZouHe()

```
int fD3Q19PFZouHe (long tpos,
                   int prop,
                   double * p0,
                   double * uwall)
```

Applies the appropriate Zou-He boundary condition for constant fluid densities based on types of collisions, interactions and direction for a three-dimensional D3Q19 lattice. (In this case, there are boundary options for cascaded LBE collisions, Swift free-energy interactions, as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying Zou-He boundary condition

fD3Q19PPSSwiftZouHe()

```
int fD3Q19PPSSwiftZouHe (double * p,
                         double * force,
                         double * f0, double * f1, double * f2,
                         double * f3, double * f4, double * f5,
                         double * f6, double * f7, double * f8,
                         double * f9, double * f10, double * f11,
                         double * f12, double * f13, double * f14,
                         double * f15, double * f16, double * f17,
                         double * f18,
                         double & vel)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q19 lattice and Swift free-energy interactions. The resulting orthogonal velocity component is calculated using the density distribution functions: this value is subsequently used for concentration distribution functions (if required for two-fluid systems) as well as solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q19PPSZouHe()

```
int fD3Q19PPSZouHe (double * p,
                    double * force,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18,
                    double & vel)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions: this routine can also be used for systems with cascaded LBE collisions as the local equilibrium distribution functions for these (with zero tangential velocity) result in the same expressions for missing distribution functions. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q19PTZouHe()

```
int fD3Q19PTZouHe (long tpos,
                   int prop,
                   double p0,
                   double * uwall)
```

Applies the appropriate Zou-He boundary condition for constant temperature based on direction (planar surface, concave edges and corners) for a three-dimensional D3Q19 lattice.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Temperature for boundary lattice point
in	uwall	Velocity at boundary site determined from applying constant velocity/density boundary condition

fD3Q19TCCZouHe()

```
int fD3Q19TCCZouHe (double p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed temperature at a concave corner using the three-dimensional D3Q19 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom-left-back concave corners (TCCTRF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature at concave corner
in	v0	Velocity component at concave corner (x-component for bottom-left-back corner)
in	v1	Velocity component at concave corner (y-component for bottom-left-back corner)
in	v2	Velocity component at concave corner (z-component for bottom-left-back corner)
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site

fD3Q19TCEZouHe()

```
int fD3Q19TCEZouHe (double p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed temperature at a concave edge using the three-dimensional D3Q19 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom-left concave edges (TCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature at concave edge
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site

fD3Q19TPSZouHe()

```
int fD3Q19TPSZouHe (double p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed temperature at a planar surface using the three-dimensional D3Q19 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom planar surfaces (TPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature for boundary lattice point
in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site

fD3Q19VCCCLBESimpleZouHe()

```
int fD3Q19VCCCLBESimpleZouHe (double * p,
                              double v0, double v1, double v2,
                              double * force,
                              double * f0, double * f1, double * f2,
                              double * f3, double * f4, double * f5,
                              double * f6, double * f7, double * f8,
                              double * f9, double * f10, double * f11,
                              double * f12, double * f13, double * f14,
                              double * f15, double * f16, double * f17,
                              double * f18)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q19 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site

fD3Q19VCCCLBEZouHe()

```
int fD3Q19VCCCLBEZouHe (double * p,
                        double v0, double v1, double v2,
                        double * force,
                        double * f0, double * f1, double * f2,
                        double * f3, double * f4, double * f5,
                        double * f6, double * f7, double * f8,
                        double * f9, double * f10, double * f11,
                        double * f12, double * f13, double * f14,
                        double * f15, double * f16, double * f17,
                        double * f18)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q19 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site

fD3Q19VCCSimpleZouHe()

```
int fD3Q19VCCSimpleZouHe (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions, including those with Swift free-energy interactions as the differences between local equiibrium distribution functions for conjugate links eliminate all density/concentration gradient and Galilean invariance terms. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site

fD3Q19VCCZouHe()

```
int fD3Q19VCCZouHe (double * p,
                    double v0, double v1, double v2,
                    double * force,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions, including those with Swift free-energy interactions as the differences between local equiibrium distribution functions for conjugate links eliminate all density/concentration gradient and Galilean invariance terms. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site

fD3Q19VCECLBESimpleZouHe()

```
int fD3Q19VCECLBESimpleZouHe (double * p,
                              double v0, double v1, double v2,
                              double * force,
                              double * f0, double * f1, double * f2,
                              double * f3, double * f4, double * f5,
                              double * f6, double * f7, double * f8,
                              double * f9, double * f10, double * f11,
                              double * f12, double * f13, double * f14,
                              double * f15, double * f16, double * f17,
                              double * f18)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q19 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site

fD3Q19VCECLBEZouHe()

```
int fD3Q19VCECLBEZouHe (double * p,
                        double v0, double v1, double v2,
                        double * force,
                        double * f0, double * f1, double * f2,
                        double * f3, double * f4, double * f5,
                        double * f6, double * f7, double * f8,
                        double * f9, double * f10, double * f11,
                        double * f12, double * f13, double * f14,
                        double * f15, double * f16, double * f17,
                        double * f18)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q19 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site

fD3Q19VCESimpleZouHe()

```
int fD3Q19VCESimpleZouHe (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions, including those with Swift free-energy interactions as the differences between local equiibrium distribution functions for conjugate links eliminate all density/concentration gradient and Galilean invariance terms. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site

fD3Q19VCEZouHe()

```
int fD3Q19VCEZouHe (double * p,
                    double v0, double v1, double v2,
                    double * force,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions, including those with Swift free-energy interactions as the differences between local equiibrium distribution functions for conjugate links eliminate all density/concentration gradient and Galilean invariance terms. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site

fD3Q19VFSimpleZouHe()

```
int fD3Q19VFSimpleZouHe (long tpos,
                         long rpos,
                         int prop,
                         double * uwall,
                         double dx,
                         double dy,
                         double dz)
```

Applies the appropriate simple Zou-He boundary condition for a constant velocity based on types of collisions and direction for a three-dimensional D3Q19 lattice. (In this case, there are boundary options for cascaded LBE collsions as well as planar surfaces, concave edges and corners.) For edges and corners with Swift free-energy interactions, the vector between the boundary lattice point and sampling point for densities can be specified to correct fluid density/concentration using gradients of those properties evalulated at the boundary point.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	rpos	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site
in	dx	Vector to move from current lattice site (x-component)
in	dy	Vector to move from current lattice site (y-component)
in	dz	Vector to move from current lattice site (z-component)

fD3Q19VFZouHe()

```
int fD3Q19VFZouHe (long tpos,
                   long rpos,
                   int prop,
                   double * uwall,
                   double dx,
                   double dy,
                   double dz)
```

Applies the appropriate Zou-He boundary condition for a constant velocity based on types of collisions and direction for a three-dimensional D3Q19 lattice. (In this case, there are boundary options for cascaded LBE collsions as well as planar surfaces, concave edges and corners.) For edges and corners with Swift free-energy interactions, the vector between the boundary lattice point and sampling point for densities can be specified to correct fluid density/concentration using gradients of those properties evalulated at the boundary point.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	rpos	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site
in	dx	Vector to move from current lattice site (x-component)
in	dy	Vector to move from current lattice site (y-component)
in	dz	Vector to move from current lattice site (z-component)

fD3Q19VPSCLBESimpleZouHe()

```
int fD3Q19VPSCLBESimpleZouHe (double v0, double v1, double v2,
                              double * force,
                              double * f0, double * f1, double * f2,
                              double * f3, double * f4, double * f5,
                              double * f6, double * f7, double * f8,
                              double * f9, double * f10, double * f11,
                              double * f12, double * f13, double * f14,
                              double * f15, double * f16, double * f17,
                              double * f18)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q19 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site

fD3Q19VPSCLBEZouHe()

```
int fD3Q19VPSCLBEZouHe (double v0, double v1, double v2,
                        double * force,
                        double * f0, double * f1, double * f2,
                        double * f3, double * f4, double * f5,
                        double * f6, double * f7, double * f8,
                        double * f9, double * f10, double * f11,
                        double * f12, double * f13, double * f14,
                        double * f15, double * f16, double * f17,
                        double * f18)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q19 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site

fD3Q19VPSSimpleZouHe()

```
int fD3Q19VPSSimpleZouHe (double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions, including those with Swift free-energy interactions as the differences between local equiibrium distribution functions for conjugate links eliminate all density/concentration gradient and Galilean invariance terms. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site

fD3Q19VPSZouHe()

```
int fD3Q19VPSZouHe (double v0, double v1, double v2,
                    double * force,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions, including those with Swift free-energy interactions as the differences between local equiibrium distribution functions for conjugate links eliminate all density/concentration gradient and Galilean invariance terms. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site

fD3Q27CCCZouHe()

```
int fD3Q27CCCZouHe (double * p,
                     double v0, double v1, double v2,
                     double * f0, double * f1, double * f2,
                     double * f3, double * f4, double * f5,
                     double * f6, double * f7, double * f8,
                     double * f9, double * f10, double * f11,
                     double * f12, double * f13, double * f14,
                     double * f15, double * f16, double * f17,
                     double * f18, double * f19, double * f20,
                     double * f21, double * f22, double * f23,
                     double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed solute concentrations at a concave corner using the three-dimensional D3Q27 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom-left-back concave corners (CCCTRF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations at concave corner
in	v0	Velocity component at concave corner (x-component for bottom-left-back corner)
in	v1	Velocity component at concave corner (y-component for bottom-left-back corner)
in	v2	Velocity component at concave corner (z-component for bottom-left-back corner)
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site

continues on next page

Table 5.6 – continued from previous page

in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site
out	f19	Distribution functions for link 19 at corner lattice site
out	f20	Distribution functions for link 20 at corner lattice site
out	f21	Distribution functions for link 21 at corner lattice site
out	f22	Distribution functions for link 22 at corner lattice site
out	f23	Distribution functions for link 23 at corner lattice site
out	f24	Distribution functions for link 24 at corner lattice site
out	f25	Distribution functions for link 25 at corner lattice site
out	f26	Distribution functions for link 26 at corner lattice site

fD3Q27CCEZouHe()

```
int fD3Q27CCEZouHe (double * p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18, double * f19, double * f20,
                    double * f21, double * f22, double * f23,
                    double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed solute concentrations at a concave edge using the three-dimensional D3Q27 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom-left concave edges (CCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations at concave edge
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site

continues on next page

Table 5.7 – continued from previous page

in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
in	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
in	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
out	f19	Distribution functions for link 19 at edge lattice site
out	f20	Distribution functions for link 20 at edge lattice site
out	f21	Distribution functions for link 21 at edge lattice site
out	f22	Distribution functions for link 22 at edge lattice site
out	f23	Distribution functions for link 23 at edge lattice site
out	f24	Distribution functions for link 24 at edge lattice site
out	f25	Distribution functions for link 25 at edge lattice site
out	f26	Distribution functions for link 26 at edge lattice site

fD3Q27CPSZouHe()

```
int fD3Q27CPSZouHe (double * p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18, double * f19, double * f20,
                    double * f21, double * f22, double * f23,
                    double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed solute concentrations at a planar surface using the three-dimensional D3Q27 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom planar surfaces (CPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations for boundary lattice point
in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site

continues on next page

Table 5.8 – continued from previous page

in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site
out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site

fD3Q27PCZouHe()

```
int fD3Q27PCZouHe (long tpos,
                   int prop,
                   double * p0,
                   double * uwall)
```

Applies the appropriate Zou-He boundary condition for constant solute concentrations based on direction (planar surface, concave edges and corners) for a three-dimensional D3Q27 lattice.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Solute concentrations for boundary lattice point
in	uwall	Velocity at boundary site determined from applying constant velocity/density boundary condition

fD3Q27PFSimpleZouHe()

```
int fD3Q27PFSimpleZouHe (long tpos,
                         int prop,
                         double * p0,
                         double * uwall)
```

Applies the appropriate simple Zou-He boundary condition for constant fluid densities based on types of collisions and direction for a three-dimensional D3Q27 lattice. (In this case, there are boundary options for cascaded LBE collisions as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying simple Zou-He boundary condition

fD3Q27PFZouHe()

```
int fD3Q27PFZouHe (long tpos,
                   int prop,
                   double * p0,
                   double * uwall)
```

Applies the appropriate Zou-He boundary condition for constant fluid densities based on types of collisions and direction for a three-dimensional D3Q27 lattice. (In this case, there are boundary options for cascaded LBE collisions as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying Zou-He boundary condition

fD3Q27PPSZouHe()

```
int fD3Q27PPSZouHe (double * p,
                    double * force,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18, double * f19, double * f20,
                    double * f21, double * f22, double * f23,
                    double * f24, double * f25, double * f26,
                    double & vel)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions: this routine can also be used for systems with cascaded LBE collisions as the local equilibrium distribution functions for these (with zero tangential velocity) result in the same expressions for missing distribution functions. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site
out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q27PTZouHe()

```
int fD3Q27PTZouHe (long tpos,
                   int prop,
                   double p0,
                   double * uwall)
```

Applies the appropriate Zou-He boundary condition for constant temperature based on direction (planar surface, concave edges and corners) for a three-dimensional D3Q27 lattice.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Temperature for boundary lattice point
in	uwall	Velocity at boundary site determined from applying constant velocity/density boundary condition

fD3Q27TCCZouHe()

```
int fD3Q27TCCZouHe (double p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18, double * f19, double * f20,
                    double * f21, double * f22, double * f23,
                    double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed temperature at a concave corner using the three-dimensional D3Q27 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom-left-back concave corners (TCCTRF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature at concave corner
in	v0	Velocity component at concave corner (x-component for bottom-left-back corner)
in	v1	Velocity component at concave corner (y-component for bottom-left-back corner)
in	v2	Velocity component at concave corner (z-component for bottom-left-back corner)
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site
out	f19	Distribution functions for link 19 at corner lattice site
out	f20	Distribution functions for link 20 at corner lattice site
out	f21	Distribution functions for link 21 at corner lattice site
out	f22	Distribution functions for link 22 at corner lattice site
out	f23	Distribution functions for link 23 at corner lattice site
out	f24	Distribution functions for link 24 at corner lattice site
out	f25	Distribution functions for link 25 at corner lattice site
out	f26	Distribution functions for link 26 at corner lattice site

fD3Q27TCEZouHe()

```
int fD3Q27TCEZouHe (double p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18, double * f19, double * f20,
                    double * f21, double * f22, double * f23,
                    double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed temperature at a concave edge using the three-dimensional D3Q27 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom-left concave edges (TCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature at concave edge
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
in	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
in	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
out	f19	Distribution functions for link 19 at edge lattice site
out	f20	Distribution functions for link 20 at edge lattice site
out	f21	Distribution functions for link 21 at edge lattice site
out	f22	Distribution functions for link 22 at edge lattice site
out	f23	Distribution functions for link 23 at edge lattice site
out	f24	Distribution functions for link 24 at edge lattice site
out	f25	Distribution functions for link 25 at edge lattice site
out	f26	Distribution functions for link 26 at edge lattice site

fD3Q27TPSZouHe()

```
int fD3Q27TPSZouHe (double p,
                    double v0, double v1, double v2,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18, double * f19, double * f20,
                    double * f21, double * f22, double * f23,
                    double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a Zou-He boundary condition for fixed temperature at a planar surface using the three-dimensional D3Q27 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom planar surfaces (TPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature for boundary lattice point
in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site
out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site

fD3Q27VCCCLBESimpleZouHe()

```
int fD3Q27VCCCLBESimpleZouHe (double * p,
                              double v0, double v1, double v2,
                              double * force,
                              double * f0, double * f1, double * f2,
                              double * f3, double * f4, double * f5,
                              double * f6, double * f7, double * f8,
                              double * f9, double * f10, double * f11,
                              double * f12, double * f13, double * f14,
                              double * f15, double * f16, double * f17,
                              double * f18, double * f19, double * f20,
                              double * f21, double * f22, double * f23,
                              double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q27 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values f
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site
out	f19	Distribution functions for link 19 at corner lattice site
out	f20	Distribution functions for link 20 at corner lattice site
out	f21	Distribution functions for link 21 at corner lattice site
out	f22	Distribution functions for link 22 at corner lattice site
out	f23	Distribution functions for link 23 at corner lattice site
out	f24	Distribution functions for link 24 at corner lattice site
out	f25	Distribution functions for link 25 at corner lattice site

Table 5.12 – continued from previous page

out	f26	Distribution functions for link 26 at corner lattice site

fD3Q27VCCCLBEZouHe()

```
int fD3Q27VCCCLBEZouHe (double * p,
                        double v0, double v1, double v2,
                        double * force,
                        double * f0, double * f1, double * f2,
                        double * f3, double * f4, double * f5,
                        double * f6, double * f7, double * f8,
                        double * f9, double * f10, double * f11,
                        double * f12, double * f13, double * f14,
                        double * f15, double * f16, double * f17,
                        double * f18, double * f19, double * f20,
                        double * f21, double * f22, double * f23,
                        double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q27 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values f
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site
out	f19	Distribution functions for link 19 at corner lattice site
out	f20	Distribution functions for link 20 at corner lattice site
out	f21	Distribution functions for link 21 at corner lattice site

Table 5.13 – continued from previous page

out	f22	Distribution functions for link 22 at corner lattice site
out	f23	Distribution functions for link 23 at corner lattice site
out	f24	Distribution functions for link 24 at corner lattice site
out	f25	Distribution functions for link 25 at corner lattice site
out	f26	Distribution functions for link 26 at corner lattice site

fD3Q27VCCSimpleZouHe()

```
int fD3Q27VCCSimpleZouHe (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18, double * f19, double * f20,
                          double * f21, double * f22, double * f23,
                          double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values f
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site

Table 5.14 – continued from previous page

out	f19	Distribution functions for link 19 at corner lattice site
out	f20	Distribution functions for link 20 at corner lattice site
out	f21	Distribution functions for link 21 at corner lattice site
out	f22	Distribution functions for link 22 at corner lattice site
out	f23	Distribution functions for link 23 at corner lattice site
out	f24	Distribution functions for link 24 at corner lattice site
out	f25	Distribution functions for link 25 at corner lattice site
out	f26	Distribution functions for link 26 at corner lattice site

fD3Q27VCCZouHe()

```
int fD3Q27VCCZouHe (double * p,
                    double v0, double v1, double v2,
                    double * force,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18, double * f19, double * f20,
                    double * f21, double * f22, double * f23,
                    double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values f
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site

Table 5.15 – continued from previous page

out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site
out	f19	Distribution functions for link 19 at corner lattice site
out	f20	Distribution functions for link 20 at corner lattice site
out	f21	Distribution functions for link 21 at corner lattice site
out	f22	Distribution functions for link 22 at corner lattice site
out	f23	Distribution functions for link 23 at corner lattice site
out	f24	Distribution functions for link 24 at corner lattice site
out	f25	Distribution functions for link 25 at corner lattice site
out	f26	Distribution functions for link 26 at corner lattice site

fD3Q27VCECLBESimpleZouHe()

```
int fD3Q27VCECLBESimpleZouHe (double * p,
                              double v0, double v1, double v2,
                              double * force,
                              double * f0, double * f1, double * f2,
                              double * f3, double * f4, double * f5,
                              double * f6, double * f7, double * f8,
                              double * f9, double * f10, double * f11,
                              double * f12, double * f13, double * f14,
                              double * f15, double * f16, double * f17,
                              double * f18, double * f19, double * f20,
                              double * f21, double * f22, double * f23,
                              double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q27 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values fo
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
in	f11	Distribution functions for link 11 at edge lattice site

Table 5.16 – continued from previous page

out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
in	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
out	f19	Distribution functions for link 19 at edge lattice site
out	f20	Distribution functions for link 20 at edge lattice site
out	f21	Distribution functions for link 21 at edge lattice site
out	f22	Distribution functions for link 22 at edge lattice site
out	f23	Distribution functions for link 23 at edge lattice site
out	f24	Distribution functions for link 24 at edge lattice site
out	f25	Distribution functions for link 25 at edge lattice site
out	f26	Distribution functions for link 26 at edge lattice site

fD3Q27VCECLBEZouHe()

```
int fD3Q27VCECLBEZouHe (double * p,
                        double v0, double v1, double v2,
                        double * force,
                        double * f0, double * f1, double * f2,
                        double * f3, double * f4, double * f5,
                        double * f6, double * f7, double * f8,
                        double * f9, double * f10, double * f11,
                        double * f12, double * f13, double * f14,
                        double * f15, double * f16, double * f17,
                        double * f18, double * f19, double * f20,
                        double * f21, double * f22, double * f23,
                        double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q27 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values fo
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site

Table 5.17 – continued from previous page

in	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
in	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
in	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
out	f19	Distribution functions for link 19 at edge lattice site
out	f20	Distribution functions for link 20 at edge lattice site
out	f21	Distribution functions for link 21 at edge lattice site
out	f22	Distribution functions for link 22 at edge lattice site
out	f23	Distribution functions for link 23 at edge lattice site
out	f24	Distribution functions for link 24 at edge lattice site
out	f25	Distribution functions for link 25 at edge lattice site
out	f26	Distribution functions for link 26 at edge lattice site

fD3Q27VCESimpleZouHe()

```
int fD3Q27VCESimpleZouHe (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18, double * f19, double * f20,
                          double * f21, double * f22, double * f23,
                          double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values fo
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site

Table 5.18 – continued from previous page

in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
in	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
in	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
out	f19	Distribution functions for link 19 at edge lattice site
out	f20	Distribution functions for link 20 at edge lattice site
out	f21	Distribution functions for link 21 at edge lattice site
out	f22	Distribution functions for link 22 at edge lattice site
out	f23	Distribution functions for link 23 at edge lattice site
out	f24	Distribution functions for link 24 at edge lattice site
out	f25	Distribution functions for link 25 at edge lattice site
out	f26	Distribution functions for link 26 at edge lattice site

fD3Q27VCEZouHe()

```
int fD3Q27VCEZouHe (double * p,
                    double v0, double v1, double v2,
                    double * force,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18, double * f19, double * f20,
                    double * f21, double * f22, double * f23,
                    double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values fo
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site

Table 5.19 – continued from previous page

in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
in	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
in	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
out	f19	Distribution functions for link 19 at edge lattice site
out	f20	Distribution functions for link 20 at edge lattice site
out	f21	Distribution functions for link 21 at edge lattice site
out	f22	Distribution functions for link 22 at edge lattice site
out	f23	Distribution functions for link 23 at edge lattice site
out	f24	Distribution functions for link 24 at edge lattice site
out	f25	Distribution functions for link 25 at edge lattice site
out	f26	Distribution functions for link 26 at edge lattice site

fD3Q27VFSimpleZouHe()

```
int fD3Q27VFSimpleZouHe (long tpos,
                         long rpos,
                         int prop,
                         double * uwall)
```

Applies the appropriate simple Zou-He boundary condition for a constant velocity based on types of collisions and direction for a three-dimensional D3Q27 lattice. (In this case, there are boundary options for cascaded LBE collsions as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	rpos	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site

fD3Q27VFZouHe()

```
int fD3Q27VFZouHe (long tpos,
                   long rpos,
                   int prop,
                   double * uwall)
```

Applies the appropriate Zou-He boundary condition for a constant velocity based on types of collisions and direction for a three-dimensional D3Q27 lattice. (In this case, there are boundary options for cascaded LBE collsions as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	rpos	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site

fD3Q27VPSCLBESimpleZouHe()

```
int fD3Q27VPSCLBESimpleZouHe (double v0, double v1, double v2,
                              double * force,
                              double * f0, double * f1, double * f2,
                              double * f3, double * f4, double * f5,
                              double * f6, double * f7, double * f8,
                              double * f9, double * f10, double * f11,
                              double * f12, double * f13, double * f14,
                              double * f15, double * f16, double * f17,
                              double * f18, double * f19, double * f20,
                              double * f21, double * f22, double * f23,
                              double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q27 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site

continues on next page

Table 5.20 – continued from previous page

out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site

fD3Q27VPSCLBEZouHe()

```
int fD3Q27VPSCLBEZouHe (double v0, double v1, double v2,
                        double * force,
                        double * f0, double * f1, double * f2,
                        double * f3, double * f4, double * f5,
                        double * f6, double * f7, double * f8,
                        double * f9, double * f10, double * f11,
                        double * f12, double * f13, double * f14,
                        double * f15, double * f16, double * f17,
                        double * f18, double * f19, double * f20,
                        double * f21, double * f22, double * f23,
                        double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q27 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site

continues on next page

Table 5.21 – continued from previous page

out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site

fD3Q27VPSSimpleZouHe()

```
int fD3Q27VPSSimpleZouHe (double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18, double * f19, double * f20,
                          double * f21, double * f22, double * f23,
                          double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a simple Zou-He boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site

continues on next page

Table 5.22 – continued from previous page

out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site

fD3Q27VPSZouHe()

```
int fD3Q27VPSZouHe (double v0, double v1, double v2,
                    double * force,
                    double * f0, double * f1, double * f2,
                    double * f3, double * f4, double * f5,
                    double * f6, double * f7, double * f8,
                    double * f9, double * f10, double * f11,
                    double * f12, double * f13, double * f14,
                    double * f15, double * f16, double * f17,
                    double * f18, double * f19, double * f20,
                    double * f21, double * f22, double * f23,
                    double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a Zou-He boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site

continues on next page

Table 5.23 – continued from previous page

out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site

5.26 lbpBOUNDInamuro.cpp

Module for applying Inamuro boundary conditions. (Header file available as lbpBOUNDInamuro.hpp.)

Applies Inamuro [63] boundary conditions at specified lattice points to give fixed fluid velocities or densities, solute concentrations and temperatures. This scheme uses local equilibrium distribution functions for 'missing' distribution functions re-entering the simulation box with enhanced densities ρ' and additional tangential slip velocities $\vec{u}_s$ added on to required fluid velocities (with non-zero values for directions tangential to the boundary and zero for orthogonal directions). Values for these properties can be obtained by using these local equilibrium distribution functions along with the following summations to conserve fluid mass and momentum:

$$\rho = \sum_i f_i,$$

$$\rho u_\alpha = \sum_i f_i e_{i,\alpha}.$$

Rearrangement of these expressions for concave edges in two dimensions or planar surfaces in three dimensions can provide the wall density or orthogonal velocity component, depending on which type of boundary condition is required, as well as the adjusted density and slip velocity to ensure the specified boundary condition can be fulfilled. In the cases of constant solute concentrations and/or temperatures, only the adjusted density ρ' (as an analogue for solute concentration or temperature) is required in local equilibrium distribution functions along with the known fluid velocity. For boundaries other than concave edges in two dimensions or planar surfaces in three dimensions, the same procedure with fewer adjusted properties can be used (e.g. only using the adjusted density for concave corners), although both fluid velocities and densities are required: for constant velocity boundaries, the fluid densities can be sampled from a nearby lattice point, while for constant density boundaries the fluid velocity can be assumed to be zero.

5.26.1 Functions

- `int` *fD2Q9VCEInamuro()*

  Applies Inamuro constant velocity boundary condition to concave edge for D2Q9 lattice.

- `int` *fD2Q9VCCInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave corner for D2Q9 lattice.

- `int` *fD2Q9VCECLBEInamuro()*

  Applies Inamuro constant velocity boundary condition to concave edge for D2Q9 lattice with cascaded LBE collisions.

- `int` *fD2Q9VCCCLBEInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave corner for D2Q9 lattice with cascaded LBE collisions.

- `int` *fD2Q9VCESwiftInamuro()*

  Applies Inamuro constant velocity boundary condition to concave edge for D2Q9 lattice with Swift free-energy interactions.

- `int` *fD2Q9VCCSwiftInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave corner for D2Q9 lattice with Swift free-energy interactions.

- `int` *fD2Q9VFInamuro()*

  Applies constant velocity Inamuro boundary condition to lattice point using D2Q9 lattice scheme.

- `int` *fD2Q9PCEInamuro()*

  Applies Inamuro constant density boundary condition to concave edge for D2Q9 lattice.

- `int` *fD2Q9PCECLBEInamuro()*

  Applies Inamuro constant density boundary condition to concave edge for D2Q9 lattice with cascaded LBE collisions.

- `int` *fD2Q9PCESwiftInamuro()*

  Applies Inamuro constant density boundary condition to concave edge for D2Q9 lattice with Swift free-energy interactions.

- `int` *fD2Q9PFInamuro()*

  Applies constant density Inamuro boundary condition to lattice point using D2Q9 lattice scheme.

- `int` *fD2Q9CCEInamuro()*

  Applies Inamuro constant solute concentration boundary condition to concave edge for D2Q9 lattice.

- `int` *fD2Q9CCCInamuro()*

  Applies Inamuro constant solute concentration boundary condition to concave corner for D2Q9 lattice.

- `int` *fD2Q9PCInamuro()*

  Applies constant solute concentration Inamuro boundary condition to lattice point using D2Q9 lattice scheme.

- `int` *fD2Q9TCEInamuro()*

  Applies Inamuro constant temperature boundary condition to concave edge for D2Q9 lattice.

- `int` *fD2Q9TCCInamuro()*

  Applies Inamuro constant temperature boundary condition to concave corner for D2Q9 lattice.

- `int` *fD2Q9PTInamuro()*

  Applies constant temperature Inamuro boundary condition to lattice point using D2Q9 lattice scheme.

- `int` *fD3Q15VPSInamuro()*

  Applies Inamuro constant velocity boundary condition to planar surface for D3Q15 lattice.

- `int` *fD3Q15VCEInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave edge for D3Q15 lattice.

- `int` *fD3Q15VCCInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave corner for D3Q15 lattice.

- `int` *fD3Q15VPSSwiftInamuro()*

  Applies Inamuro constant velocity boundary condition to planar surface for D3Q15 lattice with Swift free-energy interactions.

- `int` *fD3Q15VCESwiftInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave edge for D3Q15 lattice with Swift free-energy interactions.

- `int` *fD3Q15VCCSwiftInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave corner for D3Q15 lattice with Swift free-energy interactions.

- `int` *fD3Q15VFInamuro()*

  Applies constant velocity Inamuro boundary condition to lattice point using D3Q15 lattice scheme.
- `int` *fD3Q15PPSInamuro()*

  Applies Inamuro constant density boundary condition to planar surface for D3Q15 lattice.
- `int` *fD3Q15PPSSwiftInamuro()*

  Applies Inamuro constant density boundary condition to planar surface for D3Q15 lattice with Swift free-energy interactions.
- `int` *fD3Q15PFInamuro()*

  Applies constant density Inamuro boundary condition to lattice point using D3Q15 lattice scheme.
- `int` *fD3Q15CPSInamuro()*

  Applies Inamuro constant solute concentration boundary condition to planar surface for D3Q15 lattice.
- `int` *fD3Q15CCEInamuro()*

  Applies Inamuro constant solute concentration boundary condition to concave edge for D3Q15 lattice.
- `int` *fD3Q15CCCInamuro()*

  Applies Inamuro constant solute concentration boundary condition to concave corner for D3Q15 lattice.
- `int` *fD3Q15PCInamuro()*

  Applies constant solute concentration Inamuro boundary condition to lattice point using D3Q15 lattice scheme.
- `int` *fD3Q15TPSInamuro()*

  Applies Inamuro constant temperature boundary condition to planar surface for D3Q15 lattice.
- `int` *fD3Q15TCEInamuro()*

  Applies Inamuro constant temperature boundary condition to concave edge for D3Q15 lattice.
- `int` *fD3Q15TCCInamuro()*

  Applies Inamuro constant temperature boundary condition to concave corner for D3Q15 lattice.
- `int` *fD3Q15PTInamuro()*

  Applies constant temperature Inamuro boundary condition to lattice point using D3Q15 lattice scheme.
- `int` *fD3Q19VPSInamuro()*

  Applies Inamuro constant velocity boundary condition to planar surface for D3Q19 lattice.
- `int` *fD3Q19VCEInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave edge for D3Q19 lattice.
- `int` *fD3Q19VCCInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave corner for D3Q19 lattice.
- `int` *fD3Q19VPSCLBEInamuro()*

  Applies Inamuro constant velocity boundary condition to planar surface for D3Q19 lattice with cascaded LBE collisions.
- `int` *fD3Q19VCECLBEInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave edge for D3Q19 lattice with cascaded LBE collisions.
- `int` *fD3Q19VCCCLBEInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave corner for D3Q19 lattice with cascaded LBE collisions.

- `int` *fD3Q19VPSSwiftInamuro()*

  Applies Inamuro constant velocity boundary condition to planar surface for D3Q19 lattice with Swift free-energy interactions.
- `int` *fD3Q19VCESwiftInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave edge for D3Q19 lattice with Swift free-energy interactions.
- `int` *fD3Q19VCCSwiftInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave corner for D3Q19 lattice with Swift free-energy interactions.
- `int` *fD3Q19VFInamuro()*

  Applies constant velocity Inamuro boundary condition to lattice point using D3Q19 lattice scheme.
- `int` *fD3Q19PPSInamuro()*

  Applies Inamuro constant density boundary condition to planar surface for D3Q19 lattice.
- `int` *fD3Q19PPSCLBEInamuro()*

  Applies Inamuro constant density boundary condition to planar surface for D3Q19 lattice with cascaded LBE collisions.
- `int` *fD3Q19PPSSwiftInamuro()*

  Applies Inamuro constant density boundary condition to planar surface for D3Q19 lattice with Swift free-energy interactions.
- `int` *fD3Q19PFInamuro()*

  Applies constant density Inamuro boundary condition to lattice point using D3Q19 lattice scheme.
- `int` *fD3Q19CPSInamuro()*

  Applies Inamuro constant solute concentration boundary condition to planar surface for D3Q19 lattice.
- `int` *fD3Q19CCEInamuro()*

  Applies Inamuro constant solute concentration boundary condition to concave edge for D3Q19 lattice.
- `int` *fD3Q19CCCInamuro()*

  Applies Inamuro constant solute concentration boundary condition to concave corner for D3Q15 lattice.
- `int` *fD3Q19PCInamuro()*

  Applies constant solute concentration Inamuro boundary condition to lattice point using D3Q19 lattice scheme.
- `int` *fD3Q19TPSInamuro()*

  Applies Inamuro constant temperature boundary condition to planar surface for D3Q19 lattice.
- `int` *fD3Q19TCEInamuro()*

  Applies Inamuro constant temperature boundary condition to concave edge for D3Q19 lattice.
- `int` *fD3Q19TCCInamuro()*

  Applies Inamuro constant temperature boundary condition to concave corner for D3Q19 lattice.
- `int` *fD3Q19PTInamuro()*

  Applies constant temperature Inamuro boundary condition to lattice point using D3Q19 lattice scheme.
- `int` *fD3Q27VPSInamuro()*

  Applies Inamuro constant velocity boundary condition to planar surface for D3Q27 lattice.

- `int` *fD3Q27VCEInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave edge for D3Q27 lattice.

- `int` *fD3Q27VCCInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave corner for D3Q27 lattice.

- `int` *fD3Q27VPSCLBEInamuro()*

  Applies Inamuro constant velocity boundary condition to planar surface for D3Q27 lattice with cascaded LBE collisions.

- `int` *fD3Q27VCECLBEInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave edge for D3Q27 lattice with cascaded LBE collisions.

- `int` *fD3Q27VCCCLBEInamuro()*

  Applies Inamuro constant velocity or density boundary condition to concave corner for D3Q27 lattice with cascaded LBE collisions.

- `int` *fD3Q27VFInamuro()*

  Applies constant velocity Inamuro boundary condition to lattice point using D3Q27 lattice scheme.

- `int` *fD3Q27PPSInamuro()*

  Applies Inamuro constant density boundary condition to planar surface for D3Q27 lattice.

- `int` *fD3Q27PPSCLBEInamuro()*

  Applies Inamuro constant density boundary condition to planar surface for D3Q27 lattice with cascaded LBE collisions.

- `int` *fD3Q27PFInamuro()*

  Applies constant density Inamuro boundary condition to lattice point using D3Q27 lattice scheme.

- `int` *fD3Q27CPSInamuro()*

  Applies Inamuro constant solute concentration boundary condition to planar surface for D3Q27 lattice.

- `int` *fD3Q27CCEInamuro()*

  Applies Inamuro constant solute concentration boundary condition to concave edge for D3Q27 lattice.

- `int` *fD3Q27CCCInamuro()*

  Applies Inamuro constant solute concentration boundary condition to concave corner for D3Q27 lattice.

- `int` *fD3Q27PCInamuro()*

  Applies constant solute concentration Inamuro boundary condition to lattice point using D3Q27 lattice scheme.

- `int` *fD3Q27TPSInamuro()*

  Applies Inamuro constant temperature boundary condition to planar surface for D3Q27 lattice.

- `int` *fD3Q27TCEInamuro()*

  Applies Inamuro constant temperature boundary condition to concave edge for D3Q27 lattice.

- `int` *fD3Q27TCCInamuro()*

  Applies Inamuro constant temperature boundary condition to concave corner for D3Q27 lattice.

- `int` *fD3Q27PTInamuro()*

  Applies constant temperature Inamuro boundary condition to lattice point using D3Q27 lattice scheme.

5.26.2 Function Documentation

fD2Q9CCCInamuro()

```
int fD2Q9CCCInamuro (double * p,
                     double v0, double v1,
                     double * f0, double * f1, double * f2,
                     double * f3, double * f4, double * f5,
                     double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed solute concentrations at a concave corner using the two-dimensional D2Q9 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for the bottom-left concave corner (CCCTRF) but can be used for any concave corner by selecting different distribution functions.

Parameters

in	p	Solute concentrations for boundary lattice point
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site

fD2Q9CCEInamuro()

```
int fD2Q9CCEInamuro (double * p,
                     double v0, double v1,
                     double * f0, double * f1, double * f2,
                     double * f3, double * f4, double * f5,
                     double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed solute concentrations at a concave edge using the two-dimensional D2Q9 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom concave edges (CCETF) but can be used for any concave edge by selecting different distribution functions.

Parameters

in	p	Solute concentrations for boundary lattice point
in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site

fD2Q9PCECLBEInamuro()

```
int fD2Q9PCECLBEInamuro (double * p,
                         double * force,
                         double * f0, double * f1, double * f2,
                         double * f3, double * f4, double * f5,
                         double * f6, double * f7, double * f8,
                         double & vel)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed fluid densities at a concave edge using the two-dimensional D2Q9 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom concave edges (PCETF) but can be used for any concave edge by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD2Q9PCEInamuro()

```
int fD2Q9PCEInamuro (double * p,
                     double * force,
                     double * f0, double * f1, double * f2,
                     double * f3, double * f4, double * f5,
                     double * f6, double * f7, double * f8,
                     double & vel)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed fluid densities at a concave edge using the two-dimensional D2Q9 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom concave edges (PCETF) but can be used for any concave edge by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD2Q9PCESwiftInamuro()

```
int fD2Q9PCESwiftInamuro (double * p,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double drdx, double drdy,
                          double dpdx, double dpdy,
                          double nabr, double nabp,
                          double * omega,
                          double T,
                          double & vel)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed fluid densities at a concave edge using the two-dimensional D2Q9 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the orthogonal velocity component: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). (The tangential velocity component is assumed equal to zero.) The expressions in this subroutine are for bottom concave edges (PCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for density/concentration gradients (which may be swapped around).

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD2Q9PCInamuro()

```
int fD2Q9PCInamuro (long tpos,
                    int prop,
                    double * p0,
                    double * uwall)
```

Applies the appropriate Inamuro boundary condition for constant solute concentrations based on direction (concave edges and corners) for a two-dimensional D2Q9 lattice.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Solute concentrations for boundary lattice point
in	uwall	Velocity at boundary site determined from applying constant velocity/density boundary condition

fD2Q9PFInamuro()

```
int fD2Q9PFInamuro (long tpos,
                    int prop,
                    double * p0,
                    double * uwall,
                    double T)
```

Applies the appropriate Inamuro boundary condition for constant fluid densities based on types of collisions, interactions and direction for a two-dimensional D2Q9 lattice. (In this case, there are boundary options for cascaded LBE collisions and Swift free-energy interactions, as well as concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying Inamuro boundary condition
in	T	Temperature at boundary grid point

fD2Q9PTInamuro()

```
int fD2Q9PTInamuro (long tpos,
                    int prop,
                    double p0,
                    double * uwall)
```

Applies the appropriate Inamuro boundary condition for a constant temperature based on direction (concave edges and corners) for a two-dimensional D2Q9 lattice.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Solute concentrations for boundary lattice point
in	uwall	Velocity at boundary site determined from applying constant velocity/density boundary condition

fD2Q9TCCInamuro()

```
int fD2Q9TCCInamuro (double p,
                     double v0, double v1,
                     double * f0, double * f1, double * f2,
                     double * f3, double * f4, double * f5,
                     double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed temperature at a concave corner using the two-dimensional D2Q9 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for the bottom-left concave corner (TCCTRF) but can be used for any concave corner by selecting different distribution functions.

Parameters

in	p	Temperature for boundary lattice point
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site

fD2Q9TCEInamuro()

```
int fD2Q9TCEInamuro (double p,
                     double v0, double v1,
                     double * f0, double * f1, double * f2,
                     double * f3, double * f4, double * f5,
                     double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed temperature at a concave edge using the two-dimensional D2Q9 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom concave edges (TCETF) but can be used for any concave edge by selecting different distribution functions.

Parameters

in	p	Temperature for boundary lattice point
in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site

fD2Q9VCCCLBEInamuro()

```
int fD2Q9VCCCLBEInamuro (double * p,
                         double v0, double v1,
                         double * force,
                         double * f0, double * f1, double * f2,
                         double * f3, double * f4, double * f5,
                         double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave corner using the two-dimensional D2Q9 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site

fD2Q9VCCInamuro()

```
int fD2Q9VCCInamuro (double * p,
                     double v0, double v1,
                     double * force,
                     double * f0, double * f1, double * f2,
                     double * f3, double * f4, double * f5,
                     double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave corner using the two-dimensional D2Q9 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site

fD2Q9VCCSwiftInamuro()

```
int fD2Q9VCCSwiftInamuro (double * p,
                          double v0, double v1,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double drdx, double drdy,
                          double dpdx, double dpdy,
                          double nabr, double nabp,
                          double * omega,
                          double T)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave corner using the two-dimensional D2Q9 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the sum of both velocity components: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). The expressions in this subroutine are for bottom-left concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD2Q9VCECLBEInamuro()

```
int fD2Q9VCECLBEInamuro (double v0, double v1,
                         double * force,
                         double * f0, double * f1, double * f2,
                         double * f3, double * f4, double * f5,
                         double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity at a concave edge using the two-dimensional D2Q9 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom concave edges (VCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site

fD2Q9VCEInamuro()

```
int fD2Q9VCEInamuro (double v0, double v1,
                     double * force,
                     double * f0, double * f1, double * f2,
                     double * f3, double * f4, double * f5,
                     double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity at a concave edge using the two-dimensional D2Q9 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom concave edges (VCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site

fD2Q9VCESwiftInamuro()

```
int fD2Q9VCESwiftInamuro (double v0, double v1,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double drdx, double drdy,
                          double dpdx, double dpdy,
                          double nabr, double nabp,
                          double * omega,
                          double T)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity at a concave edge using the two-dimensional D2Q9 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the orthogonal velocity component: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). The expressions in this subroutine are for bottom concave edges (VCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around).

Parameters

in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD2Q9VFInamuro()

```
int fD2Q9VFInamuro (long tpos,
                    long tpos1,
                    int prop,
                    double * uwall,
                    double dx,
                    double dy,
                    double T)
```

Applies the appropriate Inamuro boundary condition for a constant velocity based on types of collisions, interactions and direction for a two-dimensional D2Q9 lattice. (In this case, there are boundary options for cascaded LBE collisions and Swift free-energy interactions, as well as concave edges and corners.) For corners with Swift free-energy interactions, the vector between the boundary lattice point and sampling point for densities can be specified to correct fluid density/concentration using gradients of those properties evalulated at the boundary point.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	tpos1	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site
in	dx	Vector to move from current lattice site (x-component)
in	dy	Vector to move from current lattice site (y-component)
in	T	Temperature at boundary grid point

fD3Q15CCCInamuro()

```
int fD3Q15CCCInamuro (double * p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed solute concentrations at a concave corner using the three-dimensional D3Q15 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom-left-back concave corners (CCCTRF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations at concave corner
in	v0	Velocity component at concave corner (x-component for bottom-left-back corner)
in	v1	Velocity component at concave corner (y-component for bottom-left-back corner)
in	v2	Velocity component at concave corner (z-component for bottom-left-back corner)
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site

fD3Q15CCEInamuro()

```
int fD3Q15CCEInamuro (double * p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed solute concentrations at a concave edge using the three-dimensional D3Q15 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom-left concave edges (CCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations at concave edge
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site

fD3Q15CPSInamuro()

```
int fD3Q15CPSInamuro (double * p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed solute concentrations at a planar surface using the three-dimensional D3Q15 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom planar surfaces (CPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations for boundary lattice point
in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site

fD3Q15PCInamuro()

```
int fD3Q15PCInamuro (long tpos,
                     int prop,
                     double * p0,
                     double * uwall)
```

Applies the appropriate Inamuro boundary condition for constant solute concentrations based on direction (planar surface, concave edges and corners) for a three-dimensional D3Q15 lattice.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Solute concentrations for boundary lattice point
in	uwall	Velocity at boundary site determined from applying constant velocity/density boundary condition

fD3Q15PFInamuro()

```
int fD3Q15PFInamuro (long tpos,
                     int prop,
                     double * p0,
                     double * uwall,
                     double T)
```

Applies the appropriate Inamuro boundary condition for constant fluid densities based on types of collisions, interactions and direction for a three-dimensional D3Q15 lattice. (In this case, there are boundary options for Swift free-energy interactions, as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying Inamuro boundary condition
in	T	Temperature at boundary grid point

fD3Q15PPSInamuro()

```
int fD3Q15PPSInamuro (double * p,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double & vel)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q15PPSSwiftInamuro()

```
int fD3Q15PPSSwiftInamuro (double * p,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T,
                           double & vel)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q15 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the orthogonal velocity component: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). (The tangential velocity component is assumed equal to zero.) The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q15PTInamuro()

```
int fD3Q15PTInamuro (long tpos,
                     int prop,
                     double p0,
                     double * uwall)
```

Applies the appropriate Inamuro boundary condition for constant temperature based on direction (planar surface, concave edges and corners) for a three-dimensional D3Q15 lattice.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Temperature for boundary lattice point
in	uwall	Velocity at boundary site determined from applying constant velocity/density boundary condition

fD3Q15TCCInamuro()

```
int fD3Q15TCCInamuro (double p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed temperature at a concave corner using the three-dimensional D3Q15 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom-left-back concave corners (TCCTRF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature at concave corner
in	v0	Velocity component at concave corner (x-component for bottom-left-back corner)
in	v1	Velocity component at concave corner (y-component for bottom-left-back corner)
in	v2	Velocity component at concave corner (z-component for bottom-left-back corner)
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site

fD3Q15TCEInamuro()

```
int fD3Q15TCEInamuro (double p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed temperature at a concave edge using the three-dimensional D3Q15 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom-left concave edges (TCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature at concave edge
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site

fD3Q15TPSInamuro()

```
int fD3Q15TPSInamuro (double p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed temperature at a planar surface using the three-dimensional D3Q15 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom planar surfaces (TPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature for boundary lattice point
in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site

fD3Q15VCCInamuro()

```
int fD3Q15VCCInamuro (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site

fD3Q15VCCSwiftInamuro()

```
int fD3Q15VCCSwiftInamuro (double * p,
                           double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave cprmer using the three-dimensional D3Q15 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the sum of both velocity components: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
in	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q15VCEInamuro()

```
int fD3Q15VCEInamuro (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site

fD3Q15VCESwiftInamuro()

```
int fD3Q15VCESwiftInamuro (double * p,
                           double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q15 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the sum of both velocity components: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q15VFInamuro()

```
int fD3Q15VFInamuro (long tpos,
                     long rpos,
                     int prop,
                     double * uwall,
                     double dx,
                     double dy,
                     double dz,
                     double T)
```

Applies the appropriate Inamuro boundary condition for a constant velocity based on types of interactions and direction for a three-dimensional D3Q15 lattice. (In this case, there are boundary options for Swift free-energy interactions, as well as planar surfaces, concave edges and corners.) For edges and corners with Swift free-energy interactions, the vector between the boundary lattice point and sampling point for densities can be specified to correct fluid density/concentration using gradients of those properties evalulated at the boundary point.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	rpos	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site
in	dx	Vector to move from current lattice site (x-component)
in	dy	Vector to move from current lattice site (y-component)
in	dz	Vector to move from current lattice site (z-component)
in	T	Temperature at boundary grid point

fD3Q15VPSInamuro()

```
int fD3Q15VPSInamuro (double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site

fD3Q15VPSSwiftInamuro()

```
int fD3Q15VPSSwiftInamuro (double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q15 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the orthogonal velocity component: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q19CCCInamuro()

```
int fD3Q19CCCInamuro (double * p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed solute concentrations at a concave corner using the three-dimensional D3Q19 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom-left-back concave corners (CCCTRF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations at concave corner
in	v0	Velocity component at concave corner (x-component for bottom-left-back corner)
in	v1	Velocity component at concave corner (y-component for bottom-left-back corner)
in	v2	Velocity component at concave corner (z-component for bottom-left-back corner)
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site

fD3Q19CCEInamuro()

```
int fD3Q19CCEInamuro (double * p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed solute concentrations at a concave edge using the three-dimensional D3Q19 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom-left concave edges (CCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations at concave edge
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site

fD3Q19CPSInamuro()

```
int fD3Q19CPSInamuro (double * p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed solute concentrations at a planar surface using the three-dimensional D3Q19 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom planar surfaces (CPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations for boundary lattice point
in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site

fD3Q19PCInamuro()

```
int fD3Q19PCInamuro (long tpos,
                     int prop,
                     double * p0,
                     double * uwall)
```

Applies the appropriate Inamuro boundary condition for constant solute concentrations based on direction (planar surface, concave edges and corners) for a three-dimensional D3Q19 lattice.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Solute concentrations for boundary lattice point
in	uwall	Velocity at boundary site determined from applying constant velocity/density boundary condition

fD3Q19PFInamuro()

```
int fD3Q19PFInamuro (long tpos,
                     int prop,
                     double * p0,
                     double * uwall,
                     double T)
```

Applies the appropriate Inamuro boundary condition for constant fluid densities based on types of collisions, interactions and direction for a three-dimensional D3Q19 lattice. (In this case, there are boundary options for cascaded LBE collisions, Swift free-energy interactions, as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying Inamuro boundary condition
in	T	Temperature at boundary grid point

fD3Q19PPSCLBEInamuro()

```
int fD3Q19PPSCLBEInamuro (double * p,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18,
                          double & vel)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q19 lattice and cascaded LBE (CLBE) collsions. This routine can only be used for mildly compressible using the extended local equilibrium distribution functions obtained from CLBE collisions.The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q19PPSInamuro()

```
int fD3Q19PPSInamuro (double * p,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18,
                      double & vel)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q19PPSSwiftInamuro()

```
int fD3Q19PPSSwiftInamuro (double * p,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double * f15, double * f16, double * f17,
                           double * f18,
```

(continues on next page)

(continued from previous page)

```
                              double drdx, double drdy, double drdz,
                              double dpdx, double dpdy, double dpdz,
                              double nabr, double nabp,
                              double * omega,
                              double T,
                              double & vel)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q19 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the orthogonal velocity component: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). (The tangential velocity component is assumed equal to zero.) The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q19PTInamuro()

```
int fD3Q19PTInamuro (long tpos,
                     int prop,
                     double p0,
                     double * uwall)
```

Applies the appropriate Inamuro boundary condition for constant temperature based on direction (planar surface, concave edges and corners) for a three-dimensional D3Q19 lattice.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Temperature for boundary lattice point
in	uwall	Velocity at boundary site determined from applying constant velocity/density boundary condition

fD3Q19TCCInamuro()

```
int fD3Q19TCCInamuro (double p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed temperature at a concave corner using the three-dimensional D3Q19 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom-left-back concave corners (TCCTRF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature at concave corner
in	v0	Velocity component at concave corner (x-component for bottom-left-back corner)
in	v1	Velocity component at concave corner (y-component for bottom-left-back corner)
in	v2	Velocity component at concave corner (z-component for bottom-left-back corner)
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site

fD3Q19TCEInamuro()

```
int fD3Q19TCEInamuro (double p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed temperature at a concave edge using the three-dimensional D3Q19 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom-left concave edges (TCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature at concave edge
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site

fD3Q19TPSInamuro()

```
int fD3Q19TPSInamuro (double p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed temperature at a planar surface using the three-dimensional D3Q19 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom planar surfaces (TPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature for boundary lattice point
in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site

fD3Q19VCCCLBEInamuro()

```
int fD3Q19VCCCLBEInamuro (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q19 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site

fD3Q19VCCInamuro()

```
int fD3Q19VCCInamuro (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site

fD3Q19VCCSwiftInamuro()

```
int fD3Q19VCCSwiftInamuro (double * p,
                           double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double * f15, double * f16, double * f17,
                           double * f18,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q19 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the sum of both velocity components: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q19VCECLBEInamuro()

```
int fD3Q19VCECLBEInamuro (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q19 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site

fD3Q19VCEInamuro()

```
int fD3Q19VCEInamuro (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site

fD3Q19VCESwiftInamuro()

```
int fD3Q19VCESwiftInamuro (double * p,
                           double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double * f15, double * f16, double * f17,
                           double * f18,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q19 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the sum of both velocity components: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values f
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q19VFInamuro()

```
int fD3Q19VFInamuro (long tpos,
                     long rpos,
                     int prop,
                     double * uwall,
                     double dx,
                     double dy,
                     double dz,
                     double T)
```

Applies the appropriate Inamuro boundary condition for a constant velocity based on types of collisions, interactions and direction for a three-dimensional D3Q19 lattice. (In this case, there are boundary options for cascaded LBE collsions, Swift free-energy interactions, as well as planar surfaces, concave edges and corners.) For edges and corners with Swift free-energy interactions, the vector between the boundary lattice point and sampling point for densities can be specified to correct fluid density/concentration using gradients of those properties evalulated at the boundary point.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	rpos	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site
in	dx	Vector to move from current lattice site (x-component)
in	dy	Vector to move from current lattice site (y-component)
in	dz	Vector to move from current lattice site (z-component)
in	T	Temperature at boundary grid point

fD3Q19VPSCLBEInamuro()

```
int fD3Q19VPSCLBEInamuro (double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q19 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site

fD3Q19VPSInamuro()

```
int fD3Q19VPSInamuro (double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site

fD3Q19VPSSwiftInamuro()

```
int fD3Q19VPSSwiftInamuro (double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double * f15, double * f16, double * f17,
                           double * f18,
                           double drdx, double drdy, double drdz,
```

(continues on next page)

(continued from previous page)

```
                                        double dpdx, double dpdy, double dpdz,
                                        double nabr, double nabp,
                                        double * omega,
                                        double T)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q19 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the orthogonal velocity component: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q27CCCInamuro()

```
int fD3Q27CCCInamuro (double * p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed solute concentrations at a concave corner using the three-dimensional D3Q27 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom-left-back concave corners (CCCTRF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations at concave corner
in	v0	Velocity component at concave corner (x-component for bottom-left-back corner)
in	v1	Velocity component at concave corner (y-component for bottom-left-back corner)
in	v2	Velocity component at concave corner (z-component for bottom-left-back corner)
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site
out	f19	Distribution functions for link 19 at corner lattice site
out	f20	Distribution functions for link 20 at corner lattice site
out	f21	Distribution functions for link 21 at corner lattice site
out	f22	Distribution functions for link 22 at corner lattice site
out	f23	Distribution functions for link 23 at corner lattice site
out	f24	Distribution functions for link 24 at corner lattice site
out	f25	Distribution functions for link 25 at corner lattice site
out	f26	Distribution functions for link 26 at corner lattice site

fD3Q27CCEInamuro()

```
int fD3Q27CCEInamuro (double * p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed solute concentrations at a concave edge using the three-dimensional D3Q27 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom-left concave edges (CCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations at concave edge
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
in	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
in	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
out	f19	Distribution functions for link 19 at edge lattice site
out	f20	Distribution functions for link 20 at edge lattice site
out	f21	Distribution functions for link 21 at edge lattice site
out	f22	Distribution functions for link 22 at edge lattice site
out	f23	Distribution functions for link 23 at edge lattice site
out	f24	Distribution functions for link 24 at edge lattice site
out	f25	Distribution functions for link 25 at edge lattice site
out	f26	Distribution functions for link 26 at edge lattice site

fD3Q27CPSInamuro()

```
int fD3Q27CPSInamuro (double * p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed solute concentrations at a planar surface using the three-dimensional D3Q27 lattice. This routine uses the simplified local equilibrium distribution functions for diffusive systems to represent solutes with concentration analogous to density. The expressions in this subroutine are for bottom planar surfaces (CPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Solute concentrations for boundary lattice point
in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site
out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site

fD3Q27PCInamuro()

```
int fD3Q27PCInamuro (long tpos,
                     int prop,
                     double * p0,
                     double * uwall)
```

Applies the appropriate Inamuro boundary condition for constant solute concentrations based on direction (planar surface, concave edges and corners) for a three-dimensional D3Q27 lattice.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Solute concentrations for boundary lattice point
in	uwall	Velocity at boundary site determined from applying constant velocity/density boundary condition

fD3Q27PFInamuro()

```
int fD3Q27PFInamuro (long tpos,
                     int prop,
                     double * p0,
                     double * uwall)
```

Applies the appropriate Inamuro boundary condition for constant fluid densities based on types of collisions and direction for a three-dimensional D3Q27 lattice. (In this case, there are boundary options for cascaded LBE collisions as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying Inamuro boundary condition

fD3Q27PPSCLBEInamuro()

```
int fD3Q27PPSCLBEInamuro (double * p,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18, double * f19, double * f20,
                          double * f21, double * f22, double * f23,
                          double * f24, double * f25, double * f26,
                          double & vel)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q27 lattice and cascaded LBE (CLBE) collsions. This routine can only be used for mildly compressible using the extended local equilibrium distribution functions obtained from CLBE collisions.The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site
out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q27PPSInamuro()

```
int fD3Q27PPSInamuro (double * p,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26,
                      double & vel)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site
out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q27PTInamuro()

```
int fD3Q27PTInamuro (long tpos,
                     int prop,
                     double p0,
                     double * uwall)
```

Applies the appropriate Inamuro boundary condition for constant temperature based on direction (planar surface, concave edges and corners) for a three-dimensional D3Q27 lattice.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Temperature for boundary lattice point
in	uwall	Velocity at boundary site determined from applying constant velocity/density boundary condition

fD3Q27TCCInamuro()

```
int fD3Q27TCCInamuro (double p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed temperature at a concave corner using the three-dimensional D3Q27 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom-left-back concave corners (TCCTRF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature at concave corner
in	v0	Velocity component at concave corner (x-component for bottom-left-back corner)
in	v1	Velocity component at concave corner (y-component for bottom-left-back corner)
in	v2	Velocity component at concave corner (z-component for bottom-left-back corner)
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site
out	f19	Distribution functions for link 19 at corner lattice site
out	f20	Distribution functions for link 20 at corner lattice site
out	f21	Distribution functions for link 21 at corner lattice site
out	f22	Distribution functions for link 22 at corner lattice site
out	f23	Distribution functions for link 23 at corner lattice site
out	f24	Distribution functions for link 24 at corner lattice site
out	f25	Distribution functions for link 25 at corner lattice site
out	f26	Distribution functions for link 26 at corner lattice site

fD3Q27TCEInamuro()

```
int fD3Q27TCEInamuro (double p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed temperature at a concave edge using the three-dimensional D3Q27 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom-left concave edges (TCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature at concave edge
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
in	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
in	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
out	f19	Distribution functions for link 19 at edge lattice site
out	f20	Distribution functions for link 20 at edge lattice site
out	f21	Distribution functions for link 21 at edge lattice site
out	f22	Distribution functions for link 22 at edge lattice site
out	f23	Distribution functions for link 23 at edge lattice site
out	f24	Distribution functions for link 24 at edge lattice site
out	f25	Distribution functions for link 25 at edge lattice site
out	f26	Distribution functions for link 26 at edge lattice site

fD3Q27TPSInamuro()

```
int fD3Q27TPSInamuro (double p,
                      double v0, double v1, double v2,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete an Inamuro boundary condition for fixed temperature at a planar surface using the three-dimensional D3Q27 lattice. This routine uses the simplified local equilibrium distribution function for diffusive systems to represent heat transfers with temperature analogous to density. The expressions in this subroutine are for bottom planar surfaces (TPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	p	Temperature for boundary lattice point
in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site
out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site

fD3Q27VCCCLBEInamuro()

```
int fD3Q27VCCCLBEInamuro (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18, double * f19, double * f20,
                          double * f21, double * f22, double * f23,
                          double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q27 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values f
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site
out	f19	Distribution functions for link 19 at corner lattice site
out	f20	Distribution functions for link 20 at corner lattice site
out	f21	Distribution functions for link 21 at corner lattice site
out	f22	Distribution functions for link 22 at corner lattice site
out	f23	Distribution functions for link 23 at corner lattice site
out	f24	Distribution functions for link 24 at corner lattice site
out	f25	Distribution functions for link 25 at corner lattice site

Table 5.34 – continued from previous page

out	f26	Distribution functions for link 26 at corner lattice site

fD3Q27VCCInamuro()

```
int fD3Q27VCCInamuro (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values f
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site
out	f19	Distribution functions for link 19 at corner lattice site
out	f20	Distribution functions for link 20 at corner lattice site
out	f21	Distribution functions for link 21 at corner lattice site
out	f22	Distribution functions for link 22 at corner lattice site

Table 5.35 – continued from previous page

out	f23	Distribution functions for link 23 at corner lattice site
out	f24	Distribution functions for link 24 at corner lattice site
out	f25	Distribution functions for link 25 at corner lattice site
out	f26	Distribution functions for link 26 at corner lattice site

fD3Q27VCECLBEInamuro()

```
int fD3Q27VCECLBEInamuro (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18, double * f19, double * f20,
                          double * f21, double * f22, double * f23,
                          double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q27 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values fo
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
in	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
in	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
out	f19	Distribution functions for link 19 at edge lattice site

Table 5.36 – continued from previous page

out	f20	Distribution functions for link 20 at edge lattice site
out	f21	Distribution functions for link 21 at edge lattice site
out	f22	Distribution functions for link 22 at edge lattice site
out	f23	Distribution functions for link 23 at edge lattice site
out	f24	Distribution functions for link 24 at edge lattice site
out	f25	Distribution functions for link 25 at edge lattice site
out	f26	Distribution functions for link 26 at edge lattice site

fD3Q27VCEInamuro()

```
int fD3Q27VCEInamuro (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values fo
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
in	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
in	f16	Distribution functions for link 16 at edge lattice site

Table 5.37 – continued from previous page

out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
out	f19	Distribution functions for link 19 at edge lattice site
out	f20	Distribution functions for link 20 at edge lattice site
out	f21	Distribution functions for link 21 at edge lattice site
out	f22	Distribution functions for link 22 at edge lattice site
out	f23	Distribution functions for link 23 at edge lattice site
out	f24	Distribution functions for link 24 at edge lattice site
out	f25	Distribution functions for link 25 at edge lattice site
out	f26	Distribution functions for link 26 at edge lattice site

fD3Q27VFInamuro()

```
int fD3Q27VFInamuro (long tpos,
                     long rpos,
                     int prop,
                     double * uwall)
```

Applies the appropriate Inamuro boundary condition for a constant velocity based on types of collisions and direction for a three-dimensional D3Q27 lattice. (In this case, there are boundary options for cascaded LBE collsions as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	rpos	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site

fD3Q27VPSCLBEInamuro()

```
int fD3Q27VPSCLBEInamuro (double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18, double * f19, double * f20,
                          double * f21, double * f22, double * f23,
                          double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q27 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)

continues on next page

Table 5.38 – continued from previous page

in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site
out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site

fD3Q27VPSInamuro()

```
int fD3Q27VPSInamuro (double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete an Inamuro boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)

continues on next page

Table 5.39 – continued from previous page

in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site
out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site

5.27 lbpBOUNDRegular.cpp

Module for applying regularised boundary conditions.

Applies regularised [74] boundary conditions at specified lattice points to give fixed fluid velocities or densities. This scheme calculates non-equilibrium momentum flux tensors that are then used to calculate replacement distribution functions. The non-equilibrium momentum flux tensors can be calculated using original values for distribution functions:

$$\Pi_{\alpha\beta}^{neq} = \Pi_{\alpha\beta} - \Pi_{\alpha\beta}^{eq} = \sum_i e_{i,\alpha} e_{i,\beta} \left(f_i - f_i^{eq}\right).$$

Any unknown distribution functions are obtained by using reflection of non-equilibrium contributions (similar to Zou-He boundary conditions):

$$f_i - f_i^{eq} = f_j - f_j^{eq}.$$

The replacement distribution functions are constructed from local equilibrium distribution values and additional terms dependent on the non-equilibrium momentum flux tensor:

$$f_i = f_i^{eq}\left(\rho_w, \vec{u}_w\right) + \frac{w_i}{2c_s^2}\left(\hat{e}_i\hat{e}_i - c_s^2\mathbf{I}\right) : \mathbf{\Pi}^{neq}$$

For boundaries other than concave edges in two dimensions or planar surfaces in three dimensions where ‘buried’ links that neither enter nor leave the simulation box are included, all non-buried (‘active’) links re-entering the system are determined using non-equilibrium reflection and the buried ones are obtained by ensuring the overall density and momentum are correct, as obtained using the following summations:

$$\rho = \sum_i f_i,$$

$$\rho u_\alpha = \sum_i f_i e_{i,\alpha}.$$

In cases where there are more unknown buried links than density and momentum equations, each buried link is expressed as a combination of terms for these three or four components, which are solved to give the correct density and momentum at the lattice point.

5.27.1 Functions

- `int` *fD2Q9VCERegular()*

  Applies regularised constant velocity boundary condition to concave edge for D2Q9 lattice.

- `int` *fD2Q9VCCRegular()*

  Applies regularised constant velocity or density boundary condition to concave corner for D2Q9 lattice.

- `int` *fD2Q9VCECLBERegular()*

  Applies regularised constant velocity boundary condition to concave edge for D2Q9 lattice with cascaded LBE collisions.

- `int` *fD2Q9VCCCLBERegular()*

  Applies regularised constant velocity or density boundary condition to concave corner for D2Q9 lattice with cascaded LBE collisions.

- `int` *fD2Q9VCESwiftRegular()*

  Applies regularised constant velocity boundary condition to concave edge for D2Q9 lattice with Swift free-energy interactions.

- `int` *fD2Q9VCCSwiftRegular()*

  Applies regularised constant velocity or density boundary condition to concave corner for D2Q9 lattice with Swift free-energy interactions.

- `int` *fD2Q9VFRegular()*

  Applies constant velocity regularised boundary condition to lattice point using D2Q9 lattice scheme.

- `int` *fD2Q9PCERegular()*

  Applies regularised constant density boundary condition to concave edge for D2Q9 lattice.

- `int` *fD2Q9PCESwiftRegular()*

  Applies regularised constant density boundary condition to concave edge for D2Q9 lattice with Swift free-energy interactions.

- `int` *fD2Q9PFRegular()*

  Applies constant density regularised boundary condition to lattice point using D2Q9 lattice scheme.

- `int` *fD3Q15VPSRegular()*

  Applies regularised constant velocity boundary condition to planar surface for D3Q15 lattice.

- `int` *fD3Q15VCERegular()*

  Applies regularised constant velocity or density boundary condition to concave edge for D3Q15 lattice.

- `int` *fD3Q15VCCRegular()*

  Applies regularised constant velocity or density boundary condition to concave corner for D3Q15 lattice.

- `int` *fD3Q15VPSSwiftRegular()*

  Applies regularised constant velocity boundary condition to planar surface for D3Q15 lattice with Swift free-energy interactions.

- `int` *fD3Q15VCESwiftRegular()*

  Applies regularised constant velocity or density boundary condition to concave edge for D3Q15 lattice with Swift free-energy interactions.

- `int` *fD3Q15VCCSwiftRegular()*

  Applies regularised constant velocity or density boundary condition to concave corner for D3Q15 lattice with Swift free-energy interactions.

- `int` *fD3Q15VFRegular()*

  Applies constant velocity regularised boundary condition to lattice point using D3Q15 lattice scheme.

- `int` *fD3Q15PPSRegular()*

  Applies regularised constant density boundary condition to planar surface for D3Q15 lattice.

- `int` *fD3Q15PPSSwiftRegular()*

  Applies regularised constant density boundary condition to planar surface for D3Q15 lattice with Swift free-energy interactions.

- `int` *fD3Q15PFRegular()*

  Applies constant density regularised boundary condition to lattice point using D3Q15 lattice scheme.

- `int` *fD3Q19VPSRegular()*

  Applies regularised constant velocity boundary condition to planar surface for D3Q19 lattice.

- `int` *fD3Q19VCERegular()*

  Applies regularised constant velocity or density boundary condition to concave edge for D3Q19 lattice.

- `int` *fD3Q19VCCRegular()*

  Applies regularised constant velocity or density boundary condition to concave corner for D3Q19 lattice.

- `int` *fD3Q19VPSCLBERegular()*

  Applies regularised constant velocity boundary condition to planar surface for D3Q19 lattice with cascaded LBE collisions.

- `int` *fD3Q19VCECLBERegular()*

  Applies regularised constant velocity or density boundary condition to concave edge for D3Q19 lattice with cascaded LBE collisions.

- `int` *fD3Q19VCCCLBERegular()*

  Applies regularised constant velocity or density boundary condition to concave corner for D3Q19 lattice with cascaded LBE collisions.

- `int` *fD3Q19VPSSwiftRegular()*

  Applies regularised constant velocity boundary condition to planar surface for D3Q19 lattice with Swift free-energy interactions.

- `int` *fD3Q19VCESwiftRegular()*

  Applies regularised constant velocity or density boundary condition to concave edge for D3Q19 lattice with Swift free-energy interactions.

- `int` *fD3Q19VCCSwiftRegular()*

  Applies regularised constant velocity or density boundary condition to concave corner for D3Q19 lattice with Swift free-energy interactions.

- `int` *fD3Q19VFRegular()*

  Applies constant velocity regularised boundary condition to lattice point using D3Q19 lattice scheme.

- `int` *fD3Q19PPSRegular()*

  Applies regularised constant density boundary condition to planar surface for D3Q19 lattice.

- `int` *fD3Q19PPSSwiftRegular()*

  Applies regularised constant density boundary condition to planar surface for D3Q19 lattice with Swift free-energy interactions.

- `int` *fD3Q19PFRegular()*

  Applies constant density regularised boundary condition to lattice point using D3Q19 lattice scheme.

- `int` *fD3Q27VPSRegular()*

  Applies regularised constant velocity boundary condition to planar surface for D3Q27 lattice.

- `int` *fD3Q27VCERegular()*

  Applies regularised constant velocity or density boundary condition to concave edge for D3Q27 lattice.

- `int` *fD3Q27VCCRegular()*

  Applies regularised constant velocity or density boundary condition to concave corner for D3Q27 lattice.

- `int` *fD3Q27VPSCLBERegular()*

  Applies regularised constant velocity boundary condition to planar surface for D3Q27 lattice with cascaded LBE collisions.

- `int` *fD3Q27VCECLBERegular()*

  Applies regularised constant velocity or density boundary condition to concave edge for D3Q27 lattice with cascaded LBE collisions.

- `int` *fD3Q27VCCCLBERegular()*

  Applies regularised constant velocity or density boundary condition to concave corner for D3Q27 lattice with cascaded LBE collisions.

- `int` *fD3Q27VFRegular()*

  Applies constant velocity regularised boundary condition to lattice point using D3Q27 lattice scheme.

- `int` *fD3Q27PPSRegular()*

  Applies regularised constant density boundary condition to planar surface for D3Q27 lattice.

- `int` *fD3Q27PFRegular()*

  Applies constant density regularised boundary condition to lattice point using D3Q27 lattice scheme.

5.27.2 Function Documentation

fD2Q9PCERegular()

```
int fD2Q9PCERegular (double * p,
                     double * force,
                     double * f0, double * f1, double * f2,
                     double * f3, double * f4, double * f5,
                     double * f6, double * f7, double * f8,
                     double & vel)
```

Determines the required distribution functions to complete a regularised boundary condition for fixed fluid densities at a concave edge using the two-dimensional D2Q9 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions: this routine can also be used for systems with cascaded LBE collisions as the local equilibrium distribution functions for these result in the same non-equilibrium momentum stress tensors and expressions for replacement distribution functions. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom concave edges (PCETF) but can be used for any concave edge by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in,out	f2	Distribution functions for link 2 at edge lattice site
in,out	f3	Distribution functions for link 3 at edge lattice site
in,out	f4	Distribution functions for link 4 at edge lattice site
in,out	f5	Distribution functions for link 5 at edge lattice site
in,out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD2Q9PCESwiftRegular()

```
int fD2Q9PCESwiftRegular (double * p,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double drdx, double drdy,
                          double dpdx, double dpdy,
                          double nabr, double nabp,
                          double * omega,
                          double T,
                          double & vel)
```

Determines the required distribution functions to complete a regularised boundary condition for fixed fluid densities at a concave edge using the two-dimensional D2Q9 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom concave edges (PCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for density/concentration gradients (which may be swapped around).

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in,out	f2	Distribution functions for link 2 at edge lattice site
in,out	f3	Distribution functions for link 3 at edge lattice site
in,out	f4	Distribution functions for link 4 at edge lattice site
in,out	f5	Distribution functions for link 5 at edge lattice site
in,out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD2Q9PFRegular()

```
int fD2Q9PFRegular (long tpos,
                    int prop,
                    double * p0,
                    double * uwall,
                    double T)
```

Applies the appropriate regularised boundary condition for constant fluid densities based on types of collisions, interactions and direction for a two-dimensional D2Q9 lattice. (In this case, there are boundary options for cascaded LBE collisions, Swift free-energy interactions, as well as concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying regularised boundary condition
in	T	Temperature at boundary grid point

fD2Q9VCCCLBERegular()

```
int fD2Q9VCCCLBERegular (double * p,
                         double v0, double v1,
                         double * force,
                         double * f0, double * f1, double * f2,
                         double * f3, double * f4, double * f5,
                         double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave corner using the two-dimensional D2Q9 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left concave corners

(VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in,out	f2	Distribution functions for link 2 at corner lattice site
in,out	f3	Distribution functions for link 3 at corner lattice site
in,out	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site

fD2Q9VCCRegular()

```
int fD2Q9VCCRegular (double * p,
                     double v0, double v1,
                     double * force,
                     double * f0, double * f1, double * f2,
                     double * f3, double * f4, double * f5,
                     double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave corner using the two-dimensional D2Q9 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in,out	f2	Distribution functions for link 2 at corner lattice site
in,out	f3	Distribution functions for link 3 at corner lattice site
in,out	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site

fD2Q9VCCSwiftRegular()

```
int fD2Q9VCCSwiftRegular (double * p,
                          double v0, double v1,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double drdx, double drdy,
                          double dpdx, double dpdy,
                          double nabr, double nabp,
                          double * omega,
                          double T)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave corner using the two-dimensional D2Q9 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expressions in this subroutine are for bottom-left concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in,out	f2	Distribution functions for link 2 at corner lattice site
in,out	f3	Distribution functions for link 3 at corner lattice site
in,out	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD2Q9VCECLBERegular()

```
int fD2Q9VCECLBERegular (double v0, double v1,
                         double * force,
                         double * f0, double * f1, double * f2,
                         double * f3, double * f4, double * f5,
                         double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity at a concave edge using the two-dimensional D2Q9 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom concave edges (VCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in,out	f2	Distribution functions for link 2 at edge lattice site
in,out	f3	Distribution functions for link 3 at edge lattice site
in,out	f4	Distribution functions for link 4 at edge lattice site
in,out	f5	Distribution functions for link 5 at edge lattice site
in,out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site

fD2Q9VCERegular()

```
int fD2Q9VCERegular (double v0, double v1,
                     double * force,
                     double * f0, double * f1, double * f2,
                     double * f3, double * f4, double * f5,
                     double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity at a concave edge using the two-dimensional D2Q9 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom concave edges (VCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in,out	f2	Distribution functions for link 2 at edge lattice site
in,out	f3	Distribution functions for link 3 at edge lattice site
in,out	f4	Distribution functions for link 4 at edge lattice site
in,out	f5	Distribution functions for link 5 at edge lattice site
in,out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site

fD2Q9VCESwiftRegular()

```
int fD2Q9VCESwiftRegular (double v0, double v1,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double drdx, double drdy,
                          double dpdx, double dpdy,
                          double nabr, double nabp,
                          double * omega,
                          double T)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity at a concave edge using the two-dimensional D2Q9 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expressions in this subroutine are for bottom concave edges (VCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around).

Parameters

in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in,out	f2	Distribution functions for link 2 at edge lattice site
in,out	f3	Distribution functions for link 3 at edge lattice site
in,out	f4	Distribution functions for link 4 at edge lattice site
in,out	f5	Distribution functions for link 5 at edge lattice site
in,out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD2Q9VFRegular()

```
int fD2Q9VFRegular (long tpos,
                    long tpos1,
                    int prop,
                    double * uwall,
                    double dx,
                    double dy,
                    double T)
```

Applies the appropriate regularised boundary condition for a constant velocity based on types of collisions, interactions and direction for a two-dimensional D2Q9 lattice. (In this case, there are boundary options for cascaded LBE collisions and Swift free-energy interactions, as well as concave edges and corners.) For corners with Swift free-energy interactions, the vector between the boundary lattice point and sampling point for densities can be specified to correct fluid density/concentration using gradients of those properties evalulated at the boundary point.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	tpos1	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site
in	dx	Vector to move from current lattice site (x-component)
in	dy	Vector to move from current lattice site (y-component)
in	T	Temperature at boundary grid point

fD3Q15PFRegular()

```
int fD3Q15PFRegular (long tpos,
                     int prop,
                     double * p0,
                     double * uwall,
                     double T)
```

Applies the appropriate regularised boundary condition for constant fluid densities based on types of collisions, interactions and direction for a three-dimensional D3Q15 lattice. (In this case, there are boundary options for Swift free-energy interactions, as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying regularised boundary condition
in	T	Temperature at boundary grid point

fD3Q15PPSRegular()

```
int fD3Q15PPSRegular (double * p,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double & vel)
```

Determines the required distribution functions to complete a regularised boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at surface lattice site
in,out	f1	Distribution functions for link 1 at surface lattice site
in,out	f2	Distribution functions for link 2 at surface lattice site
in,out	f3	Distribution functions for link 3 at surface lattice site
in,out	f4	Distribution functions for link 4 at surface lattice site
in,out	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in,out	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in,out	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in,out	f13	Distribution functions for link 13 at surface lattice site
in,out	f14	Distribution functions for link 14 at surface lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q15PPSSwiftRegular()

```
int fD3Q15PPSSwiftRegular (double * p,
                            double * force,
                            double * f0, double * f1, double * f2,
                            double * f3, double * f4, double * f5,
                            double * f6, double * f7, double * f8,
                            double * f9, double * f10, double * f11,
                            double * f12, double * f13, double * f14,
                            double drdx, double drdy, double drdz,
                            double dpdx, double dpdy, double dpdz,
                            double nabr, double nabp,
                            double * omega,
                            double T,
                            double & vel)
```

Determines the required distribution functions to complete a regularised boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q15 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at surface lattice site
in,out	f1	Distribution functions for link 1 at surface lattice site
in,out	f2	Distribution functions for link 2 at surface lattice site
in,out	f3	Distribution functions for link 3 at surface lattice site
in,out	f4	Distribution functions for link 4 at surface lattice site
in,out	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in,out	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in,out	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in,out	f13	Distribution functions for link 13 at surface lattice site
in,out	f14	Distribution functions for link 14 at surface lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q15VCCRegular()

```
int fD3Q15VCCRegular (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at corner lattice site
in,out	f1	Distribution functions for link 1 at corner lattice site
in,out	f2	Distribution functions for link 2 at corner lattice site
in,out	f3	Distribution functions for link 3 at corner lattice site
in,out	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site

fD3Q15VCCSwiftRegular()

```
int fD3Q15VCCSwiftRegular (double * p,
                           double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave cprmer using the three-dimensional D3Q15 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
in	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q15VCERegular()

```
int fD3Q15VCERegular (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at edge lattice site
in,out	f1	Distribution functions for link 1 at edge lattice site
in,out	f2	Distribution functions for link 2 at edge lattice site
in,out	f3	Distribution functions for link 3 at edge lattice site
in,out	f4	Distribution functions for link 4 at edge lattice site
in,out	f5	Distribution functions for link 5 at edge lattice site
out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	f9	Distribution functions for link 9 at edge lattice site
in,out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site

fD3Q15VCESwiftRegular()

```
int fD3Q15VCESwiftRegular (double * p,
                           double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q15 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at edge lattice site
in,out	f1	Distribution functions for link 1 at edge lattice site
in,out	f2	Distribution functions for link 2 at edge lattice site
in,out	f3	Distribution functions for link 3 at edge lattice site
in,out	f4	Distribution functions for link 4 at edge lattice site
in,out	f5	Distribution functions for link 5 at edge lattice site
out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	f9	Distribution functions for link 9 at edge lattice site
in,out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q15VFRegular()

```
int fD3Q15VFRegular (long tpos,
                     long rpos,
                     int prop,
                     double * uwall,
                     double dx,
                     double dy,
                     double dz,
                     double T)
```

Applies the appropriate regularised boundary condition for a constant velocity based on types of interactions and direction for a three-dimensional D3Q15 lattice. (In this case, there are boundary options for Swift free-energy interactions, as well as planar surfaces, concave edges and corners.) For edges and corners with Swift free-energy interactions, the vector between the boundary lattice point and sampling point for densities can be specified to correct fluid density/concentration using gradients of those properties evalulated at the boundary point.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	rpos	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site
in	dx	Vector to move from current lattice site (x-component)
in	dy	Vector to move from current lattice site (y-component)
in	dz	Vector to move from current lattice site (z-component)
in	T	Temperature at boundary grid point

fD3Q15VPSRegular()

```
int fD3Q15VPSRegular (double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at surface lattice site
in,out	f1	Distribution functions for link 1 at surface lattice site
in,out	f2	Distribution functions for link 2 at surface lattice site
in,out	f3	Distribution functions for link 3 at surface lattice site
in,out	f4	Distribution functions for link 4 at surface lattice site
in,out	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in,out	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in,out	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in,out	f13	Distribution functions for link 13 at surface lattice site
in,out	f14	Distribution functions for link 14 at surface lattice site

fD3Q15VPSSwiftRegular()

```
int fD3Q15VPSSwiftRegular (double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q15 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at surface lattice site
in,out	f1	Distribution functions for link 1 at surface lattice site
in,out	f2	Distribution functions for link 2 at surface lattice site
in,out	f3	Distribution functions for link 3 at surface lattice site
in,out	f4	Distribution functions for link 4 at surface lattice site
in,out	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in,out	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in,out	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in,out	f13	Distribution functions for link 13 at surface lattice site
in,out	f14	Distribution functions for link 14 at surface lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q19PFRegular()

```
int fD3Q19PFRegular (long tpos,
                     int prop,
                     double * p0,
                     double * uwall,
                     double T)
```

Applies the appropriate regularised boundary condition for constant fluid densities based on types of collisions, interactions and direction for a three-dimensional D3Q19 lattice. (In this case, there are boundary options for cascaded LBE collisions, Swift free-energy interactions, as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying regularised boundary condition
in	T	Temperature at boundary grid point

fD3Q19PPSRegular()

```
int fD3Q19PPSRegular (double * p,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18,
                      double & vel)
```

Determines the required distribution functions to complete a regularised boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions: this routine can also be used for systems with cascaded LBE collisions as the local equilibrium distribution functions for these result in the same non-equilibrium momentum stress tensors and expressions for replacement distribution functions. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at surface lattice site
in,out	f1	Distribution functions for link 1 at surface lattice site
in,out	f2	Distribution functions for link 2 at surface lattice site
in,out	f3	Distribution functions for link 3 at surface lattice site
in,out	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in,out	f6	Distribution functions for link 6 at surface lattice site
in,out	f7	Distribution functions for link 7 at surface lattice site
in,out	f8	Distribution functions for link 8 at surface lattice site
in,out	f9	Distribution functions for link 9 at surface lattice site
in,out	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in,out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in,out	f14	Distribution functions for link 14 at surface lattice site
in,out	f15	Distribution functions for link 15 at surface lattice site
in,out	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q19PPSSwiftRegular()

```
int fD3Q19PPSSwiftRegular (double * p,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double * f15, double * f16, double * f17,
                           double * f18,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T,
                           double & vel)
```

Determines the required distribution functions to complete a regularised boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q19 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at surface lattice site

continues on next page

Table 5.40 – continued from previous page

in,out	f1	Distribution functions for link 1 at surface lattice site
in,out	f2	Distribution functions for link 2 at surface lattice site
in,out	f3	Distribution functions for link 3 at surface lattice site
in,out	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in,out	f6	Distribution functions for link 6 at surface lattice site
in,out	f7	Distribution functions for link 7 at surface lattice site
in,out	f8	Distribution functions for link 8 at surface lattice site
in,out	f9	Distribution functions for link 9 at surface lattice site
in,out	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in,out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in,out	f14	Distribution functions for link 14 at surface lattice site
in,out	f15	Distribution functions for link 15 at surface lattice site
in,out	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q19VCCCLBERegular()

```
int fD3Q19VCCCLBERegular (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q19 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site

fD3Q19VCCRegular()

```
int fD3Q19VCCRegular (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at corner lattice site
in,out	f1	Distribution functions for link 1 at corner lattice site
in,out	f2	Distribution functions for link 2 at corner lattice site
in,out	f3	Distribution functions for link 3 at corner lattice site
in,out	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in,out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in,out	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site

fD3Q19VCCSwiftRegular()

```
int fD3Q19VCCSwiftRegular (double * p,
                           double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double * f15, double * f16, double * f17,
                           double * f18,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q19 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed valu
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at corner lattice site
in,out	f1	Distribution functions for link 1 at corner lattice site
in,out	f2	Distribution functions for link 2 at corner lattice site
in,out	f3	Distribution functions for link 3 at corner lattice site
in,out	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in,out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in,out	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q19VCECLBERegular()

```
int fD3Q19VCECLBERegular (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q19 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at edge lattice site
in,out	f1	Distribution functions for link 1 at edge lattice site
in,out	f2	Distribution functions for link 2 at edge lattice site
in,out	f3	Distribution functions for link 3 at edge lattice site
in,out	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in,out	f6	Distribution functions for link 6 at edge lattice site
in,out	f7	Distribution functions for link 7 at edge lattice site
in,out	f8	Distribution functions for link 8 at edge lattice site
in,out	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in,out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site

fD3Q19VCERegular()

```
int fD3Q19VCERegular (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at edge lattice site
in,out	f1	Distribution functions for link 1 at edge lattice site
in,out	f2	Distribution functions for link 2 at edge lattice site
in,out	f3	Distribution functions for link 3 at edge lattice site
in,out	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in,out	f6	Distribution functions for link 6 at edge lattice site
in,out	f7	Distribution functions for link 7 at edge lattice site
in,out	f8	Distribution functions for link 8 at edge lattice site
in,out	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in,out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site

fD3Q19VCESwiftRegular()

```
int fD3Q19VCESwiftRegular (double * p,
                           double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double * f15, double * f16, double * f17,
                           double * f18,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q19 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed value
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at edge lattice site
in,out	f1	Distribution functions for link 1 at edge lattice site
in,out	f2	Distribution functions for link 2 at edge lattice site
in,out	f3	Distribution functions for link 3 at edge lattice site
in,out	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in,out	f6	Distribution functions for link 6 at edge lattice site
in,out	f7	Distribution functions for link 7 at edge lattice site
in,out	f8	Distribution functions for link 8 at edge lattice site
in,out	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in,out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q19VFRegular()

```
int fD3Q19VFRegular (long tpos,
                     long rpos,
                     int prop,
                     double * uwall,
                     double dx,
                     double dy,
                     double dz,
                     double T)
```

Applies the appropriate regularised boundary condition for a constant velocity based on types of collisions, interactions and direction for a three-dimensional D3Q19 lattice. (In this case, there are boundary options for cascaded LBE collsions, Swift free-energy interactions, as well as planar surfaces, concave edges and corners.) For edges and corners with Swift free-energy interactions, the vector between the boundary lattice point and sampling point for densities can be specified to correct fluid density/concentration using gradients of those properties evalulated at the boundary point.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	rpos	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site
in	dx	Vector to move from current lattice site (x-component)
in	dy	Vector to move from current lattice site (y-component)
in	dz	Vector to move from current lattice site (z-component)
in	T	Temperature at boundary grid point

fD3Q19VPSCLBERegular()

```
int fD3Q19VPSCLBERegular (double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q19 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at surface lattice site
in,out	f1	Distribution functions for link 1 at surface lattice site
in,out	f2	Distribution functions for link 2 at surface lattice site
in,out	f3	Distribution functions for link 3 at surface lattice site
in,out	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in,out	f6	Distribution functions for link 6 at surface lattice site
in,out	f7	Distribution functions for link 7 at surface lattice site
in,out	f8	Distribution functions for link 8 at surface lattice site
in,out	f9	Distribution functions for link 9 at surface lattice site
in,out	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in,out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in,out	f14	Distribution functions for link 14 at surface lattice site
in,out	f15	Distribution functions for link 15 at surface lattice site
in,out	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site

fD3Q19VPSRegular()

```
int fD3Q19VPSRegular (double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at surface lattice site
in,out	f1	Distribution functions for link 1 at surface lattice site
in,out	f2	Distribution functions for link 2 at surface lattice site
in,out	f3	Distribution functions for link 3 at surface lattice site
in,out	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in,out	f6	Distribution functions for link 6 at surface lattice site
in,out	f7	Distribution functions for link 7 at surface lattice site
in,out	f8	Distribution functions for link 8 at surface lattice site
in,out	f9	Distribution functions for link 9 at surface lattice site
in,out	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in,out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in,out	f14	Distribution functions for link 14 at surface lattice site
in,out	f15	Distribution functions for link 15 at surface lattice site
in,out	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site

fD3Q19VPSSwiftRegular()

```
int fD3Q19VPSSwiftRegular (double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double * f15, double * f16, double * f17,
                           double * f18,
                           double drdx, double drdy, double drdz,
```

(continues on next page)

(continued from previous page)

```
                        double dpdx, double dpdy, double dpdz,
                        double nabr, double nabp,
                        double * omega,
                        double T)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q19 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at surface lattice site
in,out	f1	Distribution functions for link 1 at surface lattice site
in,out	f2	Distribution functions for link 2 at surface lattice site
in,out	f3	Distribution functions for link 3 at surface lattice site
in,out	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in,out	f6	Distribution functions for link 6 at surface lattice site
in,out	f7	Distribution functions for link 7 at surface lattice site
in,out	f8	Distribution functions for link 8 at surface lattice site
in,out	f9	Distribution functions for link 9 at surface lattice site
in,out	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in,out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in,out	f14	Distribution functions for link 14 at surface lattice site
in,out	f15	Distribution functions for link 15 at surface lattice site
in,out	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q27PFRegular()

```
int fD3Q27PFRegular (long tpos,
                     int prop,
                     double * p0,
                     double * uwall)
```

Applies the appropriate regularised boundary condition for constant fluid densities based on types of collisions and direction for a three-dimensional D3Q27 lattice. (In this case, there are boundary options for cascaded LBE collisions as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying regularised boundary condition

fD3Q27PPSRegular()

```
int fD3Q27PPSRegular (double * p,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26,
                      double & vel)
```

Determines the required distribution functions to complete a regularised boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions: this routine can also be used for systems with cascaded LBE collisions as the local equilibrium distribution functions for these result in the same same non-equilibrium momentum stress tensors and expressions for replacement distribution functions. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at surface lattice site
in,out	f1	Distribution functions for link 1 at surface lattice site
in,out	f2	Distribution functions for link 2 at surface lattice site
in,out	f3	Distribution functions for link 3 at surface lattice site
in,out	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in,out	f6	Distribution functions for link 6 at surface lattice site
in,out	f7	Distribution functions for link 7 at surface lattice site
in,out	f8	Distribution functions for link 8 at surface lattice site
in,out	f9	Distribution functions for link 9 at surface lattice site
in,out	f10	Distribution functions for link 10 at surface lattice site
in,out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in,out	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in,out	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in,out	f18	Distribution functions for link 18 at surface lattice site
in,out	f19	Distribution functions for link 19 at surface lattice site
in,out	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site
out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in,out	f25	Distribution functions for link 25 at surface lattice site
in,out	f26	Distribution functions for link 26 at surface lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q27VCCCLBERegular()

```
int fD3Q27VCCCLBERegular (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18, double * f19, double * f20,
                          double * f21, double * f22, double * f23,
                          double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q27 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed value
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at corner lattice site
in,out	f1	Distribution functions for link 1 at corner lattice site
in,out	f2	Distribution functions for link 2 at corner lattice site
in,out	f3	Distribution functions for link 3 at corner lattice site
in,out	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in,out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in,out	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in,out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site
out	f19	Distribution functions for link 19 at corner lattice site
out	f20	Distribution functions for link 20 at corner lattice site
out	f21	Distribution functions for link 21 at corner lattice site
out	f22	Distribution functions for link 22 at corner lattice site
out	f23	Distribution functions for link 23 at corner lattice site
out	f24	Distribution functions for link 24 at corner lattice site
out	f25	Distribution functions for link 25 at corner lattice site
out	f26	Distribution functions for link 26 at corner lattice site

fD3Q27VCCRegular()

```
int fD3Q27VCCRegular (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed value
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at corner lattice site
in,out	f1	Distribution functions for link 1 at corner lattice site
in,out	f2	Distribution functions for link 2 at corner lattice site
in,out	f3	Distribution functions for link 3 at corner lattice site
in,out	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in,out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in,out	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in,out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site
out	f19	Distribution functions for link 19 at corner lattice site
out	f20	Distribution functions for link 20 at corner lattice site
out	f21	Distribution functions for link 21 at corner lattice site
out	f22	Distribution functions for link 22 at corner lattice site
out	f23	Distribution functions for link 23 at corner lattice site
out	f24	Distribution functions for link 24 at corner lattice site
out	f25	Distribution functions for link 25 at corner lattice site
out	f26	Distribution functions for link 26 at corner lattice site

fD3Q27VCECLBERegular()

```
int fD3Q27VCECLBERegular (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18, double * f19, double * f20,
                          double * f21, double * f22, double * f23,
                          double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q27 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at edge lattice site
in,out	f1	Distribution functions for link 1 at edge lattice site
in,out	f2	Distribution functions for link 2 at edge lattice site
in,out	f3	Distribution functions for link 3 at edge lattice site
in,out	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in,out	f6	Distribution functions for link 6 at edge lattice site
in,out	f7	Distribution functions for link 7 at edge lattice site
in,out	f8	Distribution functions for link 8 at edge lattice site
in,out	f9	Distribution functions for link 9 at edge lattice site
in,out	f10	Distribution functions for link 10 at edge lattice site
in,out	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
in,out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
out	f19	Distribution functions for link 19 at edge lattice site
out	f20	Distribution functions for link 20 at edge lattice site
out	f21	Distribution functions for link 21 at edge lattice site
out	f22	Distribution functions for link 22 at edge lattice site
out	f23	Distribution functions for link 23 at edge lattice site
out	f24	Distribution functions for link 24 at edge lattice site
out	f25	Distribution functions for link 25 at edge lattice site
out	f26	Distribution functions for link 26 at edge lattice site

fD3Q27VCERegular()

```
int fD3Q27VCERegular (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at edge lattice site
in,out	f1	Distribution functions for link 1 at edge lattice site
in,out	f2	Distribution functions for link 2 at edge lattice site
in,out	f3	Distribution functions for link 3 at edge lattice site
in,out	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in,out	f6	Distribution functions for link 6 at edge lattice site
in,out	f7	Distribution functions for link 7 at edge lattice site
in,out	f8	Distribution functions for link 8 at edge lattice site
in,out	f9	Distribution functions for link 9 at edge lattice site
in,out	f10	Distribution functions for link 10 at edge lattice site
in,out	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
in,out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
out	f19	Distribution functions for link 19 at edge lattice site
out	f20	Distribution functions for link 20 at edge lattice site
out	f21	Distribution functions for link 21 at edge lattice site
out	f22	Distribution functions for link 22 at edge lattice site
out	f23	Distribution functions for link 23 at edge lattice site
out	f24	Distribution functions for link 24 at edge lattice site
out	f25	Distribution functions for link 25 at edge lattice site
out	f26	Distribution functions for link 26 at edge lattice site

fD3Q27VFRegular()

```
int fD3Q27VFRegular (long tpos,
                     long rpos,
                     int prop,
                     double * uwall)
```

Applies the appropriate regularised boundary condition for a constant velocity based on types of collisions and direction for a three-dimensional D3Q27 lattice. (In this case, there are boundary options for cascaded LBE collsions as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	rpos	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site

fD3Q27VPSCLBERegular()

```
int fD3Q27VPSCLBERegular (double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18, double * f19, double * f20,
                          double * f21, double * f22, double * f23,
                          double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q27 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at surface lattice site
in,out	f1	Distribution functions for link 1 at surface lattice site
in,out	f2	Distribution functions for link 2 at surface lattice site
in,out	f3	Distribution functions for link 3 at surface lattice site
in,out	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in,out	f6	Distribution functions for link 6 at surface lattice site
in,out	f7	Distribution functions for link 7 at surface lattice site
in,out	f8	Distribution functions for link 8 at surface lattice site
in,out	f9	Distribution functions for link 9 at surface lattice site
in,out	f10	Distribution functions for link 10 at surface lattice site
in,out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in,out	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in,out	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in,out	f18	Distribution functions for link 18 at surface lattice site
in,out	f19	Distribution functions for link 19 at surface lattice site
in,out	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site
out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in,out	f25	Distribution functions for link 25 at surface lattice site
in,out	f26	Distribution functions for link 26 at surface lattice site

fD3Q27VPSRegular()

```
int fD3Q27VPSRegular (double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a regularised boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in,out	f0	Distribution functions for link 0 at surface lattice site
in,out	f1	Distribution functions for link 1 at surface lattice site
in,out	f2	Distribution functions for link 2 at surface lattice site
in,out	f3	Distribution functions for link 3 at surface lattice site
in,out	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in,out	f6	Distribution functions for link 6 at surface lattice site
in,out	f7	Distribution functions for link 7 at surface lattice site
in,out	f8	Distribution functions for link 8 at surface lattice site
in,out	f9	Distribution functions for link 9 at surface lattice site
in,out	f10	Distribution functions for link 10 at surface lattice site
in,out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in,out	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in,out	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in,out	f18	Distribution functions for link 18 at surface lattice site
in,out	f19	Distribution functions for link 19 at surface lattice site
in,out	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site
out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in,out	f25	Distribution functions for link 25 at surface lattice site
in,out	f26	Distribution functions for link 26 at surface lattice site

5.28 lbpBOUNDKinetic.cpp

Module for applying kinetic boundary conditions. (Header file available as lbpBOUNDKinetic.hpp.)

Applies kinetic [5] boundary conditions at specified lattice points to give fixed fluid velocities or densities. This scheme uses local equilibrium distribution functions for 'missing' distribution functions re-entering the simulation box with adjusted densities ρ', which are obtained by ensuring mass conservation between distribution functions leaving and entering the system, i.e.

$$\sum_{i,in} f_i^{eq}\left(\rho', \vec{u}_w\right) = \sum_{i,out} f_i,$$

where rearranging the above expression leads to the required adjusted density. For boundaries other than concave edges in two dimensions or planar surfaces in three dimensions where 'buried' links that neither enter nor leave the simulation box are included, only the non-buried ('active') links are used for the mass conservation expression to obtain ρ'. The buried links are obtained by ensuring the overall density and momentum are correct, as obtained using the following summations:

$$\rho = \sum_i f_i,$$

$$\rho u_\alpha = \sum_i f_i e_{i,\alpha}.$$

In cases where there are more unknown buried links than density and momentum equations, each buried link is expressed as a combination of terms for these three or four components, which are solved to give the correct density and momentum at the lattice point.

5.28.1 Functions

- `int` *fD2Q9VCEKinetic()*

  Applies kinetic constant velocity boundary condition to concave edge for D2Q9 lattice.

- `int` *fD2Q9VCCKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave corner for D2Q9 lattice.

- `int` *fD2Q9VCECLBEKinetic()*

  Applies kinetic constant velocity boundary condition to concave edge for D2Q9 lattice with cascaded LBE collisions.

- `int` *fD2Q9VCCCLBEKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave corner for D2Q9 lattice with cascaded LBE collisions.

- `int` *fD2Q9VCESwiftKinetic()*

  Applies kinetic constant velocity boundary condition to concave edge for D2Q9 lattice with Swift free-energy interactions.

- `int` *fD2Q9VCCSwiftKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave corner for D2Q9 lattice with Swift free-energy interactions.

- `int` *fD2Q9VFKinetic()*

  Applies constant velocity kinetic boundary condition to lattice point using D2Q9 lattice scheme.

- `int` *fD2Q9PCEKinetic()*

  Applies kinetic constant density boundary condition to concave edge for D2Q9 lattice.

- `int` *fD2Q9PCESwiftKinetic()*

  Applies kinetic constant density boundary condition to concave edge for D2Q9 lattice with Swift free-energy interactions.

- `int` *fD2Q9PFKinetic()*

  Applies constant density kinetic boundary condition to lattice point using D2Q9 lattice scheme.

- `int` *fD3Q15VPSKinetic()*

  Applies kinetic constant velocity boundary condition to planar surface for D3Q15 lattice.

- `int` *fD3Q15VCEKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave edge for D3Q15 lattice.

- `int` *fD3Q15VCCKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave corner for D3Q15 lattice.

- `int` *fD3Q15VPSSwiftKinetic()*

  Applies kinetic constant velocity boundary condition to planar surface for D3Q15 lattice with Swift free-energy interactions.

- `int` *fD3Q15VCESwiftKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave edge for D3Q15 lattice with Swift free-energy interactions.

- `int` *fD3Q15VCCSwiftKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave corner for D3Q15 lattice with Swift free-energy interactions.

- `int` *fD3Q15VFKinetic()*

  Applies constant velocity kinetic boundary condition to lattice point using D3Q15 lattice scheme.

- `int` *fD3Q15PPSKinetic()*

  Applies kinetic constant density boundary condition to planar surface for D3Q15 lattice.

- `int` *fD3Q15PPSSwiftKinetic()*

  Applies kinetic constant density boundary condition to planar surface for D3Q15 lattice with Swift free-energy interactions.

- `int` *fD3Q15PFKinetic()*

  Applies constant density kinetic boundary condition to lattice point using D3Q15 lattice scheme.

- `int` *fD3Q19VPSKinetic()*

  Applies kinetic constant velocity boundary condition to planar surface for D3Q19 lattice.

- `int` *fD3Q19VCEKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave edge for D3Q19 lattice.

- `int` *fD3Q19VCCKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave corner for D3Q19 lattice.

- `int` *fD3Q19VPSCLBEKinetic()*

  Applies kinetic constant velocity boundary condition to planar surface for D3Q19 lattice with cascaded LBE collisions.

- `int` *fD3Q19VCECLBEKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave edge for D3Q19 lattice with cascaded LBE collisions.

- `int` *fD3Q19VCCCLBEKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave corner for D3Q19 lattice with cascaded LBE collisions.

- `int` *fD3Q19VPSSwiftKinetic()*

  Applies kinetic constant velocity boundary condition to planar surface for D3Q19 lattice with Swift free-energy interactions.

- `int` *fD3Q19VCESwiftKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave edge for D3Q19 lattice with Swift free-energy interactions.

- `int` *fD3Q19VCCSwiftKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave corner for D3Q19 lattice with Swift free-energy interactions.

- `int` *fD3Q19VFKinetic()*

  Applies constant velocity kinetic boundary condition to lattice point using D3Q19 lattice scheme.

- `int` *fD3Q19PPSKinetic()*

  Applies kinetic constant density boundary condition to planar surface for D3Q19 lattice.

- `int` *fD3Q19PPSSwiftKinetic()*

  Applies kinetic constant density boundary condition to planar surface for D3Q19 lattice with Swift free-energy interactions.

- `int` *fD3Q19PFKinetic()*

  Applies constant density kinetic boundary condition to lattice point using D3Q19 lattice scheme.

- `int` *fD3Q27VPSKinetic()*

  Applies kinetic constant velocity boundary condition to planar surface for D3Q27 lattice.

- `int` *fD3Q27VCEKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave edge for D3Q27 lattice.

- `int` *fD3Q27VCCKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave corner for D3Q27 lattice.

- `int` *fD3Q27VPSCLBEKinetic()*

  Applies kinetic constant velocity boundary condition to planar surface for D3Q27 lattice with cascaded LBE collisions.

- `int` *fD3Q27VCECLBEKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave edge for D3Q27 lattice with cascaded LBE collisions.

- `int` *fD3Q27VCCCLBEKinetic()*

  Applies kinetic constant velocity or density boundary condition to concave corner for D3Q27 lattice with cascaded LBE collisions.

- `int` *fD3Q27VFKinetic()*

  Applies constant velocity kinetic boundary condition to lattice point using D3Q27 lattice scheme.

- `int` *fD3Q27PPSKinetic()*

  Applies kinetic constant density boundary condition to planar surface for D3Q27 lattice.

- `int` *fD3Q27PFKinetic()*

  Applies constant density kinetic boundary condition to lattice point using D3Q27 lattice scheme.

5.28.2 Function Documentation

fD2Q9PCEKinetic()

```
int fD2Q9PCEKinetic (double * p,
                     double * force,
                     double * f0, double * f1, double * f2,
                     double * f3, double * f4, double * f5,
                     double * f6, double * f7, double * f8,
                     double & vel)
```

Determines the required distribution functions to complete a kinetic boundary condition for fixed fluid densities at a concave edge using the two-dimensional D2Q9 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions: this routine can also be used for systems with cascaded LBE collisions as the local equilibrium distribution functions for these result in the same adjusted densities ρ'. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom concave edges (PCETF) but can be used for any concave edge by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD2Q9PCESwiftKinetic()

```
int fD2Q9PCESwiftKinetic (double * p,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double drdx, double drdy,
                          double dpdx, double dpdy,
                          double nabr, double nabp,
                          double * omega,
                          double T,
                          double & vel)
```

Determines the required distribution functions to complete a kinetic boundary condition for fixed fluid densities at a concave edge using the two-dimensional D2Q9 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the orthogonal velocity component: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). (The tangential velocity component is assumed equal to zero.) The expressions in this subroutine are for bottom concave edges

(PCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for density/concentration gradients (which may be swapped around).

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD2Q9PFKinetic()

```
int fD2Q9PFKinetic (long tpos,
                    int prop,
                    double * p0,
                    double * uwall,
                    double T)
```

Applies the appropriate kinetic boundary condition for constant fluid densities based on types of collisions, interactions and direction for a two-dimensional D2Q9 lattice. (In this case, there are boundary options for cascaded LBE collisions, Swift free-energy interactions, as well as concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying kinetic boundary condition
in	T	Temperature at boundary grid point

fD2Q9VCCCLBEKinetic()

```
int fD2Q9VCCCLBEKinetic (double * p,
                         double v0, double v1,
                         double * force,
                         double * f0, double * f1, double * f2,
                         double * f3, double * f4, double * f5,
                         double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave corner using the two-dimensional D2Q9 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site

fD2Q9VCCKinetic()

```
int fD2Q9VCCKinetic (double * p,
                     double v0, double v1,
                     double * force,
                     double * f0, double * f1, double * f2,
                     double * f3, double * f4, double * f5,
                     double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave corner using the two-dimensional D2Q9 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site

fD2Q9VCCSwiftKinetic()

```
int fD2Q9VCCSwiftKinetic (double * p,
                          double v0, double v1,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double drdx, double drdy,
                          double dpdx, double dpdy,
                          double nabr, double nabp,
                          double * omega,
                          double T)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave corner using the two-dimensional D2Q9 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the sum of both velocity components: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). The expressions in this subroutine are for bottom-left concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left corner)
in	v1	Velocity component at concave corner (y-component for bottom-left corner)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD2Q9VCECLBEKinetic()

```
int fD2Q9VCECLBEKinetic (double v0, double v1,
                         double * force,
                         double * f0, double * f1, double * f2,
                         double * f3, double * f4, double * f5,
                         double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity at a concave edge using the two-dimensional D2Q9 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom concave edges (VCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site

fD2Q9VCEKinetic()

```
int fD2Q9VCEKinetic (double v0, double v1,
                     double * force,
                     double * f0, double * f1, double * f2,
                     double * f3, double * f4, double * f5,
                     double * f6, double * f7, double * f8)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity at a concave edge using the two-dimensional D2Q9 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom concave edges (VCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site

fD2Q9VCESwiftKinetic()

```
int fD2Q9VCESwiftKinetic (double v0, double v1,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double drdx, double drdy,
                          double dpdx, double dpdy,
                          double nabr, double nabp,
                          double * omega,
                          double T)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity at a concave edge using the two-dimensional D2Q9 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the orthogonal velocity component: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). The expressions in this subroutine are for bottom concave edges (VCETF) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around).

Parameters

in	v0	Velocity component tangential to concave edge (x-component for bottom edge)
in	v1	Velocity component orthogonal to concave edge (y-component for bottom edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
out	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD2Q9VFKinetic()

```
int fD2Q9VFKinetic (long tpos,
                    long tpos1,
                    int prop,
                    double * uwall,
                    double dx,
                    double dy,
                    double T)
```

Applies the appropriate kinetic boundary condition for a constant velocity based on types of collisions, interactions and direction for a two-dimensional D2Q9 lattice. (In this case, there are boundary options for cascaded LBE collisions and Swift free-energy interactions, as well as concave edges and corners.) For corners with Swift free-energy interactions, the vector between the boundary lattice point and sampling point for densities can be specified to correct fluid density/concentration using gradients of those properties evalulated at the boundary point.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	tpos1	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site
in	dx	Vector to move from current lattice site (x-component)
in	dy	Vector to move from current lattice site (y-component)
in	T	Temperature at boundary grid point

fD3Q15PFKinetic()

```
int fD3Q15PFKinetic (long tpos,
                     int prop,
                     double * p0,
                     double * uwall,
                     double T)
```

Applies the appropriate kinetic boundary condition for constant fluid densities based on types of collisions, interactions and direction for a three-dimensional D3Q15 lattice. (In this case, there are boundary options for Swift free-energy interactions, as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying kinetic boundary condition
in	T	Temperature at boundary grid point

fD3Q15PPSKinetic()

```
int fD3Q15PPSKinetic (double * p,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double & vel)
```

Determines the required distribution functions to complete a kinetic boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q15PPSSwiftKinetic()

```
int fD3Q15PPSSwiftKinetic (double * p,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T,
                           double & vel)
```

Determines the required distribution functions to complete a kinetic boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q15 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the orthogonal velocity component: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). (The tangential velocity component is assumed equal to zero.) The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q15VCCKinetic()

```
int fD3Q15VCCKinetic (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site

fD3Q15VCCSwiftKinetic()

```
int fD3Q15VCCSwiftKinetic (double * p,
                           double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave cprmer using the three-dimensional D3Q15 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the sum of both velocity components: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
out	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
in	f5	Distribution functions for link 5 at corner lattice site
out	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
out	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q15VCEKinetic()

```
int fD3Q15VCEKinetic (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site

fD3Q15VCESwiftKinetic()

```
int fD3Q15VCESwiftKinetic (double * p,
                           double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q15 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the sum of both velocity components: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
in	f5	Distribution functions for link 5 at edge lattice site
out	f6	Distribution functions for link 6 at edge lattice site
out	f7	Distribution functions for link 7 at edge lattice site
out	f8	Distribution functions for link 8 at edge lattice site
out	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q15VFKinetic()

```
int fD3Q15VFKinetic (long tpos,
                     long rpos,
                     int prop,
                     double * uwall,
                     double dx,
                     double dy,
                     double dz,
                     double T)
```

Applies the appropriate kinetic boundary condition for a constant velocity based on types of interactions and direction for a three-dimensional D3Q15 lattice. (In this case, there are boundary options for Swift free-energy interactions, as well as planar surfaces, concave edges and corners.) For edges and corners with Swift free-energy interactions, the vector between the boundary lattice point and sampling point for densities can be specified to correct fluid density/concentration using gradients of those properties evalulated at the boundary point.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	rpos	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site
in	dx	Vector to move from current lattice site (x-component)
in	dy	Vector to move from current lattice site (y-component)
in	dz	Vector to move from current lattice site (z-component)
in	T	Temperature at boundary grid point

fD3Q15VPSKinetic()

```
int fD3Q15VPSKinetic (double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q15 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site

fD3Q15VPSSwiftKinetic()

```
int fD3Q15VPSSwiftKinetic (double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q15 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the orthogonal velocity component: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
in	f5	Distribution functions for link 5 at surface lattice site
out	f6	Distribution functions for link 6 at surface lattice site
out	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
out	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
in	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q19PFKinetic()

```
int fD3Q19PFKinetic (long tpos,
                     int prop,
                     double * p0,
                     double * uwall,
                     double T)
```

Applies the appropriate kinetic boundary condition for constant fluid densities based on types of collisions, interactions and direction for a three-dimensional D3Q19 lattice. (In this case, there are boundary options for cascaded LBE collisions, Swift free-energy interactions, as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying kinetic boundary condition
in	T	Temperature at boundary grid point

fD3Q19PPSKinetic()

```
int fD3Q19PPSKinetic (double * p,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18,
                      double & vel)
```

Determines the required distribution functions to complete a kinetic boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions: this routine can also be used for systems with cascaded LBE collisions as the local equilibrium distribution functions for these result in the same adjusted densities ρ'. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q19PPSSwiftKinetic()

```
int fD3Q19PPSSwiftKinetic (double * p,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double * f15, double * f16, double * f17,
                           double * f18,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T,
                           double & vel)
```

Determines the required distribution functions to complete a kinetic boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q19 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the orthogonal velocity component: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero.) (The tangential velocity component is assumed equal to zero). The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site

continues on next page

Table 5.50 – continued from previous page

in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q19VCCCLBEKinetic()

```
int fD3Q19VCCCLBEKinetic (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q19 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site

fD3Q19VCCKinetic()

```
int fD3Q19VCCKinetic (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site

fD3Q19VCCSwiftKinetic()

```
int fD3Q19VCCSwiftKinetic (double * p,
                           double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double * f15, double * f16, double * f17,
                           double * f18,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q19 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the sum of both velocity components: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
out	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q19VCECLBEKinetic()

```
int fD3Q19VCECLBEKinetic (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q19 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site

fD3Q19VCEKinetic()

```
int fD3Q19VCEKinetic (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values for constant density boundaries)
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site

fD3Q19VCESwiftKinetic()

```
int fD3Q19VCESwiftKinetic (double * p,
                           double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double * f15, double * f16, double * f17,
                           double * f18,
                           double drdx, double drdy, double drdz,
                           double dpdx, double dpdy, double dpdz,
                           double nabr, double nabp,
                           double * omega,
                           double T)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q19 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the sum of both velocity components: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components and density/concentration gradients (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values f
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
out	f10	Distribution functions for link 10 at edge lattice site
out	f11	Distribution functions for link 11 at edge lattice site
in	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
out	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q19VFKinetic()

```
int fD3Q19VFKinetic (long tpos,
                     long rpos,
                     int prop,
                     double * uwall,
                     double dx,
                     double dy,
                     double dz,
                     double T)
```

Applies the appropriate kinetic boundary condition for a constant velocity based on types of collisions, interactions and direction for a three-dimensional D3Q19 lattice. (In this case, there are boundary options for cascaded LBE collsions, Swift free-energy interactions, as well as planar surfaces, concave edges and corners.) For edges and corners with Swift free-energy interactions, the vector between the boundary lattice point and sampling point for densities can be specified to correct fluid density/concentration using gradients of those properties evalulated at the boundary point.

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	rpos	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site
in	dx	Vector to move from current lattice site (x-component)
in	dy	Vector to move from current lattice site (y-component)
in	dz	Vector to move from current lattice site (z-component)
in	T	Temperature at boundary grid point

fD3Q19VPSCLBEKinetic()

```
int fD3Q19VPSCLBEKinetic (double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q19 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site

fD3Q19VPSKinetic()

```
int fD3Q19VPSKinetic (double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q19 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site

fD3Q19VPSSwiftKinetic()

```
int fD3Q19VPSSwiftKinetic (double v0, double v1, double v2,
                           double * force,
                           double * f0, double * f1, double * f2,
                           double * f3, double * f4, double * f5,
                           double * f6, double * f7, double * f8,
                           double * f9, double * f10, double * f11,
                           double * f12, double * f13, double * f14,
                           double * f15, double * f16, double * f17,
                           double * f18,
                           double drdx, double drdy, double drdz,
```

(continues on next page)

(continued from previous page)

```
                              double dpdx, double dpdy, double dpdz,
                              double nabr, double nabp,
                              double * omega,
                              double T)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q19 lattice and Swift free-energy interactions. This routine can only be used for mildly compressible fluids using the local equilibrium distribution functions for free-energy calculations that incorporate density (and concentration) gradients: relaxation times or frequencies for fluids and the site temperature are required to calculate Galilean invariance parameters and bulk pressures. The expression for the adjusted densities ρ' includes division by the orthogonal velocity component: if this is zero, the actual fluid density or concentration is used instead to avoid numerical singularities (i.e. divisions by zero). The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
out	f11	Distribution functions for link 11 at surface lattice site
in	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
in	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
out	f18	Distribution functions for link 18 at surface lattice site
in	drdx	First-order derivative of fluid density at boundary grid point (x-component)
in	drdy	First-order derivative of fluid density at boundary grid point (y-component)
in	drdz	First-order derivative of fluid density at boundary grid point (z-component)
in	dpdx	First-order derivative of fluid concentration at boundary grid point (x-component)
in	dpdy	First-order derivative of fluid concentration at boundary grid point (y-component)
in	dpdz	First-order derivative of fluid concentration at boundary grid point (z-component)
in	nabr	Second-order derivative of fluid density at boundary grid point
in	nabp	Second-order derivative of fluid concentration at boundary grid point
in	omega	Relaxation frequencies (reciprocals of relaxation times) for fluids at boundary grid point
in	T	Temperature at boundary grid point

fD3Q27PFKinetic()

fD3Q27PFKinetic lbpBOUNDKinetic.cpp lbpBOUNDKinetic.cpp fD3Q27PFKinetic

int fD3Q27PFKinetic (long tpos, int prop, double * p0, double * uwall)

Applies the appropriate kinetic boundary condition for constant fluid densities based on types of collisions and direction for a three-dimensional D3Q27 lattice. (In this case, there are boundary options for cascaded LBE collisions as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	prop	Boundary condition code indicating type and direction
in	p0	Fluid densities for boundary lattice point
in,out	uwall	Velocity at boundary site determined from applying kinetic boundary condition

fD3Q27PPSKinetic()

```
int fD3Q27PPSKinetic (double * p,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26,
                      double & vel)
```

Determines the required distribution functions to complete a kinetic boundary condition for fixed fluid densities at a planar surface using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions: this routine can also be used for systems with cascaded LBE collisions as the local equilibrium distribution functions for these result in the same adjusted densities ρ'. The resulting orthogonal velocity component is subsequently used to specify the fluid velocity for solute concentration and temperature boundaries, while the tangential velocity component is assumed to be zero. The expressions in this subroutine are for bottom planar surfaces (PPST) but can be used for any planar surface by selecting different distribution functions.

Parameters

in	p	Fluid densities for boundary lattice point
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site
out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site
out	vel	Resulting fluid velocity in direction orthogonal to boundary

fD3Q27VCCCLBEKinetic()

```
int fD3Q27VCCCLBEKinetic (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18, double * f19, double * f20,
                          double * f21, double * f22, double * f23,
                          double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q27 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values f
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site
out	f19	Distribution functions for link 19 at corner lattice site
out	f20	Distribution functions for link 20 at corner lattice site
out	f21	Distribution functions for link 21 at corner lattice site
out	f22	Distribution functions for link 22 at corner lattice site
out	f23	Distribution functions for link 23 at corner lattice site
out	f24	Distribution functions for link 24 at corner lattice site
out	f25	Distribution functions for link 25 at corner lattice site
out	f26	Distribution functions for link 26 at corner lattice site

fD3Q27VCCKinetic()

```
int fD3Q27VCCKinetic (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave corner using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left-back concave corners (VCCTRF) but can be used for any concave corner by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave corner (sampled from nearby lattice point for constant velocity boundaries, fixed values f
in	v0	Velocity component at concave corner (x-component for bottom-left edge)
in	v1	Velocity component at concave corner (y-component for bottom-left edge)
in	v2	Velocity component at concave corner (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at corner lattice site
in	f1	Distribution functions for link 1 at corner lattice site
in	f2	Distribution functions for link 2 at corner lattice site
in	f3	Distribution functions for link 3 at corner lattice site
in	f4	Distribution functions for link 4 at corner lattice site
out	f5	Distribution functions for link 5 at corner lattice site
in	f6	Distribution functions for link 6 at corner lattice site
out	f7	Distribution functions for link 7 at corner lattice site
in	f8	Distribution functions for link 8 at corner lattice site
out	f9	Distribution functions for link 9 at corner lattice site
in	f10	Distribution functions for link 10 at corner lattice site
out	f11	Distribution functions for link 11 at corner lattice site
out	f12	Distribution functions for link 12 at corner lattice site
out	f13	Distribution functions for link 13 at corner lattice site
out	f14	Distribution functions for link 14 at corner lattice site
out	f15	Distribution functions for link 15 at corner lattice site
out	f16	Distribution functions for link 16 at corner lattice site
out	f17	Distribution functions for link 17 at corner lattice site
out	f18	Distribution functions for link 18 at corner lattice site
out	f19	Distribution functions for link 19 at corner lattice site
out	f20	Distribution functions for link 20 at corner lattice site
out	f21	Distribution functions for link 21 at corner lattice site
out	f22	Distribution functions for link 22 at corner lattice site
out	f23	Distribution functions for link 23 at corner lattice site
out	f24	Distribution functions for link 24 at corner lattice site
out	f25	Distribution functions for link 25 at corner lattice site
out	f26	Distribution functions for link 26 at corner lattice site

fD3Q27VCECLBEKinetic()

```
int fD3Q27VCECLBEKinetic (double * p,
                          double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18, double * f19, double * f20,
                          double * f21, double * f22, double * f23,
                          double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q27 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values fo
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
in	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
in	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
out	f19	Distribution functions for link 19 at edge lattice site
out	f20	Distribution functions for link 20 at edge lattice site
out	f21	Distribution functions for link 21 at edge lattice site
out	f22	Distribution functions for link 22 at edge lattice site
out	f23	Distribution functions for link 23 at edge lattice site
out	f24	Distribution functions for link 24 at edge lattice site
out	f25	Distribution functions for link 25 at edge lattice site
out	f26	Distribution functions for link 26 at edge lattice site

fD3Q27VCEKinetic()

```
int fD3Q27VCEKinetic (double * p,
                      double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity or density at a concave edge using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom-left concave edges (VCETR) but can be used for any concave edge by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around). Since fluid densities at the boundary point are required, this subroutine can be used for both constant velocity and constant density boundaries.

Parameters

in	p	Fluid densities at concave edge (sampled from nearby lattice point for constant velocity boundaries, fixed values fo
in	v0	Velocity component at concave edge (x-component for bottom-left edge)
in	v1	Velocity component at concave edge (y-component for bottom-left edge)
in	v2	Velocity component at concave edge (z-component for bottom-left edge)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at edge lattice site
in	f1	Distribution functions for link 1 at edge lattice site
in	f2	Distribution functions for link 2 at edge lattice site
in	f3	Distribution functions for link 3 at edge lattice site
in	f4	Distribution functions for link 4 at edge lattice site
out	f5	Distribution functions for link 5 at edge lattice site
in	f6	Distribution functions for link 6 at edge lattice site
in	f7	Distribution functions for link 7 at edge lattice site
in	f8	Distribution functions for link 8 at edge lattice site
in	f9	Distribution functions for link 9 at edge lattice site
in	f10	Distribution functions for link 10 at edge lattice site
in	f11	Distribution functions for link 11 at edge lattice site
out	f12	Distribution functions for link 12 at edge lattice site
out	f13	Distribution functions for link 13 at edge lattice site
out	f14	Distribution functions for link 14 at edge lattice site
out	f15	Distribution functions for link 15 at edge lattice site
in	f16	Distribution functions for link 16 at edge lattice site
out	f17	Distribution functions for link 17 at edge lattice site
out	f18	Distribution functions for link 18 at edge lattice site
out	f19	Distribution functions for link 19 at edge lattice site
out	f20	Distribution functions for link 20 at edge lattice site
out	f21	Distribution functions for link 21 at edge lattice site
out	f22	Distribution functions for link 22 at edge lattice site
out	f23	Distribution functions for link 23 at edge lattice site
out	f24	Distribution functions for link 24 at edge lattice site
out	f25	Distribution functions for link 25 at edge lattice site
out	f26	Distribution functions for link 26 at edge lattice site

fD3Q27VFKinetic()

```
int fD3Q27VFKinetic (long tpos,
                     long rpos,
                     int prop,
                     double * uwall)
```

Applies the appropriate kinetic boundary condition for a constant velocity based on types of collisions and direction for a three-dimensional D3Q27 lattice. (In this case, there are boundary options for cascaded LBE collsions as well as planar surfaces, concave edges and corners.)

Parameters

in	tpos	Position of current boundary lattice site (in one-dimensional form)
in	rpos	Position of neighbouring lattice site (in one-dimensional form) for sampling fluid densities
in	prop	Boundary condition code indicating type and direction
in	uwall	Fixed velocity at boundary site

fD3Q27VPSCLBEKinetic()

```
int fD3Q27VPSCLBEKinetic (double v0, double v1, double v2,
                          double * force,
                          double * f0, double * f1, double * f2,
                          double * f3, double * f4, double * f5,
                          double * f6, double * f7, double * f8,
                          double * f9, double * f10, double * f11,
                          double * f12, double * f13, double * f14,
                          double * f15, double * f16, double * f17,
                          double * f18, double * f19, double * f20,
                          double * f21, double * f22, double * f23,
                          double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q27 lattice and cascaded LBE (CLBE) collisions. This routine can only be used for mildly compressible fluids using the extended local equilibrium distribution functions obtained from CLBE collisions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site
out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site

fD3Q27VPSKinetic()

```
int fD3Q27VPSKinetic (double v0, double v1, double v2,
                      double * force,
                      double * f0, double * f1, double * f2,
                      double * f3, double * f4, double * f5,
                      double * f6, double * f7, double * f8,
                      double * f9, double * f10, double * f11,
                      double * f12, double * f13, double * f14,
                      double * f15, double * f16, double * f17,
                      double * f18, double * f19, double * f20,
                      double * f21, double * f22, double * f23,
                      double * f24, double * f25, double * f26)
```

Determines the required distribution functions to complete a kinetic boundary condition for a fixed fluid velocity at a planar surface using the three-dimensional D3Q27 lattice. This routine can be used for mildly compressible and fully incompressible fluids with the appropriate local equilibrium distribution functions. The expressions in this subroutine are for bottom planar surfaces (VPST) but can be used for any planar surface by selecting different distribution functions and applying positive or negative values for velocity components (which may be swapped around).

Parameters

in	v0	Velocity component tangential to planar surface (x-component for bottom surface)
in	v1	Velocity component orthogonal to planar surface (y-component for bottom surface)
in	v2	Velocity component tangential to planar surface (z-component for bottom surface)
in	force	Forces acting at given boundary lattice point
in	f0	Distribution functions for link 0 at surface lattice site
in	f1	Distribution functions for link 1 at surface lattice site
in	f2	Distribution functions for link 2 at surface lattice site
in	f3	Distribution functions for link 3 at surface lattice site
in	f4	Distribution functions for link 4 at surface lattice site
out	f5	Distribution functions for link 5 at surface lattice site
in	f6	Distribution functions for link 6 at surface lattice site
in	f7	Distribution functions for link 7 at surface lattice site
in	f8	Distribution functions for link 8 at surface lattice site
in	f9	Distribution functions for link 9 at surface lattice site
in	f10	Distribution functions for link 10 at surface lattice site
in	f11	Distribution functions for link 11 at surface lattice site
out	f12	Distribution functions for link 12 at surface lattice site
out	f13	Distribution functions for link 13 at surface lattice site
in	f14	Distribution functions for link 14 at surface lattice site
out	f15	Distribution functions for link 15 at surface lattice site
in	f16	Distribution functions for link 16 at surface lattice site
out	f17	Distribution functions for link 17 at surface lattice site
in	f18	Distribution functions for link 18 at surface lattice site
in	f19	Distribution functions for link 19 at surface lattice site
in	f20	Distribution functions for link 20 at surface lattice site
out	f21	Distribution functions for link 21 at surface lattice site
out	f22	Distribution functions for link 22 at surface lattice site
out	f23	Distribution functions for link 23 at surface lattice site
out	f24	Distribution functions for link 24 at surface lattice site
in	f25	Distribution functions for link 25 at surface lattice site
in	f26	Distribution functions for link 26 at surface lattice site

5.29 lbpUSER.cpp

Module for user-created subroutines and functions.

This is a blank module (along with a header file lbpUSER.hpp) that a user can populate with new subroutines and functions for LBE simulations that expand on the default functionality available in DL_MESO_LBE.

CHAPTER

SIX

DL_MESO_LBE INPUT AND OUTPUT FILE FORMATS

6.1 lbin.sys

This ANSI text input file consists of keywords and values (either numerical or additional keywords) separated by spaces or tabs. The main keywords generally consist of a few words separated by underscore characters (_), which must be specified precisely as shown here to work: no allowances are made by DL_MESO_LBE for typographical errors or abbreviations.

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. If any of the following keywords are not included, DL_MESO_LBE will abort running with an error message.

Table 6.1: Compulsory keywords in lbin.sys input file

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 (N_f) for the system (if modelling solutes, this must be set to 1)
`number_of_solute`	sets the number of solutes (N_c) to be modelled
`temperature_scalar`	determines whether or not a lattice is needed to model heat transfers (N_t, set to 1 if needed, 0 if not)
`phase_field`	determines whether or not a lattice is needed to represent phase fields (N_p, 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_LBE 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 and data type for output files, whether fluids are compressible or incompressible, and whether or not restart files should be used. If these are omitted, DL_MESO_LBE will assume that a new simulation is to take place with compressible fluids subjected to BGK (single-relaxation-time) collisions using standard forcing and no mesophase interactions, producing VTK formatted files in big endian binary. If a customisable version of DL_MESO_LBE are being used, all of these keywords may be omitted except for `incompressible_fluids`, which is required to correctly calculate fluid velocities in initialisation and output files and apply boundary conditions. If all three flags for combining data from processor cores are switched on for three-dimensional calculations (or the x- and y-components are both switched on for two-dimensional systems), MPI-IO will be used to put together data slices in single output files.

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

Table 6.2: Algorithmic keywords in lbin.sys input file

keyword	meaning
`collision_type`	sets the type of collisions and forcing: `BGK` (0), `BGKEDM` (1), `BGKGuo` (2), `TRT` (3), `TRTEDM` (4), `TRTGuo` (5), `MRT` (6), `MRTEDM` (7), `MRTGuo` (8), `CLBE` (9), `CLBEEDM` (10) or `CLBEGuo` (11)[2]
`interaction_type`	sets the type of mesophase interactions[3] : `ShanChen` (1), `ShanChenQuadratic` (2), `Lishchuk` (10), `LishchukSpencer` (11), `LishchukSpencerTensor` (12), `LishchukLocal` (13) or `Swift` (20)
`output_format`	sets the format for output files: `VTK` (0), `LegacyVTK` (1), `Plot3D` (2)
`output_type`	sets the data type for output files (`Binary` (0), `Text` or `ANSI` (1))
`output_combine_x`	combines data from processor cores along the x-axis into single output files (0 = off, 1 = on)
`output_combine_y`	combines data from processor cores along the y-axis into single output files (0 = off, 1 = on)
`output_combine_z`	combines data from processor cores along the z-axis into single output files (0 = off, 1 = on)
`incompressible_fluids`	determines whether or not the fluids should be incompressible (set to 0 for compressible fluids, 1 for incompressible fluids)
`restart_simulation`	determines whether or not the simulation should be restarted using the *lbout.dump* restart file (0 for a new simulation, 1 to restart the simulation)

The following keywords can be used to specify other information, such as fluid densities, velocities, relaxation times or frequencies etc.: further details about possible values for several of these keywords, including default values if they are not supplied, are given in Chapter 6 of the DL_MESO User Manual. For keywords describing boundary conditions, Q in the keyword can be substituted with `top`, `bot`, `lef`, `rig`, `fro`, `bac` for the top, bottom, left, right, front and back boundaries respectively. Superfluous parameters (e.g. boundary velocities when constant density boundaries are in use) can be omitted.

Note that if there are duplicate entries for any keyword, the value associated with the last one in the file will be used.

Table 6.3: Other available keywords in lbin.sys input file

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
`dump_span`	sets interval for writing restart files
`calculation_time`	sets available calculation time (in seconds) before closing down the simulation
`boundary_type`	sets type of boundary conditions for fluid flows
`solute_boundary_type`	sets type of boundary conditions for solutes
`thermal_boundary_type`	sets type of boundary conditions for thermal flows
`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
`trt_magic_number`	sets the TRT 'magic number' Λ_{eo}

continues on next page

[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.

Table 6.3 – continued from previous page

keyword	meaning
`gas_constant`	sets the universal gas constant R for equations of state (Shan-Chen pseudopotential and Swift free-energy interactions)
`gradient_order`	sets the order of gradient approximations (i.e. number of neighbouring grid points used) at boundary/near-boundary points
`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
`oscillating_freq`	sets frequency for sinusoidal oscillating forces across system
`oscillating_period`	sets period (reciprocal of frequency) for sinusoidal oscillating forces across system
`oscillating_freq_`Q	sets frequency for sinusoidal oscillating velocity at boundary Q
`oscillating_period_`Q	sets period (reciprocal of frequency) for sinusoidal oscillating velocity at boundary Q
`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_`Q`_`n	sets constant velocity at boundary Q for all fluids (component n)
`speed_oscil_`Q`_`n	sets oscillating velocity amplitude at boundary Q for all fluids (component n)
`density_ini_`f	sets initial density for fluid f (between 0 and $N_f - 1$) throughout system
`density_inc_`f	sets constant density for incompressible fluid f (or parameter for Shan/Chen 1994 pseudopotential)
`density_`Q`_`f	sets density for fluid f at boundary Q
`rheology_fluid_`f	sets rheology model for fluid f (see below)
`rheology_parameter_a_`f	sets rheological model parameter a for fluid f
`rheology_parameter_b_`f	sets rheological model parameter b for fluid f
`rheology_parameter_c_`f	sets rheological model parameter c for fluid f
`rheology_parameter_d_`f	sets rheological model parameter d for fluid f
`rheology_power_`f	sets rheological model power index n for fluid f
`relaxation_fluid_`f	sets (initial) relaxation time (τ_f) for fluid f (symmetric relaxation time for TRT)
`relax_freq_fluid_`f	sets (initial) relaxation frequency (τ_f^{-1}) for fluid f (symmetric relaxation frequency for TRT)
`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
`mrt_relax_`i	sets ith relaxation time for MRT scheme, applicable for all fluids
`mrt_relax_freq_`i	sets ith relaxation frequency for MRT scheme, applicable for all fluids
`clbe3_relaxation_fluid_`f	sets CLBE third-order relaxation time (τ_3) for fluid f
`clbe3_relax_freq_fluid_`f	sets CLBE third-order relaxation frequency ($\omega_3 = \tau_3^{-1}$) for fluid f
`clbe4_relaxation_fluid_`f	sets CLBE fourth-order relaxation time (τ_4) for fluid f
`clbe4_relax_freq_fluid_`f	sets CLBE fourth-order relaxation frequency ($\omega_4 = \tau_4^{-1}$) for fluid f
`relax_mobility`	sets mobility relaxation time (τ_ϕ) for two-fluid Swift free-energy interactions
`relax_freq_mobility`	sets mobility relaxation frequency (τ_ϕ^{-1}) for two-fluid Swift free-energy interactions
`mobility_parameter`	sets mobility parameter Γ for two-fluid Swift free-energy interactions
`surface_tension_parameter`	sets the surface tension parameter κ for Swift free-energy interactions (both one and two fluid systems)
`solute_ini_`c	sets initial concentration for solute c throughout system (c between 0 and $N_c - 1$)
`solute_`Q`_`c	sets concentration for solute c at boundary Q
`relax_solute_`c	sets relaxation time (τ_c) for solute c
`relax_freq_solute_`c	sets relaxation frequency (τ_c^{-1}) for solute c
`temperature_ini`	sets initial temperature throughout system

continues on next page

Table 6.3 – continued from previous page

keyword	meaning
`temperature_`Q	sets temperature at boundary Q
`temperature_system`	sets temperature of entire system if using equations of state and no temperature scalar
`heating_rate_sys`	sets rate of change in temperature (with time based on real-life units) throughout system
`heating_rate_`Q	sets rate of change in temperature at boundary Q
`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
`oscillating_force_`n	sets amplitude of sinusoidal oscillating body force on fluid f: $n = 3f$ for x-component, $n = 3f + 1$ for y-component, $n = 3f + 2$ for z-component
`oscillating_force_x_`f	sets x-component of amplitude of sinusoidal oscillating body force on fluid f
`oscillating_force_y_`f	sets y-component of amplitude of sinusoidal oscillating body force on fluid f
`oscillating_force_z_`f	sets z-component of amplitude of sinusoidal oscillating 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: $n = N_f \times f_1 + f_2$
`interaction_`f_1`_`f_2	sets interaction parameter between fluids f_1 and f_2
`quadratic_weight`	sets Shan-Chen quadratic term weighting parameter β between all pairs of fluid species
`quadratic_weight_`n	sets Shan-Chen quadratic term weighting parameter β between fluids f_1 and f_2: $n = N_f \times f_1 + f_2$
`quadratic_weight_`f_1`_`f_2	sets Shan-Chen quadratic term weighting parameter β between fluids f_1 and f_2
`potential_type`	sets the pseudopotential type for Shan-Chen interactions (see below) for all fluid species or chemical potential type for Swift free-energy interactions
`potential_type_`f	sets the pseudopotential type for Shan-Chen interactions (see below) for fluid f
`equation_of_state`	sets equation of state for all fluids with Swift free-energy interactions
`eos_parameter_a`	sets equation-of-state parameter a for all fluid species
`eos_parameter_a_`f	sets equation of state parameter a for fluid f
`eos_parameter_b`	sets equation of state parameter b for all fluid species
`eos_parameter_b_`f	sets equation of state parameter b for fluid f
`potential_parameter_a`	sets chemical potential parameter a for all fluid species (Swift free-energy interactions)
`potential_parameter_b`	sets chemical potential parameter b for all fluid species (Swift free-energy interactions)
`shanchen_psi0_`f	sets Shan-Chen pseudopotential parameter ψ_0 for fluid f
`critical_temperature_`f	sets critical temperature T_c for fluid f
`critical_pressure_`f	sets critical pressure P_c for fluid f

continues on next page

Table 6.3 – continued from previous page

keyword	meaning
`acentric_factor_`f	sets acentric factor ω for fluid f
`segregation`	sets fluid segregation parameter between all fluids species
`segregation_`n	sets fluid segregation parameter between fluids f_1 and f_2: $n = N_f \times f_1 + f_2$
`segregation_`f_1`_`f_2	sets fluid segregation parameter between fluids f_1 and f_2
`wetting_type`	sets the basis for wetting interactions between solid points and all fluid species for Shan-Chen interactions or for Swift free-energy interactions
`wetting_type_`f	sets the basis for Shan-Chen wetting interactions between solid points and fluid species f
`wall_interaction_`f	sets Shan-Chen interaction parameter between fluid f and solid walls
`wetting_parameter_rho_`n	sets Swift wetting potential parameter for fluid density (a_n, $n = 0$ or 1)
`wetting_parameter_phi_`n	sets Swift wetting potential parameter for fluid concentration (b_n, $n = 0$ or 1)

This file is compulsory for a DL_MESO_LBE calculation and must be supplied in the same directory where DL_MESO_LBE is run. It can be created or modified by hand using a text editor, but use of the DL_MESO GUI for this file is recommended, particularly when starting to use DL_MESO_LBE.

6.2 lbin.spa

This ANSI text input file consists of lines, each of which includes the Cartesian coordinates of a grid point and a boundary code for that grid point, all separated by white space (spaces or tab characters), i.e.

```
x y z [boundary code]
```

DL_MESO_LBE assumes a boundary code of 0 for all grid points by default (representing fluid sites), which gives periodic boundaries for points at the outer edges of a lattice. Other categories of boundary condition can be obtained by using different values, as shown in Table 6.4.

Table 6.4: Boundary condition categories

value	meaning
0	liquid
10	domain boundary
11	inside solid (blank site)
12	on-grid bounce back boundary
13	mid-link bounce back boundary
21–99	outflow boundary
100–199	constant velocity, composition and temperature boundary
200–299	constant velocity, Neumann composition and temperature boundary
300–399	constant velocity and composition, Neumann temperature boundary
400–499	constant velocity and temperature, Neumann composition boundary
500–599	constant pressure (density), composition and temperature boundary
600–699	constant pressure (density), Neumann composition and temperature boundary
700–799	constant pressure (density) and composition, Neumann temperature boundary
800–899	constant pressure (density) and temperature, Neumann composition boundary

In the case of outflow and constant velocity/density boundary conditions, the value of the boundary code also incorporates the direction in which the condition acts (i.e. the location of the nearest fluid point relative to the boundary grid point), which is given in the last two digits. To simplify understanding of these boundary codes, words of up to eight letters in length can be used to describe a boundary condition. Table 6.5 includes the categories for boundary condition words, which have the letters given in the following order:

1. Type of boundary condition: either outflow or a combination of fluid, solute and temperature properties

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

For example, a shearing planar surface facing downwards along the y-axis with constant composition and temperature (i.e. isothermal) is represented as VCBPSD. (More details of the available boundary condition codes and words can be found in Chapter 6 of the DL_MESO User Manual.)

Table 6.5: Boundary condition categories

letter	meaning
O	Outflow
V	Constant Velocity
P	Constant Pressure (Density)
C	Constant Solute Composition
T	Constant Temperature
B	Bounce-back Boundary Condition (Solute Composition or Temperature)
PS	Planar Surface
CE	Concave Edge
CC	Concave Corner
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

The directional part of the words used to describe boundary conditions (the last letter or letters) are deciphered by the enumeration array *BoundaryType* in *lbe.hpp*. These are used directly in the modules for boundary condition schemes to identify the directions given by boundary codes and apply the subroutines using the correct orientation.

This file is compulsory for a DL_MESO_LBE calculation - even if all boundaries are intended to be periodic, a blank *lbin.spa* file must be supplied - and must be in the same directory where DL_MESO_LBE is run. Use of the DL_MESO GUI to create this file is highly recommended, especially for simulations with large grids and/or many non-fluid points (e.g. for porous media).

6.3 lbin.init

This ANSI text input file consists of lines, each of which includes the Cartesian coordinates of a grid point and all the macroscopic properties that can be specified - velocity, fluid densities, solute concentrations and temperature - all separated by white space (spaces or tab characters):

```
x y z u_x u_y u_z rho_0 ... c_0 ... T
```

The property values specified at each grid point subsequently replace the default values given in the *lbin.sys* file (given with the keywords `speed_ini`, `density_ini`, `solute_ini` and `temperature_ini`) that ordinarily apply to all grid points in the lattice. These values are then used to calculate replacement distribution functions for the specified grid points, thus providing users with a way to initialise LBE simulations.

Note that only the properties needed for the simulation (as defined by the numbers of fluids, solutes and temperature fields) need to be specified in each line of this file, although *all* of these properties need to be included.

[4] When defining boundary conditions for two-dimensional simulations, only front-facing concave edges and corners are available.

Incorrect numbers of values per line might cause DL_MESO_LBE to crash while attempting to read this file or initialise the simulation in an unexpected way.

This file is entirely optional for a DL_MESO_LBE calculation, but if it is to be used, it must be in the directory where DL_MESO_LBE is launched. It can be created by hand, although the use of a utility is recommended: *lbeinitcreate.cpp* can create one for a new simulation and use the *lbin.sys* file to specify default fluid densities, concentrations and temperature, while *lbedumpinit.cpp* can create a file from a *lbout.dump* simulation restart file.

6.4 lbout.dump

This binary file consists of information required for DL_MESO_LBE to restart and extend a LBE simulation, which can be read in if the value for `restart_simulation` in *lbin.sys* is set to 1.

The file consists of a header made up of 12 integers with the following properties:

- The number of dimensions
- The number of lattice speeds
- The number of grid points in the x-dimension
- The number of grid points in the y-dimension
- The number of grid points in the z-dimension
- The number of fluids (N_f)
- The number of solutes (N_c)
- The switch for temperature field (N_t)
- The (currently unused) switch for phase field (N_p)
- The timestep at the point when the file was created
- The number of snapshot output files previously written
- The switch for incompressible fluids

If incompressible fluids were in use, the constant densities ρ_0 used for the calculation then follow as double precision floating-point numbers. (These are not included if compressible fluids were used.)

A block of integers then follows, consisting of triples with Cartesian coordinates (x, y and z) for all of the grid points in the calculation. These are not necessarily written to the file in any particular order, but their ordering corresponds to the data written for each grid point that follows this integer block: this enables DL_MESO_LBE to determine which set of data belongs to which grid point and (for the parallel version) whether or not the grid point exists in a processor core's lattice subdomain.

The data for the grid points given in the order indicated by the previous block of integers then follow as a block of double precision floating-point numbers. For each grid point, the distribution functions - in the same order as stored in the *lbf* array (in blocks ordered by lattice link, with each block giving the distribution functions for the various fluids, solutes and temperature field) - and the relaxation frequencies for all fluids are provided, all as double precision numbers. These enable DL_MESO_LBE to read in the data for each grid point and copy it directly into the corresponding arrays.

This file is automatically generated by DL_MESO_LBE during a simulation and either at the very end once all the timesteps have been completed or when the calculation time specified in *lbin.sys* has run out. The endianness (big or small) of the file corresponds to that of the computer used to run the calculation: it can be used to restart the simulation on another machine provided the two computers have the same endianness. The *lbedumpinit.cpp* utility can be used to generate a *lbin.init* file from this file as a starting point for a new simulation, while *lbedumpvtk.cpp* can generate an XML-formatted Structured Grid VTK file to visualise the system at the point when this file was created.

6.5 lbout*.vts

By default, DL_MESO_LBE will generate a series of these files in XML-based Structured Grid VTK format[5] containing snapshots of the simulation with macroscopic properties (grid point locations given in 'real-life' units, boundary codes, velocities, fluid densities, mass fractions for fluids, solute concentrations, temperature) at each grid point every *lbsave* timesteps after any equilibration timesteps (*lbequstep*). With the exception of the boundary codes, all of the data are provided as single-precision floating-point numbers.

Two options for this format exist in DL_MESO_LBE: big endian binary (default) and ANSI text. Both use XML tags - `<DataArray>` - to define the data sets (including their unique names), although while the text version of the file puts the values directly between the relevant tags with spaces between the values, the binary version uses the tags to indicate the starting location of the data in a stream of binary numbers inside an `<AppendedData>` tag.

If running DL_MESO_LBE in serial (or using MPI-IO in parallel to combine data), a single file per snapshot is produced with the name `lbout`*yyyyyy* `.vts`, where *yyyyyy* is the snapshot number (starting from `000000`). By default for a parallel run of DL_MESO_LBE, each processor core will generate its own file for each snapshot with the name `lbout`*xxxxxx*`at`*yyyyyy*`.vts`, where *xxxxxx* is the processor core number. The number of files per snapshot can be reduced by using the `output_combine` keywords in the *lbin.sys* file: switching on each of these options combines data from processor cores along the given Cartesian (x, y or z) axis. If these options are selected for all dimensions in the simulation, DL_MESO_LBE gathers the data in all but one dimension and then uses MPI-IO to write each data group concurrently and contiguously to a single file for the snapshot. A parallel run of DL_MESO_LBE will also produce a *lbout.info* file and, if multiple files per snapshot are produced, a *lbout.ext* file to provide the grid extents covered by each file (each value of *xxxxxx*): this can be used by the *lbevtkgather.cpp* utility to generate Parallel Structured Grid XML VTK files (`lbtout`*yyyyyy*`.pvts`) that can be opened in Paraview and used to pull together data from multiple files per snapshot. (Note that the original *.vts files need to be retained as the parallel files are merely used to link these together.)

6.6 lbout*.vtk

If selected in the *lbin.sys* input file, DL_MESO_LBE will generate a series of these files in Legacy Structured Grid VTK format[5] containing snapshots of the simulation with macroscopic properties (grid point locations given in 'real-life' units, boundary codes, velocities, fluid densities, mass fractions for fluids, solute concentrations, temperature) at each grid point every *lbsave* timesteps after any equilibration timesteps (*lbequstep*). With the exception of the boundary codes, all of the data are provided as single-precision floating-point numbers.

Two options for this format exist in DL_MESO_LBE: big endian binary (default) and ANSI text. Both formats consist of text lines (headers) indicating the data to follow, which is then written either as single values for scalars (e.g. densities, concentrations) or as triples for vector properties (velocities, grid coordinates): in the case of text formatting, each line following the header represents an individual grid point.

If running DL_MESO_LBE in serial (or using MPI-IO in parallel to combine data), a single file per snapshot is produced with the name `lbout`*yyyyyy* `.vtk`, where *yyyyyy* is the snapshot number (starting from `000000`). By default for a parallel run of DL_MESO_LBE, each processor core will generate its own file for each snapshot with the name `lbout`*xxxxxx*`at`*yyyyyy*`.vtk`, where *xxxxxx* is the processor core number. The number of files per snapshot can be reduced by using the `output_combine` keywords in the *lbin.sys* file: switching on each of these options combines data from processor cores along the given Cartesian (x, y or z) axis. If these options are selected for all dimensions in the simulation, DL_MESO_LBE gathers the data in all but one dimension and then uses MPI-IO to write each data group concurrently and contiguously to a single file for the snapshot. A parallel run of DL_MESO_LBE will also produce a *lbout.info* file and, if multiple files per snapshot are produced, a *lbout.ext* file to provide the grid extents covered by each file (each value of *xxxxxx*). To date, no utility has yet been written to combine or link together multiple Legacy VTK files per snapshot.

[5] Full details of this file format can be found in File Formats for VTK Version 4.2.

6.7 lbout*.q

If selected in the *lbin.sys* input file, DL_MESO_LBE will generate a series of these files in Plot3D format[6] containing snapshots of the simulation with macroscopic properties (grid point locations given in 'real-life' units, boundary codes, velocities, fluid densities, mass fractions for fluids, solute concentrations, temperature) at each grid point every *lbsave* timesteps after any equilibration timesteps (*lbequstep*). All of the data, including boundary codes, are provided as single-precision floating-point numbers.

The format for Plot3D solution files (*.q) starts with three integers providing the extent of the lattice covered by the file, followed by four single-precision floating-point numbers with:

- Fluid 0's speed of sound in real-life units (*lbsoundv*)
- The freestream angle of attack (always given as 1)
- The flow Reynolds number (always set to 0)
- The timestep number less the number of equilibration steps (converted to a floating point number)

These are then followed by blocks of floating-point numbers:

- The property being written to the file (a fluid density, a fluid mass fraction, a solute concentration, temperature)
- x-components of velocity
- y-components of velocity
- z-components of velocity (only for three-dimensional simulations)
- Boundary codes (converted to floating point numbers)

for the available grid points. It should be noted that only a single property can be written to each file, and as such DL_MESO_LBE (by default) writes a series of Plot3D solution files for each property modelled in the simulation, starting with:

- lbout*zz*dens - for densities of fluid *zz*
- lbout*zz*frac - for mass fractions of fluid *zz*
- lbout*zz*conc - for concentrations of solute *zz*
- lbouttemp - for temperatures

The locations of the grid points are written to Plot3D grid files (*.xyz for three-dimensional simulations, *.xy for two-dimensional simulations), which start with three integers providing the extent of the lattice covered by the file (the numbers of points in each Cartesian direction), followed by blocks of floating-point numbers:

- x-components of the grid point locations (in real-life units)
- y-components of the grid point locations (in real-life units)
- z-components of the grid point locations (in real-life units, only for three-dimensional simulations)

Both solution and grid files can either be written in big endian binary (default) or in ANSI text formats: in the case of text formatting, spaces are placed between values in data blocks (e.g. lattice extent, property values) and a carriage return follows each block.

If running DL_MESO_LBE in serial (or using MPI-IO in parallel to combine data), a single solution file per snapshot per property is produced. For the example of the density of fluid 0, these files are named lbout00dens*yyyyyy*.q, where *yyyyyy* is the snapshot number (starting from 000000). Similarly, a single grid file called lbout.xyz or lbout.xy is produced at the start of the simulation.

By default for a parallel run of DL_MESO_LBE, each processor core will generate its own file for each snapshot and property, e.g. for densities of fluid 0, these will be called lbout00dens*xxxxxx*at*yyyyyy*.q, where *xxxxxx* is the processor core number. A series of grid files called lbout*xxxxxx*.xyz or lbout*xxxxxx*.xy will also be created at the start of the simulation. The number of files per snapshot and the number of grid files can be reduced by using

[6] Some details of this file format, given as Fortran code, can be found here.

the `output_combine` keywords in the *lbin.sys* file: switching on each of these options combines data from processor cores along the given Cartesian (x, y or z) axis. If these options are selected for all dimensions in the simulation, DL_MESO_LBE gathers the data in all but one dimension and then uses MPI-IO to write each data group concurrently and contiguously to a single file for the snapshot. A parallel run of DL_MESO_LBE will also produce a *lbout.info* file and, if multiple files per snapshot are produced, a *lbout.ext* file to provide the grid extents covered by each file (each value of *xxxxxx*). The *lbout.info* file can be used by the *lbeplot3dgather.cpp* utility to create combined grid (lbtout.xyz or lbtout.xy) and solution (lbtout**yyyyyy*.q) files that pull together the data from the original files and can be opened in Paraview. (Since these combined files contain all the data, the files originally created by DL_MESO_LBE can later be deleted.)

6.8 lbout.info

When running DL_MESO_LBE in parallel, this small ANSI text file is produced to provide some of the post-processinig utilities the necessary information to pull simulation data together from multiple files per snapshot. It consists of the following keywords with integer values for each described property on the same line, separated by white space:

```
numberofDimensions
numberofFluids
numberofSolutes
numberofTemperature
sizeofSystem
sizeofInteger
sizeofFloat
```

The numbers of dimensions, fluids, solutes and temperature are specific to the system being simulated (and are originally provided in *lbin.sys*), the size of the system here gives the number of files per snapshot (either the total number of processor cores or the number of I/O groups used to gather and write data to snapshot files), while the sizes of integers and (single-precision) floats are given in bytes and are equal to the standard sizes used in C++.

This file is used by the *lbeplot3dgather.cpp* and *lbevtkgather.cpp* utilities to determine information needed to gather together Plot3D and XML-based Structured Grid VTK files respectively.

6.9 lbout.ext

When running DL_MESO_LBE in parallel and generating multiple files per snapshot (i.e. when not using MPI-IO), this small ANSI text file is produced to provide the extents of each lattice subdomain (either for individual processor cores or for I/O groups gathering data together). The file consists of lines for each core or I/O group identified by a number in the format `extent_`*xxxxx* (where *xxxxxx* is the core or group number) and followed by the minimum and maximum cordinates for each dimension of the subdomain for the core or group, with white space between each value, e.g.

```
extent_000000   0 50 0 50 0 50
extent_000001   50 100 0 50 0 50
extent_000002   0 50 50 100 0 50
extent_000003   50 100 50 100 0 50
extent_000004   0 50 0 50 50 100
extent_000005   50 100 0 50 50 100
extent_000006   0 50 50 100 50 100
extent_000007   50 100 50 100 50 100
```

This file is used by the *lbevtkgather.cpp* utility to determine the extent of each file written per simulation snapshot and how the files fit together, which is subsequently written to the parallel VTK files linking the data together. (If MPI-IO is in use, this file is not produced as the data files do not need to be processed after the simulation has finished.)

CHAPTER

SEVEN

ADVICE ON DEVELOPING DL_MESO_LBE

DL_MESO_LBE has been written to allow users to either use the code as-is for their LBE simulations and/or to expand the code to implement their new functionalities and run simulations in a highly-scalable manner. In order to expand upon DL_MESO_LBE's feature set, this chapter provides some advice on what changes need to be made to the code and how these could be carried out.

7.1 User module: lbpUSER.cpp

A blank code module for DL_MESO_LBE - *lbpUSER.cpp* - has been provided for users to place their own subroutines and functions. This module and its header file (lbpUSER.hpp) are automatically linked into the header files for the parallel and serial versions of DL_MESO_LBE, *plbe.hpp* and *slbe.hpp*: as these are used by both the mainline and customisable versions of the codes, this enables user-developers to both test out their new features in a streamlined code and to incorporate their own new features into their own copy of the main code (both parallel and serial versions) later on. Keeping these changes separate from the main modules (at least to begin with) also enables the user-developer to take immediate advantage of any bug fixes subsequently made to the main code without requiring major changes to their own subroutines or functions.

7.2 Use of customisable codes

The customisable codes supplied with DL_MESO_LBE - *plbecustom.cpp, slbecustom.cpp and slbecombine.cpp* - are effectively stripped down versions of the main code with the main calculation loop incorporated in them (as opposed to the separate modules with these loops in the mainline codes). These codes allow the user to 'hardcode' their new subroutines and functions as a temporary measure to check whether and how they work, bypassing the need to add new options and/or keywords to the various input files (which can be carried out later).

Both main custom codes - plbecustom.cpp and slbecustom.cpp - are initially set up to carry out the main parts required for a lattice Boltzmann equation simulation. These include reading input files, setting up arrays, calculating initial distribution functions etc. at the start, the main calculation loop with options for file writing, calculating interaction forces (given as Shan-Chen but with comments on how to use other interaction types) and applying BGK collisions, and closing down the simulation at the end. Unlike the main codes, no check on calculation time is made during the main loop to close down the simulation before all of the specified timesteps have been completed.

In the case of the parallel custom code (plbecustom.cpp), calls to communication subroutines have been included at the appropriate points to ensure data in the boundary halo is available when they are needed (e.g. distribution functions before interaction force calculations, interaction forces before collisions). If the user-developer wishes to try out calculations with boundary halos on a smaller scale, possibly before coding up MPI communication routines, the serial custom code with boundary halos (slbecombine.cpp) can be used to check when communications would be needed for parallel running.

7.3 New collision operators

The user-developer is advised to follow a similar structure to the pre-existing collison modules - *lbpBGK.cpp*, *lbpTRT.cpp*, *lbpMRT.cpp* and *lbpCLBE.cpp* - when creating their own collision subroutines. This involves two types of subroutine:

- Collisions on a single lattice site (e.g. *fSiteFluidCollisionBGK()*), taking in the pointer for the first distribution function in *lbf* and calculated properties for the grid point (e.g. velocity, fluid relaxation times and densities, forces)
- Loops over all grid points (e.g. *fCollisionBGK()*), calculating properties and calling the site collision subroutine for each grid point

The latter subroutine is the one that can exploit OpenMP multithreading and include options for compressible and/or incompressible fluids.

7.4 New interaction forces

The general approach taken in *lbpFORCE.cpp* for calculating interaction forces on (and between) fluids based on gradients of a given property is advisable when devising new interaction models. This includes:

- Subroutines to calculate gradients or gradient-based properties at a single grid point
  - Separate versions for grid points away from and near edges of lattice subdomains, latter requiring modulo functions to find neighbouring points beyond periodic boundaries
- Subroutines to run through grid points and calculate gradients or related properties
  - Version of this subroutine for parallel running only needs to look at grid points away from edge of lattice subdomain, i.e. those not inside the boundary halo
  - Version of this subroutine for serial running needs two separate loops: one for grid points at edge of lattice subdomaiin, one for grid points away from edge

It should be noted that the parallel running version of the subroutine to calculate gradients or related properties at all grid points inside the subdomain (excluding the boundary halo) would need to be followed by a communication subroutine to copy values into the boundary halo. This is particularly important for interaction force calculations, as these values will be used for subsequent collisions.

If the user-developer wishes to try out e.g. a new pseudopotential for Shan-Chen interactions, they can start by creating their own custom copy of *fCalcPotential_ShanChen()* but removing the `switch` block inside the loop for fluids and directly calculating their new pseudopotential for each lattice site and fluid. After testing, the user-developer can later add their pseudopotential as a new option inside *fCalcPotential_ShanChen()* itself by copying and modifying a `case` block. A similar approach can be taken for e.g. new wall-fluid (wetting) interaction types.

7.5 New rheological models

If the user-developer wishes to add a new rheological model, this can be added to the *fGetRelaxationFrequency()* subroutine in the *lbpRHEOLOGY.cpp* module, either directly or in a modified copy of the subroutine that can be called by the customisable version of the code. Unless a new collision operator has also been added (necessitating new subroutines to calculate shear rates), no other major modifications need to be made to this module.

7.6 New boundary conditions

The approach to coding up a new boundary condition scheme will depend on whether or not the scheme has to be applied in specific directions. If it does not or only a limited number of directions need to be considered, a similar approach to bounce back boundary conditions can be taken, e.g. *fMidBounceBackF()*: a single subroutine with a one-dimensional grid coordinate number as an input can be written to apply the condition either in all or in a limited number of directions.

If direction needs to be specified for the boundary condition, a similar approach to the main schemes (e.g. Zou-He, Inamuro) is recommended. This takes the form of coding up each type of boundary (planar surface, concave edge, concave corner) for a single direction (e.g. *fD3Q15VPSZouHe()* etc.) and using this implementation for all other directions by selecting different permutations of distribution functions, velocity components etc. (e.g. *fD3Q15VFZouHe()*). The selected boundary directions used in the existing schemes (i.e. upwards from bottom planar surface, upwards-rightwards from bottom-left concave edge, upwards-rightwards-forwards from bottom-left-back concave corner) is highly recommended to avoid needing to work out a different set of permutations.

It should be noted for direction-based boundary schemes that:

1. each lattice scheme (e.g. D2Q9, D3Q15) requires its own implementation of the new boundary condition scheme, and
2. different versions may be required for each lattice scheme if the boundary scheme involves local equilibrium distribution functions: in DL_MESO_LBE, these will include variations for incompressible fluids, Swift free-energy interactions and cascaded LBE collisions.

If the user-developer requires more directional boundary types, e.g. convex edges and corners, additional boundary codes will need to be added to *BoundaryType* to identify the directions as well as subroutines to implement one direction for each type and additional calls in subroutines for lattice schemes to implement them in different directions. This is a more substantial modification to make, particularly if the new directions need to be implemented for all the current boundary condition schemes.

7.7 New lattice model

DL_MESO_LBE is designed to be as agnostic as possible when it comes to lattice schemes, i.e. very few subroutines require specific versions for different lattices, and all arrays with lattice-dependent parameters (e.g. *lbw*) can be used regardless of the lattice scheme in use. The subroutines in *lbpMODEL.cpp* are used to define values for the following lattice-dependent variables and arrays:

- *lbcs*, *lbcssq* and *lbrcssq* - lattice speed of sound (c_s), its square and the reciprocal of the square
- *lbw* - lattice link weighting parameters w_i used for local equilibrium distribution function calculations and calculating gradients for interaction forces
- *lbwi*, *lbw0*, *lbwpt*, *lbwxx*, *lbwyy*, *lbwzz*, *lbwxy*, *lbwxy*, *lbwyz*, *lbwgam* and *lbwdel* - lattice link weighting parameters for local equilibrium distribution functions with Swift free-energy interactions (if model is available for lattice scheme)
- *lbvx*, *lbvy* and *lbvz* - lattice link vectors $\hat{e}_i$
- *lbfevx*, *lbfevy* and *lbfevz* - lattice vectors for gradient calculation stencil used in Swift free-energy interactions (if applicable)
- *lbopv* - conjugate lattice links
- *lbvwx*, *lbvwy* and *lbvwz* - product of lattice link vectors and weighting parameters (i.e. product of *lbw* and *lbvx* etc.), used in gradient calculations
- *lbtr* - moment tranformation matrix $\mathbf{T}$ for Multiple Relaxation Time (MRT) collisions (if available for lattice scheme)
- *lbtrinv* - inverse of moment tranformation matrix $\mathbf{T}^{-1}$ for MRT collisions (if available)
- *lbmrtw* - moment parameters for some MRT collision schemes

The main substantial modifications that might be necessary for new lattice schemes could include:

- New local equilibrium distribution functions (in place of e.g. *fGetEquilibriumF()*) if the new lattice is not square
- Changing the arrays *lbvx*, *lbvy* and *lbvz* from integers to double precision floating-point if the link vectors are no longer integers (e.g. when using triangular lattices)

7.8 New or modified output file format

The simplest type of new output file to create is one that reports on system-wide values of a property. Noting that obtaining these values in parallel running might require a global communication step, e.g. summation over all processor cores using *fGlobalValue()*, the results can be printed to a file by a single processor core. By convention and to ensure compatibility for serial running, this core should be numbered 0. (Examples of this approach can be seen in *fPrintSystemMass()* and *fPrintSystemMomentum()*.)

If an additional grid-point-dependent property needs to be reported in the snapshot output files currently created by DL_MESO_LBE, much of this can be achieved by creating new subroutines similar to those found in *lbpIOAGGPAR.cpp and lbpIOAGGSER.cpp* and *lbpGET.cpp*:

- A subroutine to calculate the property for a single lattice site, e.g. *fGetOneMassSite()* for a single fluid density
- A subroutine to wrap the above subroutine calculating the property at the lattice site, e.g. *fGetOneMassSiteWrap()*
- A subroutine to put together values of the property into an array for each processor core by calling *fFillBuffer()* with the above wrapped subroutine as an input, e.g. *fPieceDensities()*
- A subroutine that gathers values of the property from all processor cores in the current I/O group into a single array for the group's root core, calling *fGroupGatherFloatData()* or *fGroupGatherIntData()* using the piece subroutine as an input, e.g. *fGroupDensities()*

The last subroutine here is then called by the subroutine creating the output file itself, before it goes on to write the data to the file. Taking the above example of fluid densities, for XML-based VTK files, the subroutine *fOutputVTK()* calls *fGroupDensities()* for each fluid before calling *fWriteVTKFloatBinaryData()* to add it to the output file. (Note that if the property can be calculated directly from values already stored in memory, only the piece and group subroutines are needed as the former would be able to fill the core's buffer directly.)

A new file format for grid-based data can be implemented by taking the existing file formats and their corresponding modules - *lbpIOVTK.cpp*, *lbpIOLegacyVTK.cpp* and *lbpIOPlot3D.cpp* - and using the same principles of gathering data among I/O groups and writing the data to either a single file (in serial or using MPI-IO in parallel) or an individual file per I/O group.

7.9 Modifications to input file reading

If the new feature has been tested and the user-developer wishes to implement it more fully into DL_MESO_LBE, they will need to make modifications to reading input files - most frequently *lbin.sys* - to read in options and parameters for the new feature. The main subroutine to modify in this case is *fInputParameters()*, which reads keywords and values in individual lines.

Once the user-developer has devised the new keywords for the *lbin.sys* file, they can then add them to the main `while` loop going through each line of the file. If the value can be given as a string (or word), this needs to be used immediately after reading the value to convert it into a number that can be assigned to an array or variable.

When comparing strings (either the word or the value) with possible values, the C++ `compare` function can be used, although care should be taken in its use: the inclusion of integers in the call indicating the starting character and length to compare strings can give a match even if the string being tested is longer. For keywords with variations that extend to allow specific circumstances to be specified, if a default value is meant to be applied for the unextended keyword, the starting character and string length for comparison must *not* be included in the call for `compare` to ensure only an exact match can be detected.

CHAPTER

EIGHT

LATTICE SCHEMES

This chapter gives details of the various lattice schemes implemented in DL_MESO_LBE code: D2Q9, D3Q15, D3Q19 and D3Q27. These include details of the defined lattice vectors, weight factors used in local equilibrium distribution functions (including those for Swift free-energy interactions), the transformation matrices, moments, relaxation frequencies and forcing terms used for MRT and cascaded LBE collisions.

8.1 D2Q9

Table 8.1: Lattice vectors for D2Q9

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

Table 8.2: Weight factors for D2Q9

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

Table 8.3: Swift free-energy weight factors for D2Q9

i	w_i	w_i^{00}	γ_i	δ_i	w_i^p	w_i^t	w_i^{xx}	w_i^{yy}	w_i^{zz}	w_i^{xy}	w_i^{xz}	w_i^{yz}
0	$\frac{4}{3}$	1	0	$-\frac{21}{8}$	$-\frac{5}{3}$	$-\frac{5}{3}$	$-\frac{1}{6}$	$-\frac{1}{6}$	0	0	0	0
1,5	$\frac{1}{12}$	0	$\frac{3}{2}$	$-\frac{3}{2}$	$\frac{1}{12}$	$\frac{1}{12}$	$-\frac{1}{24}$	$-\frac{1}{24}$	0	$-\frac{1}{4}$	0	0
2,6	$\frac{1}{3}$	0	$\frac{3}{2}$	$-\frac{3}{2}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	$-\frac{1}{6}$	0	0	0	0
3,7	$\frac{1}{12}$	0	$\frac{3}{2}$	$-\frac{3}{2}$	$\frac{1}{12}$	$\frac{1}{12}$	$-\frac{1}{24}$	$-\frac{1}{24}$	0	$\frac{1}{4}$	0	0
4,8	$\frac{1}{3}$	0	$\frac{3}{2}$	$-\frac{3}{2}$	$\frac{1}{3}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{3}$	0	0	0	0

8.1.1 Multiple relaxation time scheme

Definition of transformation matrix based on [73]:

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

For the standard local equilibrium distribution functions, the equilibrium moments are expressed for incompressible fluids as:

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

where ρ is used in place of ρ_0 for mildly compressible fluids, $j_x = \rho_0 u_x$, $j_y = \rho_0 u_y$ and, by default, $w_\epsilon = 1$ and $w_{\epsilon j} = -3$. If using Swift free-energy interactions, the equilibrium moments are expressed as:

$$\vec{M}^{eq} = \begin{pmatrix} \rho \\ e^{eq} \\ \epsilon^{eq} \\ j_x \\ q_x^{eq} \\ j_y \\ q_y^{eq} \\ p_{xx}^{eq} \\ p_{xy}^{eq} \end{pmatrix} = \begin{pmatrix} \rho \\ -4\rho + \frac{3}{\rho}\left(j_x^2 + j_y^2\right) + 6(P_0 - \kappa(\rho\nabla^2\rho + \phi\nabla^2\phi)) + 15\frac{\lambda}{\rho}(\vec{p}\cdot\nabla\rho) \\ 4\rho - \frac{3}{\rho}(j_x^2 + j_y^2) - 9(P_0 - \kappa(\rho\nabla^2\rho + \phi\nabla^2\phi)) - \frac{3}{2}\kappa\left(|\nabla\rho|^2 + |\nabla\phi|^2\right) - \frac{33\lambda}{2\rho}(\vec{p}\cdot\nabla\rho) \\ j_x \\ -j_x \\ j_y \\ -j_y \\ \frac{j_x^2 - j_y^2}{\rho} + \kappa((\partial_x\rho)^2 - (\partial_y\rho)^2 + (\partial_x\phi)^2 - (\partial_y\phi)^2) + 2\frac{\lambda}{\rho}(p_x\partial_x\rho - p_y\partial_y\rho) \\ \frac{j_x j_y}{\rho} + \kappa(\partial_x\rho\partial_y\rho + \partial_x\phi\partial_y\phi) + \frac{\lambda}{\rho}(p_x\partial_y\rho + p_y\partial_x\rho) \end{pmatrix}$$

The relaxation frequencies for the above moments can be expressed by the following diagonal matrix:

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

where the bulk viscosity can be related to the associated relaxation time by:

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

Recommended default values for the two variable relaxation frequencies s_2 and s_4 are 1.14 and 1.92 respectively for standard simulations, while both can be set to 1 for simulations with Swift free-energy interactions [103].

Guo forcing can be applied using the following moment transformations of the associated source terms:

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

and He forcing can be applied using these moment terms:

$$\vec{S}^m = \begin{pmatrix} 0 \\ 6(v_x F_x + v_y F_y) \\ -6(v_x F_x + v_y F_y) \\ F_x \\ -F_x(1-3v_y^2) + 6v_x v_y F_y \\ F_y \\ -F_y(1-3v_x^2) + 6v_x v_y F_x \\ 2(v_x F_x - v_y F_y) \\ v_x F_y + v_y F_x \end{pmatrix}.$$

8.1.2 Cascaded LBE scheme

Definitions of transformation and shift matrices based on [35]:

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

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -u_x & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -u_y & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ u_x^2 & -2u_x & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ u_y^2 & 0 & -2u_y & 0 & 1 & 0 & 0 & 0 & 0 \\ u_x u_y & -u_y & -u_x & 0 & 0 & 1 & 0 & 0 & 0 \\ -u_x^2 u_y & .2u_x u_y & u_x^2 & -u_y & 0 & -2u_x & 1 & 0 & 0 \\ -u_x u_y^2 & u_y^2 & 2u_x u_y & 0 & -u_x & -2u_y & 0 & 1 & 0 \\ u_x^2 u_y^2 & -2u_x u_y^2 & -2u_x^2 u_y & u_y^2 & u_x^2 & 4u_x u_y & -2u_y & -2u_x & 1 \end{bmatrix}.$$

The equilibrium central moments are expressed as follows:

$$\vec{\tilde{M}}^{eq} = \begin{pmatrix} \tilde{M}_{00}^{eq} \\ \tilde{M}_{10}^{eq} \\ \tilde{M}_{01}^{eq} \\ \tilde{M}_{20}^{eq} \\ \tilde{M}_{02}^{eq} \\ \tilde{M}_{11}^{eq} \\ \tilde{M}_{21}^{eq} \\ \tilde{M}_{12}^{eq} \\ \tilde{M}_{22}^{eq} \end{pmatrix} = \begin{pmatrix} \rho \\ 0 \\ 0 \\ \frac{1}{3}\rho \\ \frac{1}{3}\rho \\ 0 \\ 0 \\ 0 \\ \frac{1}{9}\rho \end{pmatrix}$$

and transformation of the above leads to the following expressions for the local equilibrium distribution functions:

$$\begin{aligned}
f_0^{eq} &= \frac{4}{9}\rho - \frac{2}{3}\rho u_x^2 - \frac{2}{3}\rho u_y^2 + \rho u_x^2 u_y^2 \\
f_1^{eq} &= \frac{1}{36}\rho - \frac{1}{12}\rho u_x + \frac{1}{12}\rho u_y + \frac{1}{12}\rho u_x^2 + \frac{1}{12}\rho u_y^2 - \frac{1}{4}\rho u_x u_y + \frac{1}{4}\rho u_x^2 u_y - \frac{1}{4}\rho u_x u_y^2 + \frac{1}{4}\rho u_x^2 u_y^2 \\
f_2^{eq} &= \frac{1}{9}\rho - \frac{1}{3}\rho u_x + \frac{1}{3}\rho u_x^2 - \frac{1}{6}\rho u_y^2 + \frac{1}{2}\rho u_x u_y^2 - \frac{1}{2}\rho u_x^2 u_y^2 \\
f_3^{eq} &= \frac{1}{36}\rho - \frac{1}{12}\rho u_x - \frac{1}{12}\rho u_y + \frac{1}{12}\rho u_x^2 + \frac{1}{12}\rho u_y^2 + \frac{1}{4}\rho u_x u_y - \frac{1}{4}\rho u_x^2 u_y - \frac{1}{4}\rho u_x u_y^2 + \frac{1}{4}\rho u_x^2 u_y^2 \\
f_4^{eq} &= \frac{1}{9}\rho - \frac{1}{3}\rho u_y - \frac{1}{6}\rho u_x^2 + \frac{1}{3}\rho u_y^2 + \frac{1}{2}\rho u_x^2 u_y - \frac{1}{2}\rho u_x^2 u_y^2 \\
f_5^{eq} &= \frac{1}{36}\rho + \frac{1}{12}\rho u_x - \frac{1}{12}\rho u_y + \frac{1}{12}\rho u_x^2 + \frac{1}{12}\rho u_y^2 - \frac{1}{4}\rho u_x u_y - \frac{1}{4}\rho u_x^2 u_y + \frac{1}{4}\rho u_x u_y^2 + \frac{1}{4}\rho u_x^2 u_y^2 \\
f_6^{eq} &= \frac{1}{9}\rho + \frac{1}{3}\rho u_x + \frac{1}{3}\rho u_x^2 - \frac{1}{6}\rho u_y^2 - \frac{1}{2}\rho u_x u_y^2 - \frac{1}{2}\rho u_x^2 u_y^2 \\
f_7^{eq} &= \frac{1}{36}\rho + \frac{1}{12}\rho u_x + \frac{1}{12}\rho u_y + \frac{1}{12}\rho u_x^2 + \frac{1}{12}\rho u_y^2 + \frac{1}{4}\rho u_x u_y + \frac{1}{4}\rho u_x^2 u_y + \frac{1}{4}\rho u_x u_y^2 + \frac{1}{4}\rho u_x^2 u_y^2 \\
f_8^{eq} &= \frac{1}{9}\rho + \frac{1}{3}\rho u_y - \frac{1}{6}\rho u_x^2 + \frac{1}{3}\rho u_y^2 - \frac{1}{2}\rho u_x^2 u_y - \frac{1}{2}\rho u_x^2 u_y^2
\end{aligned}$$

The relaxation frequencies can be expressed by the following block diagonal matrix:

$$\mathbf{\Lambda} = \text{diag}\left(1, 1, 1, \left[\begin{array}{cc} s_+ & s_- \\ s_- & s_+ \end{array}\right]\tau_f^{-1}, \omega_3, \omega_3, \omega_4\right)$$

where $s_+ = \frac{1}{2}\left(\tau_{f,bulk}^{-1} + \tau_f^{-1}\right)$ and $s_- = \frac{1}{2}\left(\tau_{f,bulk}^{-1} - \tau_f^{-1}\right)$. The bulk viscosity can be related to the associated relaxation time by:

$$\nu' = \frac{1}{3}\left(\tau_{f,bulk} - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t}.$$

Guo forcing can be applied using the following central moment transformations of the associated source terms:

$$\vec{S}^m = \left(\begin{array}{c} 0 \\ F_x \\ F_y \\ 0 \\ 0 \\ 0 \\ \left(\frac{1}{3} - v_x^2\right) F_y - 2 v_x v_y F_x \\ \left(\frac{1}{3} - v_y^2\right) F_x - 2 v_x v_y F_y \\ 4 v_x v_y \left(v_y F_x + v_x F_y\right) \end{array}\right),$$

and He forcing can be applied using these central moment terms:

$$\vec{S}^m = \left(\begin{array}{c} 0 \\ F_x \\ F_y \\ 0 \\ 0 \\ 0 \\ \frac{1}{3} F_y \\ \frac{1}{3} F_x \\ 0 \end{array}\right).$$

8.2 D3Q15

Table 8.4: Lattice vectors for D3Q15

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

Table 8.5: Weight factors for D3Q15

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

Table 8.6: Swift free-energy weight factors for D3Q15

i	w_i	w_i^{00}	γ_i	δ_i	w_i^p	w_i^t	w_i^{xx}	w_i^{yy}	w_i^{zz}	w_i^{xy}	w_i^{xz}	w_i^{yz}.
0	$\frac{2}{3}$	1	0	$-\frac{13}{2}$	$-\frac{7}{3}$	$-\frac{7}{3}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	0	0	0
1,8	$\frac{1}{3}$	0	0	1	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	$-\frac{1}{6}$	$-\frac{1}{6}$	0	0	0
2,9	$\frac{1}{3}$	0	0	1	$\frac{1}{3}$	$\frac{1}{3}$	$-\frac{1}{6}$	$\frac{1}{3}$	$-\frac{1}{6}$	0	0	0
3,10	$\frac{1}{3}$	0	0	1	$\frac{1}{3}$	$\frac{1}{3}$	$-\frac{1}{6}$	$-\frac{1}{6}$	$\frac{1}{3}$	0	0	0
4,11	$\frac{1}{24}$	0	0	-2	$\frac{1}{24}$	$\frac{1}{24}$	$-\frac{1}{48}$	$-\frac{1}{48}$	$-\frac{1}{48}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{8}$
5,12	$\frac{1}{24}$	0	0	-2	$\frac{1}{24}$	$\frac{1}{24}$	$-\frac{1}{48}$	$-\frac{1}{48}$	$-\frac{1}{48}$	$\frac{1}{8}$	$-\frac{1}{8}$	$-\frac{1}{8}$
6,13	$\frac{1}{24}$	0	0	-2	$\frac{1}{24}$	$\frac{1}{24}$	$-\frac{1}{48}$	$-\frac{1}{48}$	$-\frac{1}{48}$	$-\frac{1}{8}$	$\frac{1}{8}$	$-\frac{1}{8}$
7,14	$\frac{1}{24}$	0	0	-2	$\frac{1}{24}$	$\frac{1}{24}$	$-\frac{1}{48}$	$-\frac{1}{48}$	$-\frac{1}{48}$	$-\frac{1}{8}$	$-\frac{1}{8}$	$\frac{1}{8}$

8.2.1 Multiple relaxation time scheme

Definition of transformation matrix based on [159]:

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

For the standard local equilibrium distribution functions, the equilibrium moments are expressed for incompressible fluids as:

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

where ρ is used in place of ρ_0 for mildly compressible fluids, $j_x = \rho_0 u_x$, $j_y = \rho_0 u_y$, $j_z = \rho_0 u_z$ and, by default, $w_\epsilon = 1$ and $w_{\epsilon j} = -5$. If using Swift free-energy interactions, the equilibrium moments are expressed as:

$$
\vec{M}^{eq} = \begin{pmatrix}
\rho \\
-2\rho + \frac{3}{\rho}\left(j_x^2 + j_y^2 + j_z^2\right) + 3(P_0 - \kappa(\rho \nabla^2 \rho + \phi \nabla^2 \phi)) - \frac{1}{2}\kappa\left(|\nabla\rho|^2 + |\nabla\phi|^2\right) + 5\frac{\lambda}{\rho}(\vec{p} \cdot \nabla\rho) \\
16\rho - \frac{5}{\rho}(j_x^2 + j_y^2 + j_z^2) - 45(P_0 - \kappa(\rho \nabla^2 \rho + \phi \nabla^2 \phi)) + \frac{5}{2}\kappa\left(|\nabla\rho|^2 + |\nabla\phi|^2\right) - 85\frac{\lambda}{\rho}(\vec{p} \cdot \nabla\rho) \\
j_x \\
-\frac{7}{3} j_x \\
j_y \\
-\frac{7}{3} j_y \\
j_z \\
-\frac{7}{3} j_z \\
\frac{2j_x^2 - j_y^2 - j_z^2}{\rho} + \kappa(2(\partial_x\rho)^2 - (\partial_y\rho)^2 - (\partial_z\rho)^2 + 2(\partial_x\phi)^2 - (\partial_y\phi)^2 - (\partial_z\phi)^2) + \frac{2\lambda}{\rho}(2p_x\partial_x\rho - p_y\partial_y\rho - p_z\partial_z\rho) \\
\frac{j_y^2 - j_z^2}{\rho} + \kappa((\partial_y\rho)^2 - (\partial_z\rho)^2 + (\partial_y\phi)^2 - (\partial_z\phi)^2) + \frac{2\lambda}{\rho}(p_y\partial_y\rho - p_z\partial_z\rho) \\
\frac{j_x j_y}{\rho} + \kappa(\partial_x\rho\partial_y\rho + \partial_x\phi\partial_y\phi) + \frac{\lambda}{\rho}(p_x\partial_y\rho + p_y\partial_x\rho) \\
\frac{j_y j_z}{\rho} + \kappa(\partial_y\rho\partial_z\rho + \partial_y\phi\partial_z\phi) + \frac{\lambda}{\rho}(p_y\partial_z\rho + p_z\partial_y\rho) \\
\frac{j_x j_z}{\rho} + \kappa(\partial_x\rho\partial_z\rho + \partial_x\phi\partial_z\phi) + \frac{\lambda}{\rho}(p_x\partial_z\rho + p_z\partial_x\rho) \\
0
\end{pmatrix}
$$

The relaxation frequencies for the above moments can be expressed by the following diagonal matrix:

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

where the bulk viscosity can be related to the associated relaxation time by:

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

Recommended default values for the three variable relaxation frequencies s_2, s_4 and s_{14} are 1.2, 1.6 and 1.2 respectively.

Guo forcing can be applied using the following moment transformations of the associated source terms:

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

and He forcing can be applied using these moment terms:

$$\vec{S}^m = \begin{pmatrix} 0 \\ 2(v_x F_x + v_y F_y + v_z F_z) \\ -10(v_x F_x + v_y F_y + v_z F_z) \\ F_x \\ \left(-\frac{7}{3} + 5v_y^2 + 5v_z^2\right) F_x + 10 v_x v_y F_y + 10 v_x v_z F_z \\ F_y \\ \left(-\frac{7}{3} + 5v_x^2 + 5v_z^2\right) F_y + 10 v_x v_y F_x + 10 v_y v_z F_z \\ F_z \\ \left(-\frac{7}{3} + 5v_x^2 + 5v_y^2\right) F_z + 10 v_x v_z F_x + 10 v_y v_z F_y \\ 2(2v_x F_x - v_y F_y - v_z F_z) \\ 2(v_y F_y - v_z F_z) \\ v_x F_y + v_y F_x \\ v_y F_z + v_z F_y \\ v_z F_x + v_x F_z \\ 3v_y v_z F_x + 3v_x v_z F_y + 3v_x v_y F_z \end{pmatrix}.$$

8.3 D3Q19

Table 8.7: Lattice vectors for D3Q19

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

Table 8.8: Weight factors for D3Q19

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

Table 8.9: Swift free-energy weight factors for D3Q19

i	w_i	w_i^{00}	γ_i	δ_i	w_i^p	w_i^t	w_i^{xx}	w_i^{yy}	w_i^{zz}	w_i^{xy}	w_i^{xz}	w_i^{yz}.
0	1	1	0	$-\frac{9}{2}$	-2	-2	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0
1,10	$\frac{1}{6}$	0	$\frac{3}{2}$	$-\frac{3}{2}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{5}{12}$	$-\frac{1}{3}$	$-\frac{1}{3}$	0	0	0
2,11	$\frac{1}{6}$	0	$\frac{3}{2}$	$-\frac{3}{2}$	$\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$	$\frac{5}{12}$	$-\frac{1}{3}$	0	0	0
3,12	$\frac{1}{6}$	0	$\frac{3}{2}$	$-\frac{3}{2}$	$\frac{1}{6}$	$\frac{1}{6}$	$-\frac{1}{3}$	$-\frac{1}{3}$	$\frac{5}{12}$	0	0	0
4,13	$\frac{1}{12}$	0	$\frac{3}{2}$	$-\frac{3}{2}$	$\frac{1}{12}$	$\frac{1}{12}$	$-\frac{1}{24}$	$-\frac{1}{24}$	$\frac{1}{12}$	$\frac{1}{4}$	0	0
5,14	$\frac{1}{12}$	0	$\frac{3}{2}$	$-\frac{3}{2}$	$\frac{1}{12}$	$\frac{1}{12}$	$-\frac{1}{24}$	$-\frac{1}{24}$	$\frac{1}{12}$	$-\frac{1}{4}$	0	0
6,15	$\frac{1}{12}$	0	$\frac{3}{2}$	$-\frac{3}{2}$	$\frac{1}{12}$	$\frac{1}{12}$	$-\frac{1}{24}$	$\frac{1}{12}$	$-\frac{1}{24}$	0	$\frac{1}{4}$	0
7,16	$\frac{1}{12}$	0	$\frac{3}{2}$	$-\frac{3}{2}$	$\frac{1}{12}$	$\frac{1}{12}$	$-\frac{1}{24}$	$\frac{1}{12}$	$-\frac{1}{24}$	0	$-\frac{1}{4}$	0
8,17	$\frac{1}{12}$	0	$\frac{3}{2}$	$-\frac{3}{2}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$-\frac{1}{24}$	$-\frac{1}{24}$	0	0	$\frac{1}{4}$
9,18	$\frac{1}{12}$	0	$\frac{3}{2}$	$-\frac{3}{2}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$-\frac{1}{24}$	$-\frac{1}{24}$	0	0	$-\frac{1}{4}$

8.3.1 Multiple relaxation time scheme

Definition of transformation matrix based on [159]:

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

For the standard local equilibrium distribution functions, the equilibrium moments are expressed for incompressible fluids as:

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

where ρ is used in place of ρ_0 for mildly compressible fluids, $j_x = \rho_0 u_x$, $j_y = \rho_0 u_y$, $j_z = \rho_0 u_z$ and, by default, $w_\epsilon = 1$, $w_{\epsilon j} = -\frac{11}{2}$ and $w_{xx} = \frac{1}{2}$. If using Swift free-energy interactions, the equilibrium moments are expressed as:

$$\vec{M}^{eq} = \begin{pmatrix} \rho \\ -30\rho + \frac{19}{\rho}\left(j_x^2 + j_y^2 + j_z^2\right) + 57(P_0 - \kappa(\rho\nabla^2\rho + \phi\nabla^2\phi)) - \frac{19}{2}\kappa\left(|\nabla\rho|^2 + |\nabla\phi|^2\right) + 152\frac{\lambda}{\rho}(\vec{p}\cdot\nabla\rho) \\ 12\rho - \frac{11}{2\rho}(j_x^2 + j_y^2 + j_z^2) - 27(P_0 - \kappa(\rho\nabla^2\rho + \phi\nabla^2\phi)) + 8\kappa\left(|\nabla\rho|^2 + |\nabla\phi|^2\right) - \frac{109\lambda}{2\rho}(\vec{p}\cdot\nabla\rho) \\ j_x \\ -\frac{2}{3}j_x \\ j_y \\ -\frac{2}{3}j_y \\ j_z \\ -\frac{2}{3}j_z \\ \frac{2j_x^2 - j_y^2 - j_z^2}{\rho} + \kappa(2(\partial_x\rho)^2 - (\partial_y\rho)^2 - (\partial_z\rho)^2 + 2(\partial_x\phi)^2 - (\partial_y\phi)^2 - (\partial_z\phi)^2) + \frac{2\lambda}{\rho}(2p_x\partial_x\rho - p_y\partial_y\rho - p_z\partial_z\rho \\ -\frac{2j_x^2 - j_y^2 - j_z^2}{2\rho} - \frac{7}{2}\kappa(2(\partial_x\rho)^2 - (\partial_y\rho)^2 - (\partial_z\rho)^2 + 2(\partial_x\phi)^2 - (\partial_y\phi)^2 - (\partial_z\phi)^2) - \frac{\lambda}{\rho}(2p_x\partial_x\rho - p_y\partial_y\rho - p_z\partial_z\rho) \\ \frac{j_y^2 - j_z^2}{\rho} + \kappa((\partial_y\rho)^2 - (\partial_z\rho)^2 + (\partial_y\phi)^2 - (\partial_z\phi)^2) + \frac{2\lambda}{\rho}(p_y\partial_y\rho - p_z\partial_z\rho) \\ -\frac{j_y^2 - j_z^2}{2\rho} - \frac{7}{2}\kappa((\partial_y\rho)^2 - (\partial_z\rho)^2 + (\partial_y\phi)^2 - (\partial_z\phi)^2) - \frac{\lambda}{\rho}(p_y\partial_y\rho - p_z\partial_z\rho)\left(j_y^2 - j_z^2\right) \\ \frac{j_xj_y}{\rho} + \kappa(\partial_x\rho\partial_y\rho + \partial_x\phi\partial_y\phi) + \frac{\lambda}{\rho}(p_x\partial_y\rho + p_y\partial_x\rho) \\ \frac{j_yj_z}{\rho} + \kappa(\partial_y\rho\partial_z\rho + \partial_y\phi\partial_z\phi) + \frac{\lambda}{\rho}(p_y\partial_z\rho + p_z\partial_y\rho) \\ \frac{j_xj_z}{\rho} + \kappa(\partial_x\rho\partial_z\rho + \partial_x\phi\partial_z\phi) + \frac{\lambda}{\rho}(p_x\partial_z\rho + p_z\partial_x\rho) \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The relaxation frequencies for the above moments can be expressed by the following diagonal matrix:

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

where the bulk viscosity can be related to the associated relaxation time by:

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

Recommended default values for the three variable relaxation frequencies s_2, s_4 and s_{16} are 1.4, 1.4 and 1.98 respectively.

Guo forcing can be applied using the following moment transformations of the associated source terms:

$$\vec{S}^m = \begin{pmatrix} 0 \\ 38(v_xF_x + v_yF_y + v_zF_z) \\ -11(v_xF_x + v_yF_y + v_zF_z) \\ F_x \\ -\frac{2}{3}F_x \\ F_y \\ -\frac{2}{3}F_y \\ F_z \\ -\frac{2}{3}F_z \\ 2(2v_xF_x - v_yF_y - v_zF_z) \\ -(2v_xF_x - v_yF_y - v_zF_z) \\ 2(v_yF_y - v_zF_z) \\ -(v_yF_y - v_zF_z) \\ v_xF_y + v_yF_x \\ v_yF_z + v_zF_y \\ v_zF_x + v_xF_z \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

and He forcing can be applied using these moment terms:

$$\vec{S}^m = \begin{pmatrix} 0 \\ 38(v_xF_x + v_yF_y + v_zF_z) \\ -11(v_xF_x + v_yF_y + v_zF_z) \\ F_x \\ \left(-\frac{2}{3} + \frac{5}{2}v_y^2 + \frac{5}{2}v_z^2\right) F_x + 5v_xv_yF_y + 5v_xv_zF_z \\ F_y \\ \left(-\frac{2}{3} + \frac{5}{2}v_x^2 + \frac{5}{2}v_z^2\right) F_y + 5v_xv_yF_x + 5v_yv_zF_z \\ F_z \\ \left(-\frac{2}{3} + \frac{5}{2}v_x^2 + \frac{5}{2}v_y^2\right) F_z + 5v_xv_zF_x + 5v_yv_zF_y \\ 2(2v_xF_x - v_yF_y - v_zF_z) \\ -(2v_xF_x - v_yF_y - v_zF_z) \\ 2(v_yF_y - v_zF_z) \\ -(v_yF_y - v_zF_z) \\ v_xF_y + v_yF_x \\ v_yF_z + v_zF_y \\ v_zF_x + v_xF_z \\ \frac{3}{2}\left(v_y^2 - v_z^2\right) F_x + 3v_xv_yF_y - 3v_xv_zF_z \\ \frac{3}{2}\left(v_z^2 - v_x^2\right) F_y + 3v_yv_zF_z - 3v_xv_yF_x \\ \frac{3}{2}\left(v_x^2 - v_y^2\right) F_z + 3v_xv_zF_x - 3v_yv_zF_y \end{pmatrix}.$$

8.3.2 Cascaded LBE scheme

Definitions of transformation and shift matrices based on [35]:

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

$$
\mathbf{N} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_x & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_y & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_z & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_x u_y & -u_y & -u_x & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_x u_z & -u_z & 0 & -u_x & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_y u_z & 0 & -u_z & -u_y & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_x^2 & -2u_x & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_y^2 & 0 & -2u_y & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_z^2 & 0 & 0 & -2u_z & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_x u_y^2 & u_y^2 & 2u_x u_y & 0 & -2u_y & 0 & 0 & 0 & -u_x & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_x u_z^2 & u_z^2 & 0 & 2u_x u_z & 0 & -2u_z & 0 & 0 & 0 & -u_x & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_x^2 u_y & 2u_x u_y & u_x^2 & 0 & -2u_x & 0 & 0 & -u_y & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_x^2 u_z & 2u_x u_z & 0 & u_x^2 & 0 & -2u_x & 0 & -u_z & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
-u_y u_z^2 & 0 & u_z^2 & 2u_y u_z & 0 & 0 & -2u_z & 0 & 0 & -u_y & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
-u_y^2 u_z & 0 & 2u_y u_z & u_y^2 & 0 & 0 & -2u_y & 0 & -u_z & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
u_x^2 u_y^2 & -2u_x u_y^2 & -2u_x^2 u_y & 0 & 4u_x u_y & 0 & 0 & u_y^2 & u_x^2 & 0 & -2u_x & 0 & -2u_y & 0 & 0 & 0 & 1 & 0 & 0 \\
u_x^2 u_z^2 & -2u_x u_z^2 & 0 & -2u_x^2 u_z & 0 & 4u_x u_z & 0 & u_z^2 & 0 & u_x^2 & 0 & -2u_x & 0 & -2u_z & 0 & 0 & 0 & 1 & 0 \\
u_y^2 u_z^2 & 0 & -2u_y u_z^2 & -2u_y^2 u_z & 0 & 0 & 4u_y u_z & 0 & u_z^2 & u_y^2 & 0 & 0 & 0 & 0 & -2u_y & -2u_z & 0 & 0 & 1
\end{bmatrix}.
$$

The equilibrium central moments are expressed as follows:

$$\vec{\tilde{M}}^{eq} = \begin{pmatrix} \tilde{M}^{eq}_{000} \\ \tilde{M}^{eq}_{100} \\ \tilde{M}^{eq}_{010} \\ \tilde{M}^{eq}_{001} \\ \tilde{M}^{eq}_{110} \\ \tilde{M}^{eq}_{101} \\ \tilde{M}^{eq}_{011} \\ \tilde{M}^{eq}_{200} \\ \tilde{M}^{eq}_{020} \\ \tilde{M}^{eq}_{002} \\ \tilde{M}^{eq}_{120} \\ \tilde{M}^{eq}_{102} \\ \tilde{M}^{eq}_{210} \\ \tilde{M}^{eq}_{201} \\ \tilde{M}^{eq}_{012} \\ \tilde{M}^{eq}_{021} \\ \tilde{M}^{eq}_{220} \\ \tilde{M}^{eq}_{202} \\ \tilde{M}^{eq}_{022} \end{pmatrix} = \begin{pmatrix} \rho \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{3}\rho \\ \frac{1}{3}\rho \\ \frac{1}{3}\rho \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{9}\rho \\ \frac{1}{9}\rho \\ \frac{1}{9}\rho \end{pmatrix}$$

and transformation of the above leads to the following expressions for the local equilibrium distribution functions:

$$
\begin{aligned}
f_0^{eq} &= \frac{1}{3}\rho - \frac{1}{3}\rho u_x^2 - \frac{1}{3}\rho u_y^2 - \frac{1}{3}\rho u_z^2 + \rho u_x^2 u_y^2 + \rho u_x^2 u_z^2 + \rho u_y^2 u_z^2 \\
f_1^{eq} &= \frac{1}{18}\rho - \frac{1}{6}\rho u_x + \frac{1}{6}\rho u_x^2 - \frac{1}{6}\rho u_y^2 - \frac{1}{6}\rho u_z^2 + \frac{1}{2}\rho u_x u_y^2 + \frac{1}{2}\rho u_x u_z^2 - \frac{1}{2}\rho u_x^2 u_y^2 - \frac{1}{2}\rho u_x^2 u_z^2 \\
f_2^{eq} &= \frac{1}{18}\rho - \frac{1}{6}\rho u_y - \frac{1}{6}\rho u_x^2 + \frac{1}{6}\rho u_y^2 - \frac{1}{6}\rho u_z^2 + \frac{1}{2}\rho u_x^2 u_y + \frac{1}{2}\rho u_y u_z^2 - \frac{1}{2}\rho u_x^2 u_y^2 - \frac{1}{2}\rho u_y^2 u_z^2 \\
f_3^{eq} &= \frac{1}{18}\rho - \frac{1}{6}\rho u_z - \frac{1}{6}\rho u_x^2 - \frac{1}{6}\rho u_y^2 + \frac{1}{6}\rho u_z^2 + \frac{1}{2}\rho u_x^2 u_z + \frac{1}{2}\rho u_y^2 u_z - \frac{1}{2}\rho u_x^2 u_z^2 - \frac{1}{2}\rho u_y^2 u_z^2 \\
f_4^{eq} &= \frac{1}{36}\rho - \frac{1}{12}\rho u_x - \frac{1}{12}\rho u_y + \frac{1}{12}\rho u_x^2 + \frac{1}{12}\rho u_y^2 + \frac{1}{4}\rho u_x u_y - \frac{1}{4}\rho u_x^2 u_y - \frac{1}{4}\rho u_x u_y^2 + \frac{1}{4}\rho u_x^2 u_y^2 \\
f_5^{eq} &= \frac{1}{36}\rho - \frac{1}{12}\rho u_x + \frac{1}{12}\rho u_y + \frac{1}{12}\rho u_x^2 + \frac{1}{12}\rho u_y^2 - \frac{1}{4}\rho u_x u_y + \frac{1}{4}\rho u_x^2 u_y - \frac{1}{4}\rho u_x u_y^2 + \frac{1}{4}\rho u_x^2 u_y^2 \\
f_6^{eq} &= \frac{1}{36}\rho - \frac{1}{12}\rho u_x - \frac{1}{12}\rho u_z + \frac{1}{12}\rho u_x^2 + \frac{1}{12}\rho u_z^2 + \frac{1}{4}\rho u_x u_z - \frac{1}{4}\rho u_x^2 u_z - \frac{1}{4}\rho u_x u_z^2 + \frac{1}{4}\rho u_x^2 u_z^2 \\
f_7^{eq} &= \frac{1}{36}\rho - \frac{1}{12}\rho u_x + \frac{1}{12}\rho u_z + \frac{1}{12}\rho u_x^2 + \frac{1}{12}\rho u_z^2 - \frac{1}{4}\rho u_x u_z + \frac{1}{4}\rho u_x^2 u_z - \frac{1}{4}\rho u_x u_z^2 + \frac{1}{4}\rho u_x^2 u_z^2 \\
f_8^{eq} &= \frac{1}{36}\rho - \frac{1}{12}\rho u_y - \frac{1}{12}\rho u_z + \frac{1}{12}\rho u_y^2 + \frac{1}{12}\rho u_z^2 + \frac{1}{4}\rho u_y u_z - \frac{1}{4}\rho u_y^2 u_z - \frac{1}{4}\rho u_y u_z^2 + \frac{1}{4}\rho u_y^2 u_z^2 \\
f_9^{eq} &= \frac{1}{36}\rho - \frac{1}{12}\rho u_y + \frac{1}{12}\rho u_z + \frac{1}{12}\rho u_y^2 + \frac{1}{12}\rho u_z^2 - \frac{1}{4}\rho u_y u_z + \frac{1}{4}\rho u_y^2 u_z - \frac{1}{4}\rho u_y u_z^2 + \frac{1}{4}\rho u_y^2 u_z^2 \\
f_{10}^{eq} &= \frac{1}{18}\rho + \frac{1}{6}\rho u_x + \frac{1}{6}\rho u_x^2 - \frac{1}{6}\rho u_y^2 - \frac{1}{6}\rho u_z^2 - \frac{1}{2}\rho u_x u_y^2 - \frac{1}{2}\rho u_x u_z^2 - \frac{1}{2}\rho u_x^2 u_y^2 - \frac{1}{2}\rho u_x^2 u_z^2 \\
f_{11}^{eq} &= \frac{1}{18}\rho + \frac{1}{6}\rho u_y - \frac{1}{6}\rho u_x^2 + \frac{1}{6}\rho u_y^2 - \frac{1}{6}\rho u_z^2 - \frac{1}{2}\rho u_x^2 u_y - \frac{1}{2}\rho u_y u_z^2 - \frac{1}{2}\rho u_x^2 u_y^2 - \frac{1}{2}\rho u_y^2 u_z^2 \\
f_{12}^{eq} &= \frac{1}{18}\rho + \frac{1}{6}\rho u_z - \frac{1}{6}\rho u_x^2 - \frac{1}{6}\rho u_y^2 + \frac{1}{6}\rho u_z^2 - \frac{1}{2}\rho u_x^2 u_z - \frac{1}{2}\rho u_y^2 u_z - \frac{1}{2}\rho u_x^2 u_z^2 - \frac{1}{2}\rho u_y^2 u_z^2 \\
f_{13}^{eq} &= \frac{1}{36}\rho + \frac{1}{12}\rho u_x + \frac{1}{12}\rho u_y + \frac{1}{12}\rho u_x^2 + \frac{1}{12}\rho u_y^2 + \frac{1}{4}\rho u_x u_y + \frac{1}{4}\rho u_x^2 u_y + \frac{1}{4}\rho u_x u_y^2 + \frac{1}{4}\rho u_x^2 u_y^2 \\
f_{14}^{eq} &= \frac{1}{36}\rho + \frac{1}{12}\rho u_x - \frac{1}{12}\rho u_y + \frac{1}{12}\rho u_x^2 + \frac{1}{12}\rho u_y^2 - \frac{1}{4}\rho u_x u_y - \frac{1}{4}\rho u_x^2 u_y + \frac{1}{4}\rho u_x u_y^2 + \frac{1}{4}\rho u_x^2 u_y^2 \\
f_{15}^{eq} &= \frac{1}{36}\rho + \frac{1}{12}\rho u_x + \frac{1}{12}\rho u_z + \frac{1}{12}\rho u_x^2 + \frac{1}{12}\rho u_z^2 + \frac{1}{4}\rho u_x u_z + \frac{1}{4}\rho u_x^2 u_z + \frac{1}{4}\rho u_x u_z^2 + \frac{1}{4}\rho u_x^2 u_z^2 \\
f_{16}^{eq} &= \frac{1}{36}\rho + \frac{1}{12}\rho u_x - \frac{1}{12}\rho u_z + \frac{1}{12}\rho u_x^2 + \frac{1}{12}\rho u_z^2 - \frac{1}{4}\rho u_x u_z - \frac{1}{4}\rho u_x^2 u_z + \frac{1}{4}\rho u_x u_z^2 + \frac{1}{4}\rho u_x^2 u_z^2 \\
f_{17}^{eq} &= \frac{1}{36}\rho + \frac{1}{12}\rho u_y + \frac{1}{12}\rho u_z + \frac{1}{12}\rho u_y^2 + \frac{1}{12}\rho u_z^2 + \frac{1}{4}\rho u_y u_z + \frac{1}{4}\rho u_y^2 u_z + \frac{1}{4}\rho u_y u_z^2 + \frac{1}{4}\rho u_y^2 u_z^2 \\
f_{18}^{eq} &= \frac{1}{36}\rho + \frac{1}{12}\rho u_y - \frac{1}{12}\rho u_z + \frac{1}{12}\rho u_y^2 + \frac{1}{12}\rho u_z^2 - \frac{1}{4}\rho u_y u_z - \frac{1}{4}\rho u_y^2 u_z + \frac{1}{4}\rho u_y u_z^2 + \frac{1}{4}\rho u_y^2 u_z^2
\end{aligned}
$$

The relaxation frequencies can be expressed by the following block diagonal matrix:

$$
\mathbf{\Lambda} = \text{diag}\left(1, 1, 1, 1, \tau_f^{-1}, \tau_f^{-1}, \tau_f^{-1}, \begin{bmatrix} s_+ & s_- & s_- \\ s_- & s_+ & s_- \\ s_- & s_- & s_+ \end{bmatrix}, \omega_3, \omega_3, \omega_3, \omega_3, \omega_3, \omega_3, \omega_4, \omega_4, \omega_4\right)
$$

where $s_+ = \frac{1}{3}\left(\tau_{f,bulk}^{-1} + 2\tau_f^{-1}\right)$ and $s_- = \frac{1}{3}\left(\tau_{f,bulk}^{-1} - \tau_f^{-1}\right)$. The bulk viscosity can be related to the associated relaxation time by:

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

Guo forcing can be applied using the following central moment transformations of the associated source terms:

$$\vec{S}^m = \begin{pmatrix} 0 \\ F_x \\ F_y \\ F_z \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \left(\frac{1}{3} - v_y^2\right) F_x - 2v_x v_y F_y \\ \left(\frac{1}{3} - v_z^2\right) F_x - 2v_x v_z F_z \\ \left(\frac{1}{3} - v_x^2\right) F_y - 2v_x v_y F_x \\ \left(\frac{1}{3} - v_x^2\right) F_z - 2v_x v_z F_x \\ \left(\frac{1}{3} - v_z^2\right) F_y - 2v_y v_z F_z \\ \left(\frac{1}{3} - v_y^2\right) F_z - 2v_y v_z F_y \\ 4v_x v_y \left(v_y F_x + v_x F_y\right) - \frac{1}{3} v_z F_z \\ 4v_x v_z \left(v_z F_x + v_x F_z\right) - \frac{1}{3} v_y F_y \\ 4v_y v_z \left(v_z F_y + v_y F_z\right) - \frac{1}{3} v_x F_x \end{pmatrix},$$

and He forcing can be applied using these central moment terms:

$$\vec{S}^m = \begin{pmatrix} 0 \\ F_x \\ F_y \\ F_z \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{3} F_x \\ \frac{1}{3} F_x \\ \frac{1}{3} F_y \\ \frac{1}{3} F_z \\ \frac{1}{3} F_y \\ \frac{1}{3} F_z \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

8.4 D3Q27

Table 8.10: Lattice vectors for D3Q27

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 8.11: Weight factors for D3Q27

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	$\frac{1}{216}$

8.4.1 Multiple relaxation time scheme

Definition of transformation matrix based on [134]:

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

For the standard local equilibrium distribution functions, the equilibrium moments are expressed for incompressible fluids as:

$$\vec{M}^{eq} = \begin{pmatrix} \rho \\ j_x \\ j_y \\ j_z \\ e^{eq} \\ 3p_{xx}^{eq} \\ p_{ww}^{eq} \\ p_{xy}^{eq} \\ p_{yz}^{eq} \\ p_{zx}^{eq} \\ q_x^{eq} \\ q_y^{eq} \\ q_z^{eq} \\ \kappa_x^{eq} \\ \kappa_y^{eq} \\ \kappa_z^{eq} \\ \epsilon^{eq} \\ e_3^{eq} \\ 3\pi_{xx}^{eq} \\ \pi_{ww}^{eq} \\ \pi_{xy}^{eq} \\ \pi_{yz}^{eq} \\ \pi_{zx}^{eq} \\ \tau_x^{eq} \\ \tau_y^{eq} \\ \tau_z^{eq} \\ q_{xyz}^{eq} \end{pmatrix} = \begin{pmatrix} \rho \\ j_x \\ j_y \\ j_z \\ -\rho + \frac{1}{\rho_0}(j_x^2 + j_y^2 + j_z^2) \\ \frac{2j_x^2 - j_y^2 - j_z^2}{\rho_0} \\ \frac{j_y^2 - j_z^2}{\rho_0} \\ \frac{j_x j_y}{\rho_0} \\ \frac{j_y j_z}{\rho_0} \\ \frac{j_z j_x}{\rho_0} \\ -2j_x \\ -2j_y \\ -2j_z \\ j_x \\ j_y \\ j_z \\ \rho - \frac{2}{\rho_0}(j_x^2 + j_y^2 + j_z^2) \\ -\rho + \frac{1}{\rho_0}(j_x^2 + j_y^2 + j_z^2) \\ -\frac{2j_x^2 - j_y^2 - j_z^2}{\rho_0} \\ -\frac{j_y^2 - j_z^2}{\rho_0} \\ -\frac{j_x j_y}{\rho_0} \\ -\frac{j_y j_z}{\rho_0} \\ -\frac{j_z j_x}{\rho_0} \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

where ρ is used in place of ρ_0 for mildly compressible fluids, $j_x = \rho_0 u_x$, $j_y = \rho_0 u_y$ and $j_z = \rho_0 u_z$.

The relaxation frequencies for the above moments can be expressed by the following diagonal matrix:

$$\vec{s} = \text{diag}\left(1, 1, 1, 1, \tau_{f,bulk}^{-1}, \tau_f^{-1}, \tau_f^{-1}, \tau_f^{-1}, \tau_f^{-1}, \tau_f^{-1}, s_{10}, s_{10}, s_{10}, s_{13}, s_{13}, s_{13}, s_{16}, s_{17}, s_{18}, s_{18}, s_{20}, s_{20}, s_{20}, s_{23}, s_{23}, s_{23}, s_{26}\right)$$

where the bulk viscosity can be related to the associated relaxation time by:

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

Recommended default values for the eight variable relaxation frequencies are: $s_{10} = 1.5$, $s_{13} = 1.83$, $s_{16} = 1.4$, $s_{17} = 1.61$, $s_{18} = s_{20} = 1.98$ and $s_{23} = s_{26} = 1.74$.

Guo forcing can be applied using the following moment transformations of the associated source terms:

$$\vec{S}^m = \begin{pmatrix} 0 \\ F_x \\ F_y \\ F_z \\ 2(v_x F_x + v_y F_y + v_z F_z) \\ 2(2v_x F_x - v_y F_y - v_z F_z) \\ 2(v_y F_y - v_z F_z) \\ v_y F_x + v_x F_y \\ v_z F_y + v_y F_z \\ v_x F_z + v_z F_x \\ -2F_x \\ -2F_y \\ -2F_z \\ F_x \\ F_y \\ F_z \\ -4(v_x F_x + v_y F_y + v_z F_z) \\ 6(v_x F_x + v_y F_y + v_z F_z) \\ -2(2v_x F_x - v_y F_y - v_z F_z) \\ -2(v_y F_y - v_z F_z) \\ -(v_y F_x + v_x F_y) \\ -(v_z F_y + v_y F_z) \\ -(v_x F_z + v_z F_x) \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

and He forcing can be applied using these moment terms:

$$\vec{S}^m = \begin{pmatrix} 0 \\ F_x \\ F_y \\ F_z \\ 2(v_x F_x + v_y F_y + v_z F_z) \\ 2(2v_x F_x - v_y F_y - v_z F_z) \\ 2(v_y F_y - v_z F_z) \\ v_y F_x + v_x F_y \\ v_z F_y + v_y F_z \\ v_x F_z + v_z F_x \\ \left(-2 + 3v_y^2 + 3v_z^2\right) F_x + 6v_x v_y F_y + 6v_x v_z F_z \\ \left(-2 + 3v_x^2 + 3v_z^2\right) F_y + 6v_x v_y F_x + 6v_y v_z F_z \\ \left(-2 + 3v_x^2 + 3v_y^2\right) F_z + 6v_x v_y F_x + 6v_y v_z F_y \\ F_x \left(1 - 3v_y^2 - 3v_z^2\right) - 6v_x v_y F_y - 6v_x v_z F_z \\ F_y \left(1 - 3v_x^2 - 3v_z^2\right) - 6v_x v_y F_x - 6v_y v_z F_z \\ F_z \left(1 - 3v_x^2 - 3v_y^2\right) - 6v_x v_y F_x - 6v_y v_z F_y \\ -4(v_x F_x + v_y F_y + v_z F_z) \\ 6(v_x F_x + v_y F_y + v_z F_z) \\ -2(2v_x F_x - v_y F_y - v_z F_z) \\ -2(v_y F_y - v_z F_z) \\ -(v_y F_x + v_x F_y) \\ -(v_z F_y + v_y F_z) \\ -(v_x F_z + v_z F_x) \\ \left(v_y^2 - v_z^2\right) F_x + 2v_x v_y F_y - 2v_x v_z F_z \\ \left(v_z^2 - v_x^2\right) F_y + 2v_y v_z F_z - 2v_x v_y F_x \\ \left(v_x^2 - v_y^2\right) F_z + 2v_x v_z F_x - 2v_y v_z F_y \\ v_y v_z F_x + v_x v_z F_y + v_x v_y F_z \end{pmatrix}.$$

8.4.2 Cascaded LBE scheme

Definitions of transformation and shift matrices based on [33]:

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

$$\mathbf{N} = \left[\begin{array}{*{27}{c}}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_x & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_y & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_z & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_x u_y & -u_y & -u_x & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_x u_z & -u_z & 0 & -u_x & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_y u_z & 0 & -u_z & -u_y & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_x^2 & -2u_x & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_y^2 & 0 & -2u_y & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_z^2 & 0 & 0 & -2u_z & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_x u_y^2 & u_y^2 & 2u_x u_y & 0 & -2u_y & 0 & 0 & 0 & -u_x & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_x u_z^2 & u_z^2 & 0 & 2u_x u_z & 0 & -2u_z & 0 & 0 & 0 & -u_x & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_x^2 u_y & 2u_x u_y & u_x^2 & 0 & -2u_x & 0 & 0 & -u_y & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_x^2 u_z & 2u_x u_z & 0 & u_x^2 & 0 & -2u_x & 0 & -u_z & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_y u_z^2 & 0 & u_z^2 & 2u_y u_z & 0 & 0 & -2u_z & 0 & 0 & -u_y & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_y^2 u_z & 0 & 2u_y u_z & u_y^2 & 0 & 0 & -2u_y & 0 & -u_z & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-u_x u_y u_z & u_y u_z & u_x u_z & u_x u_y & -u_z & -u_y & -u_x & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_x^2 u_y^2 & -2u_x u_y^2 & -2u_x^2 u_y & 0 & 4u_x u_y & 0 & 0 & u_y^2 & u_x^2 & 0 & -2u_x & 0 & -2u_y & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_x^2 u_z^2 & -2u_x u_z^2 & 0 & -2u_x^2 u_z & 0 & 4u_x u_z & 0 & u_z^2 & 0 & u_x^2 & 0 & -2u_x & 0 & -2u_z & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_y^2 u_z^2 & 0 & -2u_y u_z^2 & -2u_y^2 u_z & 0 & 0 & 4u_y u_z & 0 & u_z^2 & u_y^2 & 0 & 0 & 0 & 0 & -2u_y & -2u_z & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_x^2 u_y u_z & -2u_x u_y u_z & -u_x^2 u_z & -u_x^2 u_y & 2u_x u_z & 2u_x u_y & u_x^2 & u_y u_z & 0 & 0 & 0 & 0 & -u_z & -u_y & 0 & 0 & -2u_x & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
u_x u_y^2 u_z & -u_y^2 u_z & -2u_x u_y u_z & -u_x u_y^2 & 2u_y u_z & u_y^2 & 2u_x u_y & 0 & u_x u_z & 0 & -u_z & 0 & 0 & 0 & 0 & -u_x & -2u_y & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
u_x u_y u_z^2 & -u_y u_z^2 & -u_x u_z^2 & -2u_x u_y u_z & u_z^2 & 2u_y u_z & 2u_x u_z & 0 & 0 & u_x u_y & 0 & -u_y & 0 & 0 & -u_x & 0 & -2u_z & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
-u_x u_y^2 u_z^2 & u_y^2 u_z^2 & 2u_x u_y u_z^2 & 2u_x u_y^2 u_z & -2u_y u_z^2 & -2u_y^2 u_z & -4u_x u_y u_z & 0 & -u_x u_z^2 & -u_x u_y^2 & u_z^2 & u_y^2 & 0 & 0 & 2u_x u_y & 2u_x u_z & 4u_y u_z & 0 & 0 & -u_x & 0 & -2u_z & -2u_y & 1 & 0 & 0 & 0 \\
-u_x^2 u_y u_z^2 & 2u_x u_y u_z^2 & u_x^2 u_z^2 & 2u_x^2 u_y u_z & -2u_x u_z^2 & -4u_x u_y u_z & -2u_x^2 u_z & -u_y u_z^2 & 0 & -u_x^2 u_y & 0 & 2u_x u_y & u_z^2 & 2u_y u_z & u_x^2 & 0 & 4u_x u_z & 0 & -u_y & 0 & -2u_z & 0 & -2u_x & 0 & 1 & 0 & 0 \\
-u_x^2 u_y^2 u_z & 2u_x u_y^2 u_z & 2u_x^2 u_y u_z & u_x^2 u_y^2 & -4u_x u_y u_z & -2u_x u_y^2 & -2u_x^2 u_y & -u_y^2 u_z & -u_x^2 u_z & 0 & 2u_x u_z & 0 & 2u_y u_z & u_y^2 & 0 & u_x^2 & 4u_x u_y & -u_z & 0 & 0 & -2u_y & -2u_x & 0 & 0 & 0 & 1 & 0 \\
u_z^2 u_y^2 u_z^2 & -2u_x u_y^2 u_z^2 & -2u_x^2 u_y u_z^2 & -2u_x^2 u_y^2 u_z & 4u_x u_y u_z^2 & 4u_x u_y^2 u_z & 4u_x^2 u_y u_z & u_y^2 u_z^2 & u_x^2 u_z^2 & u_x^2 u_y^2 & -2u_x u_z^2 & -2u_x u_y^2 & -2u_y u_z^2 & -2u_y^2 u_z & -2u_x^2 u_y & -2u_x^2 u_z & -8u_x u_y u_z & u_z^2 & u_y^2 & u_x^2 & 4u_y u_z & 4u_x u_z & 4u_x u_y & -2u_x & -2u_y & -2u_z & 1
\end{array}\right].$$

The equilibrium central moments are expressed as follows:

$$\vec{\tilde{M}}^{eq} = \begin{pmatrix} \tilde{M}^{eq}_{000} \\ \tilde{M}^{eq}_{100} \\ \tilde{M}^{eq}_{010} \\ \tilde{M}^{eq}_{001} \\ \tilde{M}^{eq}_{110} \\ \tilde{M}^{eq}_{101} \\ \tilde{M}^{eq}_{011} \\ \tilde{M}^{eq}_{200} \\ \tilde{M}^{eq}_{020} \\ \tilde{M}^{eq}_{002} \\ \tilde{M}^{eq}_{120} \\ \tilde{M}^{eq}_{102} \\ \tilde{M}^{eq}_{210} \\ \tilde{M}^{eq}_{201} \\ \tilde{M}^{eq}_{012} \\ \tilde{M}^{eq}_{021} \\ \tilde{M}^{eq}_{111} \\ \tilde{M}^{eq}_{220} \\ \tilde{M}^{eq}_{202} \\ \tilde{M}^{eq}_{022} \\ \tilde{M}^{eq}_{211} \\ \tilde{M}^{eq}_{121} \\ \tilde{M}^{eq}_{112} \\ \tilde{M}^{eq}_{122} \\ \tilde{M}^{eq}_{212} \\ \tilde{M}^{eq}_{221} \\ \tilde{M}^{eq}_{222} \end{pmatrix} = \begin{pmatrix} \rho \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{3}\rho \\ \frac{1}{3}\rho \\ \frac{1}{3}\rho \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{9}\rho \\ \frac{1}{9}\rho \\ \frac{1}{9}\rho \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{27}\rho \end{pmatrix}$$

and transformation of the above leads to the following expressions for the local equilibrium distribution functions:

$$
\begin{aligned}
f_0^{eq} = & \frac{8}{27}\rho - \frac{4}{9}\rho u_x^2 - \frac{4}{9}\rho u_y^2 - \frac{4}{9}\rho u_z^2 + \frac{2}{3}\rho u_x^2 u_y^2 + \frac{2}{3}\rho u_x^2 u_z^2 + \frac{2}{3}\rho u_y^2 u_z^2 - \rho u_x^2 u_y^2 u_z^2 \\
f_1^{eq} = & \frac{2}{27}\rho - \frac{2}{9}\rho u_x + \frac{2}{9}\rho u_x^2 - \frac{1}{9}\rho u_y^2 - \frac{1}{9}\rho u_z^2 + \frac{1}{3}\rho u_x u_y^2 + \frac{1}{3}\rho u_x u_z^2 - \frac{1}{3}\rho u_x^2 u_y^2 - \frac{1}{3}\rho u_x^2 u_z^2 + \frac{1}{6}\rho u_y^2 u_z^2 \\
& - \frac{1}{2}\rho u_x u_y^2 u_z^2 + \frac{1}{2}\rho u_x^2 u_y^2 u_z^2 \\
f_2^{eq} = & \frac{2}{27}\rho - \frac{2}{9}\rho u_y - \frac{1}{9}\rho u_x^2 + \frac{2}{9}\rho u_y^2 - \frac{1}{9}\rho u_z^2 + \frac{1}{3}\rho u_x^2 u_y + \frac{1}{3}\rho u_y u_z^2 - \frac{1}{3}\rho u_x^2 u_y^2 + \frac{1}{6}\rho u_x^2 u_z^2 - \frac{1}{3}\rho u_y^2 u_z^2 \\
& - \frac{1}{2}\rho u_x^2 u_y u_z^2 + \frac{1}{2}\rho u_x^2 u_y^2 u_z^2 \\
f_3^{eq} = & \frac{2}{27}\rho - \frac{2}{9}\rho u_z - \frac{1}{9}\rho u_x^2 - \frac{1}{9}\rho u_y^2 + \frac{2}{9}\rho u_z^2 + \frac{1}{3}\rho u_x^2 u_z + \frac{1}{3}\rho u_y^2 u_z + \frac{1}{6}\rho u_x^2 u_y^2 - \frac{1}{3}\rho u_x^2 u_z^2 - \frac{1}{3}\rho u_y^2 u_z^2 \\
& - \frac{1}{2}\rho u_x^2 u_y^2 u_z + \frac{1}{2}\rho u_x^2 u_y^2 u_z^2 \\
f_4^{eq} = & \frac{1}{54}\rho - \frac{1}{18}\rho u_x - \frac{1}{18}\rho u_y + \frac{1}{18}\rho u_x^2 + \frac{1}{18}\rho u_y^2 - \frac{1}{36}\rho u_z^2 + \frac{1}{6}\rho u_x u_y - \frac{1}{6}\rho u_x u_y^2 - \frac{1}{6}\rho u_x^2 u_y + \frac{1}{12}\rho u_x u_z^2 + \frac{1}{12}\rho u_y u_z^2 \\
& + \frac{1}{6}\rho u_x^2 u_y^2 - \frac{1}{12}\rho u_x^2 u_z^2 - \frac{1}{12}\rho u_y^2 u_z^2 - \frac{1}{4}\rho u_x u_y u_z^2 + \frac{1}{4}\rho u_x u_y^2 u_z^2 + \frac{1}{4}\rho u_x^2 u_y u_z^2 - \frac{1}{4}\rho u_x^2 u_y^2 u_z^2 \\
f_5^{eq} = & \frac{1}{54}\rho - \frac{1}{18}\rho u_x + \frac{1}{18}\rho u_y + \frac{1}{18}\rho u_x^2 + \frac{1}{18}\rho u_y^2 - \frac{1}{36}\rho u_z^2 - \frac{1}{6}\rho u_x u_y - \frac{1}{6}\rho u_x u_y^2 + \frac{1}{6}\rho u_x^2 u_y + \frac{1}{12}\rho u_x u_z^2 - \frac{1}{12}\rho u_y u_z^2 \\
& + \frac{1}{6}\rho u_x^2 u_y^2 - \frac{1}{12}\rho u_x^2 u_z^2 - \frac{1}{12}\rho u_y^2 u_z^2 + \frac{1}{4}\rho u_x u_y u_z^2 + \frac{1}{4}\rho u_x u_y^2 u_z^2 - \frac{1}{4}\rho u_x^2 u_y u_z^2 - \frac{1}{4}\rho u_x^2 u_y^2 u_z^2 \\
f_6^{eq} = & \frac{1}{54}\rho - \frac{1}{18}\rho u_x - \frac{1}{18}\rho u_z + \frac{1}{18}\rho u_x^2 - \frac{1}{36}\rho u_y^2 + \frac{1}{18}\rho u_z^2 + \frac{1}{6}\rho u_x u_z - \frac{1}{6}\rho u_x u_z^2 - \frac{1}{6}\rho u_x^2 u_z + \frac{1}{12}\rho u_x u_y^2 + \frac{1}{12}\rho u_y^2 u_z \\
& - \frac{1}{12}\rho u_x^2 u_y^2 + \frac{1}{6}\rho u_x^2 u_z^2 - \frac{1}{12}\rho u_y^2 u_z^2 - \frac{1}{4}\rho u_x u_y^2 u_z + \frac{1}{4}\rho u_x^2 u_y^2 u_z + \frac{1}{4}\rho u_x u_y^2 u_z^2 - \frac{1}{4}\rho u_x^2 u_y^2 u_z^2 \\
f_7^{eq} = & \frac{1}{54}\rho - \frac{1}{18}\rho u_x + \frac{1}{18}\rho u_z + \frac{1}{18}\rho u_x^2 - \frac{1}{36}\rho u_y^2 + \frac{1}{18}\rho u_z^2 - \frac{1}{6}\rho u_x u_z - \frac{1}{6}\rho u_x u_z^2 + \frac{1}{6}\rho u_x^2 u_z + \frac{1}{12}\rho u_x u_y^2 - \frac{1}{12}\rho u_y^2 u_z \\
& - \frac{1}{12}\rho u_x^2 u_y^2 + \frac{1}{6}\rho u_x^2 u_z^2 - \frac{1}{12}\rho u_y^2 u_z^2 + \frac{1}{4}\rho u_x u_y^2 u_z - \frac{1}{4}\rho u_x^2 u_y^2 u_z + \frac{1}{4}\rho u_x u_y^2 u_z^2 - \frac{1}{4}\rho u_x^2 u_y^2 u_z^2 \\
f_8^{eq} = & \frac{1}{54}\rho - \frac{1}{18}\rho u_y - \frac{1}{18}\rho u_z - \frac{1}{36}\rho u_x^2 + \frac{1}{18}\rho u_y^2 + \frac{1}{18}\rho u_z^2 + \frac{1}{6}\rho u_y u_z - \frac{1}{6}\rho u_y^2 u_z - \frac{1}{6}\rho u_y u_z^2 + \frac{1}{12}\rho u_x^2 u_y + \frac{1}{12}\rho u_x^2 u_z \\
& - \frac{1}{12}\rho u_x^2 u_y^2 - \frac{1}{12}\rho u_x^2 u_z^2 + \frac{1}{6}\rho u_y^2 u_z^2 - \frac{1}{4}\rho u_x^2 u_y u_z + \frac{1}{4}\rho u_x^2 u_y^2 u_z + \frac{1}{4}\rho u_x^2 u_y u_z^2 - \frac{1}{4}\rho u_x^2 u_y^2 u_z^2 \\
f_9^{eq} = & \frac{1}{54}\rho - \frac{1}{18}\rho u_y + \frac{1}{18}\rho u_z - \frac{1}{36}\rho u_x^2 + \frac{1}{18}\rho u_y^2 + \frac{1}{18}\rho u_z^2 - \frac{1}{6}\rho u_y u_z + \frac{1}{6}\rho u_y^2 u_z - \frac{1}{6}\rho u_y u_z^2 + \frac{1}{12}\rho u_x^2 u_y - \frac{1}{12}\rho u_x^2 u_z \\
& - \frac{1}{12}\rho u_x^2 u_y^2 - \frac{1}{12}\rho u_x^2 u_z^2 + \frac{1}{6}\rho u_y^2 u_z^2 + \frac{1}{4}\rho u_x^2 u_y u_z - \frac{1}{4}\rho u_x^2 u_y^2 u_z + \frac{1}{4}\rho u_x^2 u_y u_z^2 - \frac{1}{4}\rho u_x^2 u_y^2 u_z^2 \\
f_{10}^{eq} = & \frac{1}{216}\rho - \frac{1}{72}\rho u_x - \frac{1}{72}\rho u_y - \frac{1}{72}\rho u_z + \frac{1}{72}\rho u_x^2 + \frac{1}{72}\rho u_y^2 + \frac{1}{72}\rho u_z^2 + \frac{1}{24}\rho u_x u_y + \frac{1}{24}\rho u_x u_z + \frac{1}{24}\rho u_y u_z \\
& - \frac{1}{24}\rho u_x^2 u_y - \frac{1}{24}\rho u_x^2 u_z - \frac{1}{24}\rho u_x u_y^2 - \frac{1}{24}\rho u_y^2 u_z - \frac{1}{24}\rho u_x u_z^2 - \frac{1}{24}\rho u_y u_z^2 - \frac{1}{8}\rho u_x u_y u_z \\
& + \frac{1}{24}\rho u_x^2 u_y^2 + \frac{1}{24}\rho u_x^2 u_z^2 + \frac{1}{24}\rho u_y^2 u_z^2 + \frac{1}{8}\rho u_x^2 u_y u_z + \frac{1}{8}\rho u_x u_y^2 u_z + \frac{1}{8}\rho u_x u_y u_z^2 \\
& - \frac{1}{8}\rho u_x^2 u_y^2 u_z - \frac{1}{8}\rho u_x^2 u_y u_z^2 - \frac{1}{8}\rho u_x u_y^2 u_z^2 + \frac{1}{8}\rho u_x^2 u_y^2 u_z^2 \\
f_{11}^{eq} = & \frac{1}{216}\rho - \frac{1}{72}\rho u_x - \frac{1}{72}\rho u_y + \frac{1}{72}\rho u_z + \frac{1}{72}\rho u_x^2 + \frac{1}{72}\rho u_y^2 + \frac{1}{72}\rho u_z^2 + \frac{1}{24}\rho u_x u_y - \frac{1}{24}\rho u_x u_z - \frac{1}{24}\rho u_y u_z \\
& - \frac{1}{24}\rho u_x^2 u_y + \frac{1}{24}\rho u_x^2 u_z - \frac{1}{24}\rho u_x u_y^2 + \frac{1}{24}\rho u_y^2 u_z - \frac{1}{24}\rho u_x u_z^2 - \frac{1}{24}\rho u_y u_z^2 + \frac{1}{8}\rho u_x u_y u_z \\
& + \frac{1}{24}\rho u_x^2 u_y^2 + \frac{1}{24}\rho u_x^2 u_z^2 + \frac{1}{24}\rho u_y^2 u_z^2 - \frac{1}{8}\rho u_x^2 u_y u_z - \frac{1}{8}\rho u_x u_y^2 u_z + \frac{1}{8}\rho u_x u_y u_z^2 \\
& + \frac{1}{8}\rho u_x^2 u_y^2 u_z - \frac{1}{8}\rho u_x^2 u_y u_z^2 - \frac{1}{8}\rho u_x u_y^2 u_z^2 + \frac{1}{8}\rho u_x^2 u_y^2 u_z^2
\end{aligned}
$$

$$\begin{aligned}
f_{12}^{eq} = & \frac{1}{216}\rho - \frac{1}{72}\rho u_x + \frac{1}{72}\rho u_y - \frac{1}{72}\rho u_z + \frac{1}{72}\rho u_x^2 + \frac{1}{72}\rho u_y^2 + \frac{1}{72}\rho u_z^2 - \frac{1}{24}\rho u_x u_y + \frac{1}{24}\rho u_x u_z - \frac{1}{24}\rho u_y u_z \\
& + \frac{1}{24}\rho u_x^2 u_y - \frac{1}{24}\rho u_x^2 u_z - \frac{1}{24}\rho u_x u_y^2 - \frac{1}{24}\rho u_y^2 u_z - \frac{1}{24}\rho u_x u_z^2 + \frac{1}{24}\rho u_y u_z^2 + \frac{1}{8}\rho u_x u_y u_z \\
& + \frac{1}{24}\rho u_x^2 u_y^2 + \frac{1}{24}\rho u_x^2 u_z^2 + \frac{1}{24}\rho u_y^2 u_z^2 - \frac{1}{8}\rho u_x^2 u_y u_z + \frac{1}{8}\rho u_x u_y^2 u_z - \frac{1}{8}\rho u_x u_y u_z^2 \\
& - \frac{1}{8}\rho u_x^2 u_y^2 u_z + \frac{1}{8}\rho u_x^2 u_y u_z^2 - \frac{1}{8}\rho u_x u_y^2 u_z^2 + \frac{1}{8}\rho u_x^2 u_y^2 u_z^2 \\
f_{13}^{eq} = & \frac{1}{216}\rho - \frac{1}{72}\rho u_x + \frac{1}{72}\rho u_y + \frac{1}{72}\rho u_z + \frac{1}{72}\rho u_x^2 + \frac{1}{72}\rho u_y^2 + \frac{1}{72}\rho u_z^2 - \frac{1}{24}\rho u_x u_y - \frac{1}{24}\rho u_x u_z + \frac{1}{24}\rho u_y u_z \\
& + \frac{1}{24}\rho u_x^2 u_y + \frac{1}{24}\rho u_x^2 u_z - \frac{1}{24}\rho u_x u_y^2 + \frac{1}{24}\rho u_y^2 u_z - \frac{1}{24}\rho u_x u_z^2 + \frac{1}{24}\rho u_y u_z^2 - \frac{1}{8}\rho u_x u_y u_z \\
& + \frac{1}{24}\rho u_x^2 u_y^2 + \frac{1}{24}\rho u_x^2 u_z^2 + \frac{1}{24}\rho u_y^2 u_z^2 + \frac{1}{8}\rho u_x^2 u_y u_z - \frac{1}{8}\rho u_x u_y^2 u_z - \frac{1}{8}\rho u_x u_y u_z^2 \\
& + \frac{1}{8}\rho u_x^2 u_y^2 u_z + \frac{1}{8}\rho u_x^2 u_y u_z^2 - \frac{1}{8}\rho u_x u_y^2 u_z^2 + \frac{1}{8}\rho u_x^2 u_y^2 u_z^2 \\
f_{14}^{eq} = & \frac{2}{27}\rho + \frac{2}{9}\rho u_x + \frac{2}{9}\rho u_x^2 - \frac{1}{9}\rho u_y^2 - \frac{1}{9}\rho u_z^2 - \frac{1}{3}\rho u_x u_y^2 - \frac{1}{3}\rho u_x u_z^2 - \frac{1}{3}\rho u_x^2 u_y^2 - \frac{1}{3}\rho u_x^2 u_z^2 + \frac{1}{6}\rho u_y^2 u_z^2 \\
& + \frac{1}{2}\rho u_x u_y^2 u_z^2 + \frac{1}{2}\rho u_x^2 u_y^2 u_z^2 \\
f_{15}^{eq} = & \frac{2}{27}\rho + \frac{2}{9}\rho u_y - \frac{1}{9}\rho u_x^2 + \frac{2}{9}\rho u_y^2 - \frac{1}{9}\rho u_z^2 - \frac{1}{3}\rho u_x^2 u_y - \frac{1}{3}\rho u_y u_z^2 - \frac{1}{3}\rho u_x^2 u_y^2 + \frac{1}{6}\rho u_x^2 u_z^2 - \frac{1}{3}\rho u_y^2 u_z^2 \\
& + \frac{1}{2}\rho u_x^2 u_y u_z^2 + \frac{1}{2}\rho u_x^2 u_y^2 u_z^2 \\
f_{16}^{eq} = & \frac{2}{27}\rho + \frac{2}{9}\rho u_z - \frac{1}{9}\rho u_x^2 - \frac{1}{9}\rho u_y^2 + \frac{2}{9}\rho u_z^2 - \frac{1}{3}\rho u_x^2 u_z - \frac{1}{3}\rho u_y^2 u_z + \frac{1}{6}\rho u_x^2 u_y^2 - \frac{1}{3}\rho u_x^2 u_z^2 - \frac{1}{3}\rho u_y^2 u_z^2 \\
& + \frac{1}{2}\rho u_x^2 u_y^2 u_z + \frac{1}{2}\rho u_x^2 u_y^2 u_z^2 \\
f_{17}^{eq} = & \frac{1}{54}\rho + \frac{1}{18}\rho u_x + \frac{1}{18}\rho u_y + \frac{1}{18}\rho u_x^2 + \frac{1}{18}\rho u_y^2 - \frac{1}{36}\rho u_z^2 + \frac{1}{6}\rho u_x u_y + \frac{1}{6}\rho u_x u_y^2 + \frac{1}{6}\rho u_x^2 u_y - \frac{1}{12}\rho u_x u_z^2 - \frac{1}{12}\rho u_y u_z^2 \\
& + \frac{1}{6}\rho u_x^2 u_y^2 - \frac{1}{12}\rho u_x^2 u_z^2 - \frac{1}{12}\rho u_y^2 u_z^2 - \frac{1}{4}\rho u_x u_y u_z^2 - \frac{1}{4}\rho u_x u_y^2 u_z^2 - \frac{1}{4}\rho u_x^2 u_y u_z^2 - \frac{1}{4}\rho u_x^2 u_y^2 u_z^2 \\
f_{18}^{eq} = & \frac{1}{54}\rho + \frac{1}{18}\rho u_x - \frac{1}{18}\rho u_y + \frac{1}{18}\rho u_x^2 + \frac{1}{18}\rho u_y^2 - \frac{1}{36}\rho u_z^2 - \frac{1}{6}\rho u_x u_y + \frac{1}{6}\rho u_x u_y^2 - \frac{1}{6}\rho u_x^2 u_y - \frac{1}{12}\rho u_x u_z^2 + \frac{1}{12}\rho u_y u_z^2 \\
& + \frac{1}{6}\rho u_x^2 u_y^2 - \frac{1}{12}\rho u_x^2 u_z^2 - \frac{1}{12}\rho u_y^2 u_z^2 + \frac{1}{4}\rho u_x u_y u_z^2 - \frac{1}{4}\rho u_x u_y^2 u_z^2 + \frac{1}{4}\rho u_x^2 u_y u_z^2 - \frac{1}{4}\rho u_x^2 u_y^2 u_z^2 \\
f_{19}^{eq} = & \frac{1}{54}\rho + \frac{1}{18}\rho u_x + \frac{1}{18}\rho u_z + \frac{1}{18}\rho u_x^2 - \frac{1}{36}\rho u_y^2 + \frac{1}{18}\rho u_z^2 + \frac{1}{6}\rho u_x u_z + \frac{1}{6}\rho u_x u_z^2 + \frac{1}{6}\rho u_x^2 u_z - \frac{1}{12}\rho u_x u_y^2 - \frac{1}{12}\rho u_y^2 u_z \\
& - \frac{1}{12}\rho u_x^2 u_y^2 + \frac{1}{6}\rho u_x^2 u_z^2 - \frac{1}{12}\rho u_y^2 u_z^2 - \frac{1}{4}\rho u_x u_y^2 u_z - \frac{1}{4}\rho u_x^2 u_y^2 u_z - \frac{1}{4}\rho u_x u_y^2 u_z^2 - \frac{1}{4}\rho u_x^2 u_y^2 u_z^2 \\
f_{20}^{eq} = & \frac{1}{54}\rho + \frac{1}{18}\rho u_x - \frac{1}{18}\rho u_z + \frac{1}{18}\rho u_x^2 - \frac{1}{36}\rho u_y^2 + \frac{1}{18}\rho u_z^2 - \frac{1}{6}\rho u_x u_z + \frac{1}{6}\rho u_x u_z^2 - \frac{1}{6}\rho u_x^2 u_z - \frac{1}{12}\rho u_x u_y^2 + \frac{1}{12}\rho u_y^2 u_z \\
& - \frac{1}{12}\rho u_x^2 u_y^2 + \frac{1}{6}\rho u_x^2 u_z^2 - \frac{1}{12}\rho u_y^2 u_z^2 + \frac{1}{4}\rho u_x u_y^2 u_z + \frac{1}{4}\rho u_x^2 u_y^2 u_z - \frac{1}{4}\rho u_x u_y^2 u_z^2 - \frac{1}{4}\rho u_x^2 u_y^2 u_z^2 \\
f_{21}^{eq} = & \frac{1}{54}\rho + \frac{1}{18}\rho u_y + \frac{1}{18}\rho u_z - \frac{1}{36}\rho u_x^2 + \frac{1}{18}\rho u_y^2 + \frac{1}{18}\rho u_z^2 + \frac{1}{6}\rho u_y u_z + \frac{1}{6}\rho u_y^2 u_z + \frac{1}{6}\rho u_y u_z^2 - \frac{1}{12}\rho u_x^2 u_y - \frac{1}{12}\rho u_x^2 u_z \\
& - \frac{1}{12}\rho u_x^2 u_y^2 - \frac{1}{12}\rho u_x^2 u_z^2 + \frac{1}{6}\rho u_y^2 u_z^2 - \frac{1}{4}\rho u_x^2 u_y u_z - \frac{1}{4}\rho u_x^2 u_y^2 u_z - \frac{1}{4}\rho u_x^2 u_y u_z^2 - \frac{1}{4}\rho u_x^2 u_y^2 u_z^2 \\
f_{22}^{eq} = & \frac{1}{54}\rho + \frac{1}{18}\rho u_y - \frac{1}{18}\rho u_z - \frac{1}{36}\rho u_x^2 + \frac{1}{18}\rho u_y^2 + \frac{1}{18}\rho u_z^2 - \frac{1}{6}\rho u_y u_z - \frac{1}{6}\rho u_y^2 u_z + \frac{1}{6}\rho u_y u_z^2 - \frac{1}{12}\rho u_x^2 u_y + \frac{1}{12}\rho u_x^2 u_z \\
& - \frac{1}{12}\rho u_x^2 u_y^2 - \frac{1}{12}\rho u_x^2 u_z^2 + \frac{1}{6}\rho u_y^2 u_z^2 + \frac{1}{4}\rho u_x^2 u_y u_z + \frac{1}{4}\rho u_x^2 u_y^2 u_z - \frac{1}{4}\rho u_x^2 u_y u_z^2 - \frac{1}{4}\rho u_x^2 u_y^2 u_z^2
\end{aligned}$$

$$\begin{aligned}
f_{23}^{eq} = \; & \frac{1}{216}\rho + \frac{1}{72}\rho u_x + \frac{1}{72}\rho u_y + \frac{1}{72}\rho u_z + \frac{1}{72}\rho u_x^2 + \frac{1}{72}\rho u_y^2 + \frac{1}{72}\rho u_z^2 + \frac{1}{24}\rho u_x u_y + \frac{1}{24}\rho u_x u_z + \frac{1}{24}\rho u_y u_z \\
& + \frac{1}{24}\rho u_x^2 u_y + \frac{1}{24}\rho u_x^2 u_z + \frac{1}{24}\rho u_x u_y^2 + \frac{1}{24}\rho u_y^2 u_z + \frac{1}{24}\rho u_x u_z^2 + \frac{1}{24}\rho u_y u_z^2 + \frac{1}{8}\rho u_x u_y u_z \\
& + \frac{1}{24}\rho u_x^2 u_y^2 + \frac{1}{24}\rho u_x^2 u_z^2 + \frac{1}{24}\rho u_y^2 u_z^2 + \frac{1}{8}\rho u_x^2 u_y u_z + \frac{1}{8}\rho u_x u_y^2 u_z + \frac{1}{8}\rho u_x u_y u_z^2 \\
& + \frac{1}{8}\rho u_x^2 u_y^2 u_z + \frac{1}{8}\rho u_x^2 u_y u_z^2 + \frac{1}{8}\rho u_x u_y^2 u_z^2 + \frac{1}{8}\rho u_x^2 u_y^2 u_z^2 \\
f_{24}^{eq} = \; & \frac{1}{216}\rho + \frac{1}{72}\rho u_x + \frac{1}{72}\rho u_y - \frac{1}{72}\rho u_z + \frac{1}{72}\rho u_x^2 + \frac{1}{72}\rho u_y^2 + \frac{1}{72}\rho u_z^2 + \frac{1}{24}\rho u_x u_y - \frac{1}{24}\rho u_x u_z - \frac{1}{24}\rho u_y u_z \\
& + \frac{1}{24}\rho u_x^2 u_y - \frac{1}{24}\rho u_x^2 u_z + \frac{1}{24}\rho u_x u_y^2 - \frac{1}{24}\rho u_y^2 u_z + \frac{1}{24}\rho u_x u_z^2 + \frac{1}{24}\rho u_y u_z^2 - \frac{1}{8}\rho u_x u_y u_z \\
& + \frac{1}{24}\rho u_x^2 u_y^2 + \frac{1}{24}\rho u_x^2 u_z^2 + \frac{1}{24}\rho u_y^2 u_z^2 - \frac{1}{8}\rho u_x^2 u_y u_z - \frac{1}{8}\rho u_x u_y^2 u_z + \frac{1}{8}\rho u_x u_y u_z^2 \\
& - \frac{1}{8}\rho u_x^2 u_y^2 u_z + \frac{1}{8}\rho u_x^2 u_y u_z^2 + \frac{1}{8}\rho u_x u_y^2 u_z^2 + \frac{1}{8}\rho u_x^2 u_y^2 u_z^2 \\
f_{25}^{eq} = \; & \frac{1}{216}\rho + \frac{1}{72}\rho u_x - \frac{1}{72}\rho u_y + \frac{1}{72}\rho u_z + \frac{1}{72}\rho u_x^2 + \frac{1}{72}\rho u_y^2 + \frac{1}{72}\rho u_z^2 - \frac{1}{24}\rho u_x u_y + \frac{1}{24}\rho u_x u_z - \frac{1}{24}\rho u_y u_z \\
& - \frac{1}{24}\rho u_x^2 u_y + \frac{1}{24}\rho u_x^2 u_z + \frac{1}{24}\rho u_x u_y^2 + \frac{1}{24}\rho u_y^2 u_z + \frac{1}{24}\rho u_x u_z^2 - \frac{1}{24}\rho u_y u_z^2 - \frac{1}{8}\rho u_x u_y u_z \\
& + \frac{1}{24}\rho u_x^2 u_y^2 + \frac{1}{24}\rho u_x^2 u_z^2 + \frac{1}{24}\rho u_y^2 u_z^2 - \frac{1}{8}\rho u_x^2 u_y u_z + \frac{1}{8}\rho u_x u_y^2 u_z - \frac{1}{8}\rho u_x u_y u_z^2 \\
& + \frac{1}{8}\rho u_x^2 u_y^2 u_z - \frac{1}{8}\rho u_x^2 u_y u_z^2 + \frac{1}{8}\rho u_x u_y^2 u_z^2 + \frac{1}{8}\rho u_x^2 u_y^2 u_z^2 \\
f_{26}^{eq} = \; & \frac{1}{216}\rho + \frac{1}{72}\rho u_x - \frac{1}{72}\rho u_y - \frac{1}{72}\rho u_z + \frac{1}{72}\rho u_x^2 + \frac{1}{72}\rho u_y^2 + \frac{1}{72}\rho u_z^2 - \frac{1}{24}\rho u_x u_y - \frac{1}{24}\rho u_x u_z + \frac{1}{24}\rho u_y u_z \\
& - \frac{1}{24}\rho u_x^2 u_y - \frac{1}{24}\rho u_x^2 u_z + \frac{1}{24}\rho u_x u_y^2 - \frac{1}{24}\rho u_y^2 u_z + \frac{1}{24}\rho u_x u_z^2 - \frac{1}{24}\rho u_y u_z^2 + \frac{1}{8}\rho u_x u_y u_z \\
& + \frac{1}{24}\rho u_x^2 u_y^2 + \frac{1}{24}\rho u_x^2 u_z^2 + \frac{1}{24}\rho u_y^2 u_z^2 + \frac{1}{8}\rho u_x^2 u_y u_z - \frac{1}{8}\rho u_x u_y^2 u_z - \frac{1}{8}\rho u_x u_y u_z^2 \\
& - \frac{1}{8}\rho u_x^2 u_y^2 u_z - \frac{1}{8}\rho u_x^2 u_y u_z^2 + \frac{1}{8}\rho u_x u_y^2 u_z^2 + \frac{1}{8}\rho u_x^2 u_y^2 u_z^2
\end{aligned}$$

The relaxation frequencies can be expressed by the following block diagonal matrix:

$$\mathbf{\Lambda} = \mathrm{diag}\left(1, 1, 1, 1, \tau_f^{-1}, \tau_f^{-1}, \tau_f^{-1}, \begin{bmatrix} s_+ & s_- & s_- \\ s_- & s_+ & s_- \\ s_- & s_- & s_+ \end{bmatrix} \omega_3, \omega_3, \omega_3, \omega_3, \omega_3, \omega_3, 1, \omega_4, \omega_4, \omega_4, 1, 1, 1, 1, 1, 1, 1\right)$$

where $s_+ = \frac{1}{3}\left(\tau_{f,bulk}^{-1} + 2\tau_f^{-1}\right)$ and $s_- = \frac{1}{3}\left(\tau_{f,bulk}^{-1} - \tau_f^{-1}\right)$. The bulk viscosity can be related to the associated relaxation time by:

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

Guo forcing can be applied using the following central moment transformations of the associated source terms:

$$\vec{S}^{m} = \begin{pmatrix} 0 \\ F_x \\ F_y \\ F_z \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \left(\frac{1}{3} - v_y^2\right) F_x - 2v_x v_y F_y \\ \left(\frac{1}{3} - v_z^2\right) F_x - 2v_x v_z F_z \\ \left(\frac{1}{3} - v_x^2\right) F_y - 2v_x v_y F_x \\ \left(\frac{1}{3} - v_x^2\right) F_z - 2v_x v_z F_x \\ \left(\frac{1}{3} - v_z^2\right) F_y - 2v_y v_z F_z \\ \left(\frac{1}{3} - v_y^2\right) F_z - 2v_y v_z F_y \\ -v_y v_z F_x + v_x v_z F_y + v_x v_y F_z \\ 4v_x v_y \left(v_y F_x + v_x F_y\right) \\ 4v_x v_z \left(v_z F_x + v_x F_z\right) \\ 4v_y v_z \left(v_z F_y + v_y F_z\right) \\ 2v_x \left(2v_y v_z F_x + v_x v_z F_y + v_x v_y F_z\right) \\ 2v_y \left(v_y v_z F_x + 2v_x v_z F_y + v_x v_y F_z\right) \\ 2v_z \left(v_y v_z F_x + v_x v_z F_y + 2v_x v_y F_z\right) \\ F_x \left(\frac{1}{9} - \frac{1}{3}v_y^2 - \frac{1}{3}v_z^2 - 3v_y^2 v_z^2\right) - \left(v_z F_y + v_y F_z\right)\left(\frac{2}{3}v_x + 6v_x v_y v_z\right) \\ F_y \left(\frac{1}{9} - \frac{1}{3}v_x^2 - \frac{1}{3}v_z^2 - 3v_x^2 v_z^2\right) - \left(v_z F_x + v_x F_z\right)\left(\frac{2}{3}v_y + 6v_x v_y v_z\right) \\ F_z \left(\frac{1}{9} - \frac{1}{3}v_x^2 - \frac{1}{3}v_y^2 - 3v_x^2 v_y^2\right) - \left(v_y F_x + v_x F_y\right)\left(\frac{2}{3}v_z + 6v_x v_y v_z\right) \\ \frac{4}{3}v_x v_y \left(v_y F_x + v_x F_y\right) + \frac{4}{3}v_x v_z \left(v_z F_x + v_x F_z\right) + \frac{4}{3}v_y v_z \left(v_z F_y + v_y F_z\right) + 8v_x v_y v_z \left(v_y v_z F_x + v_x v_z F_y + v_x v_y F_z\right) \end{pmatrix},$$

and He forcing can be applied using these central moment terms:

$$\vec{S}^{m} = \begin{pmatrix} 0 \\ F_x \\ F_y \\ F_z \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{3}F_x \\ \frac{1}{3}F_x \\ \frac{1}{3}F_y \\ \frac{1}{3}F_z \\ \frac{1}{3}F_y \\ \frac{1}{3}F_z \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{9}F_x \\ \frac{1}{9}F_y \\ \frac{1}{9}F_z \\ 0 \end{pmatrix}.$$

CHAPTER

NINE

DL_MESO_DPD PROGRAMMING BACKGROUND

9.1 Basic concepts

DL_MESO_DPD is a particle dynamics solver specialising in using pairwise thermostats (most notably Dissipative Particle Dynamics, DPD) to model systems of freely-moving spherical particles that interact with each other. The governing equations for DPD are similar to those for classical molecular dynamics (MD):

$$\frac{d\vec{v}_i}{dt} = \frac{\vec{F}_i}{m_i} \tag{9.1}$$

$$\frac{d\vec{r}_i}{dt} = \vec{v}_i \tag{9.2}$$

where $\vec{r}_i$, $\vec{v}_i$ and $\vec{F}_i$ are respectively the position, velocity and force for particle i, which has mass m_i. To evolve these properties over time, numerical integration of (9.1) and (9.2) over a timestep Δt is required. The most frequently used integration scheme for DPD is Velocity Verlet, which is carried out over two stages. The first stage advances the particle velocities by half a timestep and uses these values to advance the positions to the end of the timestep:

$$\vec{v}_i\left(t + \tfrac{1}{2}\Delta t\right) = \vec{v}_i\left(t\right) + \frac{\Delta t}{2}\frac{\vec{F}_i\left(t\right)}{m_i}$$

$$\vec{r}_i\left(t + \Delta t\right) = \vec{v}_i\left(t\right) + \Delta t \vec{v}_i\left(t + \tfrac{1}{2}\Delta t\right)$$

After adjusting them due to any boundary conditions, the new particle positions are used to calculate forces at the end of the timestep, $\vec{F}_i\left(t + \Delta t\right)$. These are then used to complete integration of particle velocities to the end of the timestep in the second stage:

$$\vec{v}_i\left(t + \Delta t\right) = \vec{v}_i\left(t + \tfrac{1}{2}\Delta t\right) + \frac{\Delta t}{2}\frac{\vec{F}_i\left(t + \Delta t\right)}{m_i}.$$

The forces calculated for each particle in a DPD simulation are typically sums of pairwise forces:

$$\vec{F}_i = \sum_{j\neq i}^{N}\left(\vec{F}_{ij}^{C} + \vec{F}_{ij}^{D} + \vec{F}_{ij}^{R}\right) \tag{9.3}$$

where $\vec{F}_{ij}^{C}$, $\vec{F}_{ij}^{D}$ and $\vec{F}_{ij}^{R}$ are respectively the *conservative* (interaction), *dissipative* (drag) and *random* forces acting on particle i due to the presence of particle j. The dissipative force is defined as:

$$\vec{F}_{ij}^{D} = -\gamma_{ij} w^{D}\left(r_{ij}\right)\left(\vec{r}_{ij}\cdot\vec{v}_{ij}\right)\frac{\vec{r}_{ij}}{r_{ij}^{2}} \tag{9.4}$$

where $\vec{r}_{ij}$ is the vector between the two particles, w^{D} is a distance-dependent switching function used to impose a finite limit on the force's value and $\vec{v}_{ij}$ is the relative velocity between the particles[1]. Similarly, the random force is defined as:

$$\vec{F}_{ij}^{R} = \sigma_{ij} w^{R}\left(r_{ij}\right)\zeta_{ij}\Delta t^{-\frac{1}{2}}\frac{\vec{r}_{ij}}{r_{ij}} \tag{9.5}$$

[1] The vector and relative velocity between a pair of particles is defined as differences in position and velocity respectively. Which particle's property is subtracted from the other's does not matter, although that choice needs to be applied consistently.

where w^R is another distance-dependent switching function and ζ_{ij} is a Gaussian random number with zero mean and unit variance for the particle pair[2]. These two forces act as a momentum-conserving (Galilean invariant) thermostat when:

$$\sigma_{ij}^2 = 2\gamma_{ij} k_B T$$

and

$$w^D\left(r_{ij}\right) = \left[w^D\left(r_{ij}\right)\right]^2.$$

9.2 Parallelisation strategies

DL_MESO_DPD consists of two methods to divide computational work among the available processor cores and threads:

- Equipartition domain decomposition
- Multithreading of force calculations

Domain decomposition involves the division of computational work among the available processor cores, with each core carrying out a section of the calculation with as little input from other cores as possible. This strategy is known to work well for particle dynamics calculations [124], since the most computational intensive parts are typically calculation of interaction forces and integration of these forces to determine particle motion. As such, each processor core is assigned its own section of the box volume - described as a *subdomain* - and carries out force calculations primarily for and integration solely to the particles that exist in that subdomain. If any particles leave the subdomain during force integration, these are *deported* to neighbouring processor cores via MPI communication. Dividing the volume equally among the available processor cores - *domain_decompose()* - normally ensures each core holds roughly the same number of particles and thus carries out the same amount of computational work. This form of *equipartition* domain decomposition works well when the particle density throughout the box volume is roughly equal and each processor core holds approximately the same numbers of particles[3].

The domain decomposition strategy underpinning DL_MESO_DPD is based on the *link cell algorithm* [57] for calculating pairwise potentials and forces between particles with distances between them that are within a relatively short cutoff value. Each subdomain is subsequently divided into smaller cells with sides no less than the interaction cutoff distance and for each particles in each link cell, all other particles within the cutoff distance will be found in either the same cell or a nearest-neighbouring cell *but no further*. Even with the need to construct lists of particles in each cell before carrying out force calculations, this approach makes pairwise force calculations scale linearly with the number of particles as opposed to quadratic scaling for a more naïve search through all possible particle pairs.

To ensure the forces for all particles in a subdomain are calculated correctly, a *boundary halo* is defined to hold data for particles in neighbouring subdomains that could be involved in interactions with the current subdomain's particles. The halo extends no less than the interaction cutoff distance beyond the current subdomain and the particles contained therein are also assigned into link cells: although these are not included in the main loop through the link cells for a given subdomain, they are included and required as nearest-neighbour cells. (The boundary halo size may be set larger than the cutoff distance to ensure bonded interactions are dealt with correctly, but only one link cell is needed for pairwise force calculations.) An *export* step is carried out to fill the boundary halo: for parallel running, this involves MPI communications with nearest neighbouring processor cores to exchange data for particles close to the boundaries, while an equivalent for serial running copies data for particles close to the simulation box edges into the same arrays. Both parallel and serial versions of DL_MESO_DPD therefore require arrays for holding particle positions, velocities and forces (among other properties) to be large enough for both each subdomain's particles *and* those in the boundary halo.

[2] The use of the Gaussian random number with the timestep size Δt provides a means to calculate a Wiener increment needed for this force. The Gaussian random number itself can either be generated using a transform process for uniformly-distributed random numbers $0 \leq u_{ij} < 1$ or approximated using the uniform random numbers directly: $\zeta_{ij} \approx \sqrt{12}\left(u_{ij} - \frac{1}{2}\right)$ [26].

[3] A limited number of DPD simulations may be less well-suited to this form of domain decomposition due to very uneven particle distributions (e.g. those involving many-body DPD interactions). This does not prevent these calculations from being run in DL_MESO_DPD at all, although they are less likely to perform or scale well due to greater communication latency and memory demands from some cores needing to hold more particles at a time.

When working through the link cells in a given subdomain, only half of the nearest-neighbour cells are normally searched to avoid double-counting of interacting particle pairs. For link cells at the edge of the subdomain, this search strategy can either (i) be maintained to ensure each pair is only ever counted once, or (ii) be modified to include all pairs that involve particles in the boundary halo. In the case of the half-search strategy (i), forces for particles in the boundary halo subsequently have to be sent back to originating processor cores using an *import* communication step, while the full-search strategy (ii) does not require the communication as the forces for boundary halo particles will be calculated correctly in their originating cores due to the double-counting involved. In DL_MESO_DPD, the half-search strategy with import communication is used for the main force calculations due to difficulties in calculating identical pairwise DPD random forces (or relative velocity corrections for other thermostats) on multiple processor cores[4], while the full-search strategy is used for deterministic calculations (i.e. localised density calculations for many-body DPD, recalculations of dissipative forces for the DPD Velocity Verlet scheme) and those where random forces can be calculated consistently on multiple cores (i.e. integration of DPD dissipative and random forces for Shardlow splitting).

Additional acceleration of the main force calculations can be achieved in DL_MESO_DPD by dividing the subdomain's link cells among available threads. This is achieved in the OpenMP versions of modules involved in force calculations (e.g. *field_module.F90*) with OpenMP directives on the main loops over the link cells, which use previously-created lists of link cells for the subdomain and their neighbours to avoid using nested loops representing each Cartesian direction. Since different threads may attempt to update a particular particle's force simultaneously, two strategies are available in DL_MESO_DPD to avoid this kind of race condition:

1. Allocate temporary force arrays to enable each thread to assign forces to particles separately, which are combined together afterwards (default); or
2. Use of a OpenMP critical section to only allow one thread at a time to assign forces to particles.

The first (default) option requires additional memory per thread for assigning forces, which can adversely impact computational performance: DL_MESO_DPD can therefore restrict the number of OpenMP threads used in force calculations based on available memory for the temporary arrays. The second option enables better scaling with larger numbers of OpenMP threads at the cost of assigning forces one thread at a time: this option can be selected using the `openmp critical` directive in the *CONTROL* file.

9.3 Data storage

DL_MESO_DPD uses two-dimensional arrays to store its main calculation properties:

- *xxyyzz* - Particle positions $\vec{x}_i$
- *vxvyvz* - Particle velocities $\vec{v}_i$
- *fxfyfz* - (Most) particle forces $\vec{F}_i$
- *vfxfyfz* - Variable particle forces
- *cfxfyfz* - Corrective forces for frozen particles to remove reciprocal-space Ewald sum contributions

where the variable particle forces are used for force integration schemes or thermostats that require separate contributions to other forces: dissipative forces $\vec{F}_i^D$ for DPD Velocity Verlet, dissipative and random forces $\vec{F}_i^D + \vec{F}_i^R$ for DPD with Shardlow splitting, and pairwise Nosé-Hoover thermostatting forces $\vec{F}_i^T$ for the Stoyanov-Groot thermostat. Each of the arrays are defined with the first index representing the Cartesian coordinates (1 = x, 2 = x, 3 = x), with the maximum size equal to the constant *csize* to improve cache usage when reading from or writing values to memory for individual particles, and the second index defined with a maximum value of *maxdim* (the calculated maximum number of particles per processor core). Each array is also associated with three pointers that can be used to represent each individual Cartesian dimension as a one-dimensional array, e.g. *xxx*, *yyy* and *zzz* for *xxyyzz*, the use of which helps make the code more readable. An additional array *rhomb* is also defined for many-body DPD interactions and used to store localised densities for each particle on a species basis.

[4] These are due to the choice of random number generator, whose state cannot often be synchronised on two or more processor cores to generate identical random numbers for given particle pairs. Random number generators that can accept particle and timestep numbers as seeds (e.g. Saru [1]) enable the full-search strategy to be used for these force calculations, but other calculations (e.g. bond interactions) have been designed with the half-search strategy in mind and would require substantial modifications to full exploit this change.

Other properties for individual particles are stored as one-dimensional integer arrays, each allocated to the size of *maxdim*:

- *lab* - Global particle index (unique number to identify particle)
- *ltp* - Species (type) for particle
- *ltm* - Molecule type for particle
- *lmp* - Processor core currently holding particle
- *loc* - Local index for particle on processor core currently holding it

The global particle index is used to identify each unique particle and can range from 1 to *nsyst*: these are ordered to put *nusyst* non-molecular (e.g. solvent) particles first before any particles involved in molecules, which is used to distinguish between the two particle classifications. Each processor core holds *nbeads* particles often in a different localised order, which can vary as particles move between cores' subdomains: if any frozen particles exist in the subdomain (which do not move), these occupy the first *nfbeads* entries in any particle property arrays and enables them to be skipped easily in e.g. force integration.

The species types allow properties such as mass (*amass*), charge (*chge*), whether frozen or not (*lfrzn*), and name (*namspe*) to be specified without needing to store them separately for each particle. The molecule types do the same for molecule names (*nammol*) and potentially can identify other molecule properties (e.g. numbers of particles - *nbdmol* - or bonds per molecule, *nbond*), although only the names are used during a DL_MESO_DPD simulation. The *lmp* and *loc* arrays are mainly used to identify particles in boundary halos for import communications when summing up contributions to particle forces and putting together lists of particle pairs used for pairwise thermostats other than DPD (e.g. Lowe-Andersen).

Aside from indicating properties of individual particle species, arrays exist to identify parameters for interactions:

- *vvv* - Parameters for conservative interactions (other than bonds and electrostatics)
- *vvsrf* - Parameters for surface interactions
- *aabond*, *bbbond*, *ccbond*, *ddbond* - Parameters for (stretching) bond interactions
- *aaang*, *bbang*, *ccang*, *ddang* - Parameters for bond angle interactions
- *aadhd*, *bbdhd*, *ccdhd*, *dddhd* - Parameters for bond dihedral interactions

of which *vvv* is a two-dimensional array defined for a maximum number of parameters (*mxprm*) and the number of possible pairs of species (*npot*), *vvsrf* is a two-dimensional array for a maximum number of parameters (:ref`mxsprm`) and the number of particle species (*nspe*), *aabond* etc. for the number of unique bond types (*nbonddef*), *aaang* etc. for the number of unique angle types (*nangdef*), and *aadhd* etc. for the number of unique dihedral types (*ndhddef*). The kinds of these interaction types can be found in *ktype*, *srfktype*, *bdtype*, *angtype* and *dhdtype* respectively.

While the particle properties themselves can be used to determine the conservative, surface and electrostatic interactions between pairs, book-keeping arrays (*bndtbl*, *angtbl* and *dhdtbl*) are required to keep track of which particles are involved in bond, angle and dihedral interactions between pairs, triples and quadruples of particles respectively. Derived from the molecule data specified in the *FIELD* file, these arrays are constructed to identify the particles by global numbers. Each processor core can potentially hold every bond, angle and/or dihedral in the simulation box, although this is less likely to happen if the default method of dealing with bonded interactions is in use. An array *lblclst* is used to find local particle indices from global particle numbers.

9.4 Communications

The three main types of communication in DL_MESO_DPD are:

- deport - Movement of particles to neighbouring processor cores after first Velocity Verlet integration stage
- export - Copying of particle data from neighbouring processor cores into boundary halo for force calculations
- import - Summation of all force contributions for particles in boundary halos

Each main communication type is invoked during calculations with the subroutines *deportdata()*, *exportdata()* and *importdata()* respectively. Variations of *exportdata()* exist for many-body DPD localised density calculations - *exportdensitydata()* - and to update particle velocities for Shardlow splitting and recalculation of dissipative forces for DPD Velocity Verlet - *exportvelocitydata()* - while variations of *importdata()* are available for DPD Velocity Verlet integration and the Stoyanov-Groot thermostat, *importdata_dpdvv()* and (*importdata_stoyanov()* respectively, both of which require two separated sets of particle forces to be brought together.

These subroutines all represent common interfaces for parallel and serial running, where the contents of each subroutine vary depending on the version of DL_MESO_DPD and the associated form of *domain_module.F90* in use. While the parallel version of DL_MESO_DPD includes explicit subroutines to deport, export and import particle data in a given direction, the serial version only requires subroutines for the export step (which copies particles into the boundary halo from the opposite side of the box): both particle deport (i.e. adjusting positions for particles moving across periodic boundaries) and force import in this case can be carried out using simple loops over the required particles, which form the content of the serial versions of *deportdata()* and *importdata()* respectively.

Each communication type consists of at least six MPI communications (or copying of particle data for serial calculations) corresponding to Cartesian directions - $-x$ (1), $+x$ (2), $-y$ (3), $+y$ (4), $-z$ (5) and $+z$ (6) - and carried out in pairs for each axis (e.g. $-x$ and $+x$). The numbers of the processor cores corresponding to these directions are assigned to the array *map* by the *domain_decompose()* subroutine. Each set of communications includes checks for all particles - including those already in the boundary halo[5] - to see if they are in range for being sent to the neighbouring processor core: checking particles already in the boundary halo ensures that the edges and corners are also dealt with correctly, bringing in particles from cores with subdomains at diagonals to the current one that are not directly involved in communications. If reflecting surfaces are required orthogonal to particular axes, a switch is used with the single-direction subroutines, e.g. `skip` in *export*, to not send particles in a particular direction. Variant subroutines are also available - e.g. *export_shear()* to apply the communication for a Lees-Edwards shearing boundary: these enable particles to be shifted laterally by a time-dependent distance, which leads to data being sent to and received by up to four other processor cores.

The MPI communications themselves involve the creation of buffer arrays to pack/send the particle data in the given direction and to receive/unpack particle data coming from another processor core. While a standard communication requires one buffer array to send data and another to receive, the shearing variants require four of each (the maximum number of cores to which particles could be sent at a given timestep). All of these are defined using the *commsoutbuf* and *commsinbuf* arrays, which are associated with pointers when the communication subroutine starts that are nullified at the end. After packing the send arrays with the required particle data (converting integers to double-precision real numbers as a single data type is normally required for an MPI communication), unblocked MPI send and receive commands are used along with a message wait command near the very end of the subroutine: these allow each processor core to unpack and process the data it receives while simultaneously sending its own data and reduce latency (delays) as a result.

For the deport and export steps, the message wait command can include an enquiry about the size of the incoming array to check the receiving core has enough space in memory to store the particle data. The main export subroutines (for particle data required to calculate forces) keep track of the numbers of particles received and use the maximum value to determine if the buffer arrays need to be resized: the resizing by *resize_buffer()* can take place if many-body DPD interactions and/or a barostat are in use or while the system is equilibrating.

All communication subroutines in *domain_module.F90* use at least one unique message tag to identify the MPI messages (data arrays) being sent and received. In the case of the variant subroutines to apply Lees-Edwards

[5] When checking particle positions, the deport step uses a larger boundary halo size than the export and import steps - the size of the subdomain in the particle direction - to ensure particles that have travelled further away than the usual boundary halo size in that timestep are correctly dealt with. (This effectively sets a maximum possible particle velocity that DL_MESO_DPD can manage, $\frac{\min(L_x, L_y, L_z)}{\Delta t}$)

shearing, an additional 'switch' (`shft`) is used to enable the same subroutines to be called twice by processor cores (for smaller systems with one core in the Cartesian axis of the shearing boundary) by using additional message tags, avoiding conflicts in communications going in both directions.

The communication subroutines - *deportdata()*, *exportdata()*, *importdata()* etc. - all keep track of the changing number of particles each processor core holds in memory as the integer variable `nlimit`. This number is used to update the total number of particles in the processor core - *nbeads* - after the deport step, while it is also used after the export step when constructing linked-cell lists in *parlnk()*.

The deport step *deportdata()* sends all data for particles leaving the current subdomain: positions, velocities, global particle indices, species and molecule types, plus all bonds, angles and dihedrals associated with each particle. (No forces are required since these will shortly be updated.) The main export step *exportdata()* sends particle positions, velocities, global particle indices, local particle indices in their original subdomains, species and molecule types, plus localised densities if many-body DPD interactions are required. The export step before calculating localised densities *exportdensitydata()* sends particle positions, global particle indices, local particle indices in their original subdomains and species, while the export step to update velocities *exportvelocitydata()* sends particle positions, velocities, global and local particle indices, species and molecule types. The main import step *importdata()* sends particle positions, forces, global and local particle indices, and original processor core numbers: the variant import steps for DPD Velocity Verlet and the Stoyanov-Groot thermostat additionally include the variable forces.

9.5 Linked-cell lists for pairwise force calculations

The subroutine *parlnk()* is used to construct the linked-cell lists required for pairwise force calculations. Three sets of linked-cell lists are constructed by DL_MESO_DPD for different purposes:

- Main force calculations (e.g. conservative, dissipative and random DPD forces)
- Electrostatic (real-space Ewald sum) calculations with a larger cutoff distance
- Localised density calculations for many-body DPD with a smaller cutoff distance

with their parameters (e.g. numbers of cells in each direction for the subdomain) set and the required arrays allocated in the subroutine *domain_dimensions()*. These include arrays listing the link cells included in each subdomain (excluding the boundary halo), the neighbouring link cells for each link cell in all 27 possible directions (including itself) and (for main force calculations only) whether or not the neighbouring cell crosses a Lees-Edwards shearing boundary. A switch is also set for each set of linked-cell lists to determine if the simulation is small enough to only have one link cell in any direction, which is used to apply more strigent conditions for finding particle pairs to avoid double-counting.

Two arrays are assigned during the *parlnk()* subroutine: one indicating the first particle in each cell, and the other giving subsequent particles in the current list's cell (terminating with 0). The numbers of particles in each cell are counted when these arrays are constructed and the maximum number reported, which is used when going through the lists for force calculations. A switch is available (`typ`) to indicate that only charged particles should be assigned for Ewald sum calculations. After the lists are constructed, each cell is checked to see if any infinite loops have inadvertently been created, which are subsequently broken open.

The subroutines for calculating pairwise forces or localised densities start by allocating arrays to directly list the particles in a given link cell (using the maximum number of particles per link cell to determine its size), to count the numbers of particles in a current link cell and its neighbours, to identify the particles pairs that are within the cutoff distance, and to store the vector and squared distance for each pair. The subroutines then loop through all the link cells in the subdomain (excluding boundary halos), load in the particles in the current link cell - e.g. *loadpart()* - and then loop through the neighbouring cells to load in their particles: the total numbers of particles from all of these link cells are then used to check the arrays for pair data are sufficiently large enough (and resized if not). All of the interacting pairs (excluding those where both particles are frozen) within required the cutoff distance(s) are then found[6] - using a subroutine, e.g. *diff()* - and the particle numbers and vectors are loaded into their respective arrays, before the main subroutine goes through the list of pairs, calculates and assigns the relevant forces or localised densities. (See *above* for details of how many neighbouring cells are searched and the consequences of this choice.)

[6] Since forces are divided by the scalar distance between a pair of particles, pairs that are closer than a minimum distance of 10^{-8} in DPD length units are omitted to avoid divisions by zero.

9.6 Force calculations

The main force calculations are carried out with subroutines in *field_module.F90*. Two main sets of subroutines exist in this module: the routines dedicated to calculating the main pairwise forces - e.g. *forces_mdvv()* - and those that apply core-to-core communications, set up linked-cell lists and launch the calculations, e.g. *plcfor_mdvv()*. Versions of each subroutine are available for the different thermostats and integrators - DPD with standard (MD) Velocity Verlet, DPD with DPD Velocity Verlet, DPD with Shardlow splitting, Lowe-Andersen, Peters, Stoyanov-Groot - which each calculate and assign particle forces differently.

The setup subroutines start by assigning local particle indices to the array *loc* and the current processor core number to *lmp*: if a particle belongs to a defined molecule, the global and local particle indices are also added to the list *lblclst*. If any many-body DPD calculations are required, the localised densities are then calculated. The *exportdata()* subroutine is then called to create the required boundary halos before *parlnk()* constructs the required linked-cell lists, using the maximum interaction cutoff distance *rcut* as the minimum cuboidal size and the arrays *lct* and *link* to store the first particles in each cell and the subsequent particles in the current cell.

The pairwise force subroutine is then called, which calculates the main conservative forces on particle pairs within the main interaction cutoff distance *rcut* and, if required, the dissipative and random forces for the DPD thermostat for particle pairs within the cutoff distance *rtcut*. The *conservativeforce()* subroutine calculates the various forms of conservative interaction forces and potentials[7] available in DL_MESO_DPD for a particle pair: the forces are expressed as the scalar quantity divided by the distance between the particles, $\frac{F_{ij}^C}{r_{ij}}$, to reduce the number of divisions required for calculating the product of scalar force and unit vector. While all three forces are assigned to the main force arrays in *forces_mdvv()*, the dissipative forces are assigned to the variable force arrays in *forces_dpdvv()* as these are later recalculated by the *dragforces_dpdvv()* subroutine (which does not require an import communication afterwards). The dissipative and random forces are not required for DPD with Shardlow splitting as the random and dissipative forces are integrated separately in *shardlow_integrate()*, while other pairwise thermostats (Lowe-Andersen, Peters, Stoyanov-Groot) alternatively put together lists of particle pairs that need their relative velocities modified during the second Velocity Verlet integration stage. The Stoyanov-Groot thermostat also calculates additional pairwise Nosé-Hoover thermostatting forces for particle pairs not included in the list that depend on the instantaneous temperature: this property is also calculated along with the pairwise forces and used to modify the thermostatting forces afterwards in the variable force array. The forces are used to calculate the stress tensors (separated into potential, dissipative and random components here) as well as the system and instantaneous virials.

If many-body DPD interactions are involved, the self-energy potentials are then calculated. Any electrostatic forces (via Ewald sums or SPME), bonded and surface/wall forces are subsequently calculated, before the import communication step is applied to send force contributions for particles in boundary halos back to their original processor cores in order to sum up the total forces on those particles. Any frozen particles have their forces and velocities reset to zero.

When a simulation is initialised, the *plcfor_initial()* subroutine is used to carry out a similar set of communications and calculations as for the main simulation to calculate forces (if not supplied in a *CONFIG* file), potential energies, virials, stress tensors etc. for the system at its initial state. The *forces_mdvv()* subroutine is called if both the conservative and DPD thermostat (dissipative and random) forces are needed (for DPD with MD and DPD Velocity Verlet integration), or the *forces_shardlow()* subroutine is called if only the conservative forces are required, along with the standard subroutines to calculate electrostatic, bonded and surface/wall forces: if forces are not required, alternative subroutines to calculate potentials etc. *without* the forces, e.g. *potentials_initial()*, are used to obtain the remaining properties.

[7] It should be noted that any thermostatting forces (e.g. dissipative and random forces for the DPD thermostat) do *not* contribute to the system potential energy. They do contribute to the system virial and resulting pressure, but these are not included for the instantaneous values used with barostats.

9.7 Many-body DPD

The main forces for many-body DPD interactions are calculated along with other conservative, dissipative and random forces, although these require localised densities to be calculated for each particle. The *local_density()* subroutine carries out a search for particle pairs within the many-body DPD cutoff distance *rmbcut* and adds values of a weighting function - given in *weight_rho()* - to the localised densities for each particle in the pair, assigning the contribution based on the other particle's species. The search for particle pairs includes all pairs with particles in the boundary halo and thus no import communication step afterwards is required.

The subroutine *manybody_potential()* calculates self-energies for particles with the specified many-body DPD interactions. For the implemented many-body DPD model for vapour-liquid coexistence [142], the standard Groot-Warren part of the potential is calculated in *conservativeforce()*, while this subroutine calculates the density-dependent term for each particle and all pairs of particle species (except when both are frozen).

9.8 Intramolecular bond interactions

Forces and potentials due to bond stretching, angles and dihedrals are calculated in subroutines contained in *bond_module.F90*, specifically in the *bond_force()*, *angle_force()* and *dihedral_force()* subroutines for each bond, angle and dihedral respectively. Book-keeping arrays - *bndtbl*, *angtbl* and *dhdtbl* are constructed during simulation setup by DL_MESO_DPD to store the global indices of particles involved in bonds, angles and dihedrals as well as types with parameter sets identified from the *FIELD* file. To find each particle in the lists for bonds, angles and dihedrals, each subdomain puts together the array *lblclst* as a list of global and local indices for all particles involved in molecules: this list is sorted by global particle index by the subroutine *shellsort_list()* to enable the global index of a given particle to be found using a binary search in *search_list()*, which returns the index in *lblclst* to read off the local particle index or a negative number if the global particle index cannot be found.

The default approach for DL_MESO_DPD to deal with bonded interactions is for each processor core to only hold bond/angle/dihedral data for particles found in its subdomain. One of the particles in each bonded interaction - the first for bonds, the second for angles and dihedrals - is considered to 'hold' the interaction: whenever a particle moves to another processor core's subdomain during a deport communication, its associated entries in the bond/angle/dihedral tables are sent with the particle and added to the new subdomain's tables. The *contract_bndtbl()*, *contract_angtbl()* and *contract_dhdtbl()* subroutines are used to remove any old bonds, angles and dihedrals, marked during the deport step by setting the type to a negative value. The *bondforceslocal()* subroutine goes through each list in turn, finding the particles involved in each bond/angle/dihedral, calculating the vectors between the required particle pairs (using *images()* to obtain the minimum possible distance if the particles lie on opposite sides of a periodic boundary) and then calculating and assigning the required forces, potential, virial and stress tensors. Being able to complete these calculations is dependent on all the relevant particles' positions being known by the given processor core: as such, the boundary halo size can be adjusted by the user to ensure all particle pairs (in bonds and for vectors used for angles and dihedrals) can be found. If any pairs of particles cannot have their vector calculated due to one not being found in a subdomain and/or boundary halo, or the pair is too long for particular bond interactions - e.g. greater than the maximum specified distance for a finitely-extensible non-linear elastic (FENE) bond - DL_MESO_DPD will report the particle pairs and close down the simulation with an error message. The bonded interaction forces assigned to particles in boundary halos need to be sent back to the original processor cores holding those particles: this is carried out by the *importdata()* communication step along with other interaction forces.

An option to ensure DL_MESO_DPD can calculate all bonded interactions is invoked using the `global bonds` directive in the *CONTROL* file. This uses a replicated data approach: all processor cores hold the entire bond, angle and dihedral tables, which *bondforcesglobal()* goes through. The positions of all particles involved in molecules are shared among all processor cores and are used to calculate the particle vectors (using *images()* to obtain the minimum possible distance) before calculating the forces and potentials. The forces are assigned to all particles that exist in the current subdomain excluding any such particles in the boundary halo (achieved by *not* including boundary halo particles in the searchable list of global/local particle indices): if the 'holding' particle exists in that subdomain, the potential energy, virial and stress tensor are also assigned. While this approach requires more memory per processor core and does not scale well with larger numbers of cores, it does guarantee that the bonded interactions can be calculated regardless of the boundary halo size: it is mainly intended for simulation equilibration, particularly if the molecules are not energy-minimised in the initial configuration.

OpenMP can be used to speed up bonded interaction calculations using either approach by dividing the bond/angle/dihedral tables among the avaiilable threads. As for pairwise force calculations, either temporary arrays to store force contributions from each thread or a critical region to assign forces one thread at a time are required to ensure the forces are accumulated correctly.

9.9 Surface interactions

The subroutines in *surface_module.F90* deal with calculations of wall potentials, reflections of particles by hard surfaces, settings for Lees-Edwards shearing boundaries and setup of frozen particle walls.

The *surfacenodes()* subroutine is used to identify which processor cores hold surfaces (Lees-Edwards boundaries or hard walls), setting switches (*srflgc*) to indicate that no particle data at the nearest edges for the subdomain are communicated in particular directions, especially during the export step. In the case of Lees-Edwards boundaries, the normal communications in those directions are substituted with alternatives that can move particles tangentially as well as orthogonally to the required axis. If hard surfaces are required, their locations are also determined in local subdomain cooordinates: whiie they lie at the planar surfaces on the outside of the simulation box by default, they can be shifted further inwards to accomodate frozen bead walls. This subroutine is only called once for constant volume (NVT) simulations, but needs to be called again whenever the system volume changes due to the use of a barostat.

The wall potentials and forces are calculated for all particles within a cutoff distance *srfzcut* from the hard surfaces in the *wallforces()* subroutine, using the functional forms specified in *surfaceforce()*. As for pairwise forces, the magnitudes of wall forces are divided by the distances from the surface, which are then multiplied by orthogonal vectors from the wall to the particles to obtain the correct directions (e.g. repulsions away from the wall).

The *hardreflect_specular()* and *hardreflect_bounceback()* subroutines apply reflections to particles that would otherwise pass through hard surfaces: these are called during the first Velocity Verlet stage of force integration after the particle positions have been updated. Specular reflection only reverses the particle directions orthogonal to the surface, maintaining tangential motion and providing a free-slip boundary, while bounce back reflection reverses particle motion in all directions and provides a no-slip boundary. These reflections can be applied even if no wall potentials are specified, which can be used to prevent particles from penetrating frozen particle walls.

The *frozenbead()* subroutine calculates the numbers of particles and their spacings required for walls made up of frozen particles, which are added to the simulation box during initialisation. These are determined based on the specified particle density and wall thickness (given in the *FIELD* file), assuming the particles form face-centred cubic lattices on both sides of the box.

The *shearslide()* subroutine calculates the required tangential shift for particles passing through Lees-Edwards shearing boundaries. The resulting vector is only non-zero from the first timestep specified for shearing to start and depends on the time after that timestep and the specified wall velocity.

9.10 Electrostatic interactions

Electrostatic interactions in DL_MESO_DPD are currently supplied by Ewald summation, using either a standard Ewald sum or Smooth Particle Mesh Ewald (SPME), and are calculated shortly after the main force calculations. Both forms of Ewald sum require pairwise force and potential calculations in real-space for particle pairs up to *relec* apart, which are carried out by one of the following subroutines:

- *ewald_real_point()* for point charges
- *ewald_real_linear()* for linearly-smeared charges
- *ewald_real_slater_exact()* when using the exact form of Slater charge smearing
- *ewald_real_slater_approx()* when using an approximate form of Slater charge smearing
- *ewald_real_gauss()* when using Gaussian charge smearing
- *ewald_real_sinusoidal()* when using sinusoidal charge smearing

All of these use linked-cell lists with larger cell sizes to find charged particle pairs within the real-space cutoff distance: only a single charge-smearing scheme and a single set of corresponding parameters can be used for a given simulation, both of which are specified in the *CONTROL* file. (The pairwise calculations here can be sped up using OpenMP to divide the link cells among the available threads.)

Standard Ewald sums calculate the reciprocal-space part analytically in the *ewald_reciprocal()* subroutine, which is used regardless of charge smearing scheme (as this only affects the real-space part). The list of reciprocal-space vectors is initially constructed in the *ewald_reciprocal_map()* subroutine, which takes any vacuum gaps and Lees-Edwards shearing boundaries into account: this subroutine only needs to be called once for constant volume (NVT) simulations without Lees-Edwards shearing boundaries, but otherwise has to be called every timestep after the first Velocity Verlet stage. To obtain the full sums of products for particle charges and the required complex exponentials, the first loop through the reciprocal vectors is followed by a global summation among all processor cores: this is required to correctly calculate the potential and the forces acting on the particles. The calculations of potentials and forces can be sped up using OpenMP by dividing the available reciprocal-space vectors for potential calculations and the particles for force calculations among the available threads: the latter can be carried out safely without other threads attempting to assign forces to each particle.

Smooth Particle Mesh Ewald (SPME) calculates the reciprocal-space part of the Ewald sum by assigning charges to a grid using B-spline interpolation, finding the Fourier transform of the charge grid using Fast Fourier Transforms (FFTs) and using the results to calculate the potential and forces on charged particles, the latter making use of gradients for the B-splines originally used to construct the charge grid. The *spme_ewald_reciprocal()* subroutine constructs the charge grid, setting up the B-splines for all charged particles in the *spme_bspline_gen()* subroutine with the spline interpolation order *mxspl*, using the reciprocal-space vector list prepared in *spme_reciprocal_map()* and global summing the grid if running in parallel. The subsequent inverse and forward FFTs are implemented by default using *fft3d()*, the FFT solver supplied with DL_MESO_DPD, but compile-time options are available to substitute in either the 3D FFT solver from the IBM Engineering and Scientific Subroutine Library (ESSL) or the Fastest Fourier Transform in the West (FFTW)[8]. Since all cores have access to the full charge grid, they all carry out the Fourier transforms autonomously and use the results to calculate the potential, virials, stress tensors and particle forces: the latter normally do not quite sum to zero, so corrections to these forces are calculated and applied to all charged particles. OpenMP can be used to speed up SPME calculations by dividing the particles among available threads for force calculations and, if available, by applying it to the FFT solver (e.g. FFTW) when calculating Fourier transforms of the charge grid.

A vacuum gap can be specified in the *CONTROL* file to extend the effective box volume in reciprocal-space calculations for simulations with non-periodic boundary conditions (e.g. with hard surfaces or frozen particle walls). Corrections are made to forces and potentials during reciprocal-space calculations due to charge dipoles if hard surfaces are included or a vacuum gap is specified (which can be sped up by dividing the particles among available OpenMP threads), as well as additional corrections to remove reciprocal space contributions for pairs of frozen charged particles, calculated in the *ewald_frozen()* subroutine either once for constant volume calculations without shear or during every timestep for other simulations. Standard corrections to potentials for self-energy terms and for net-charged systems are also applied during reciprocal-space calculations.

9.11 Force integration and barostats

Integration of forces in DL_MESO_DPD is generally carried out with the two-stage Velocity Verlet (VV) algorithm, using one of the following modules per simulation:

- *integrate_dpd_mdvv.F90* to integrate interaction and DPD forces using the standard (MD) VV scheme
- *integrate_dpd_dpdvv.F90* to integrate interaction and DPD forces using VV but recalculate DPD dissipative forces, known as DPD Velocity Verlet
- *integrate_dpd_shardlow.F90* to integrate interaction forces using VV and DPD forces using Shardlow splitting
- *integrate_lowe.F90* to integrate interaction forces using VV and apply the Lowe-Andersen thermostat to at least some particle pairs

[8] The FFTW implementation requires the charge array to be allocated as a C double complex type and for ‘plans’ to be setup to carry out the transforms: these are prepared in the *spme_initialize()* subroutine.

- *integrate_peters.F90* to integrate interaction forces using VV and apply the Peters thermostat to all particle pairs within the thermostat cutoff distance
- *integrate_stoyanov.F90* to integrate interaction and pairwise thermostat forces using VV and apply the Lowe-Andersen thermostat to some particle pairs, known as Stoyanov-Groot

The subroutines in these modules take the stage number (1 or 2) as inputs and applies the appropriate stage:

- Stage 1 advances the particle velocities from time t to $t + \frac{1}{2}\Delta t$ and the particle positions from time t to $t + \Delta t$
- Stage 2 advances the particle velocities from time $t + \frac{1}{2}\Delta t$ to $t + \Delta t$

After advancing the particle positions in stage 1, the subroutines then apply any boundary conditions (e.g. reflections) and, if Lees-Edwards shearing and/or a barostat is in use, Ewald reciprocal-space vector maps for electrostatic interactions are recalculated. If a barostat changes the box volume, the *domain_dimensions()* subroutine is recalled to calculate new subdomain volume sizes and reallocate arrays for linked-cell lists if the numbers of link cells have changed. Once the velocities have advanced during stage 2, the kinetic part of the stress tensor is calculated. During both stages, any external forces - gravity and electric fields - are applied at the same time as the particle forces to advance the velocities.

No modifications to the standard VV algorithm are made for the MD-VV option as the DPD thermostat (dissipative and random) forces are included together with the conservative interaction forces. The DPD-VV option includes a call to *dragforces_dpdvv()* to recalculate the dissipative forces at the end of the second VV stage. The Shardlow splitting options include calls to *shardlow_integrate()* to integrate the DPD dissipative and random forces: first-order Shardlow calls this subroutine once at the beginning of the first VV stage, while second-order Shardlow also calls this subroutine again after the velocities are advanced during the second VV stage. (Both DPD-VV and Shardlow splitting options require updates to velocities for particles in the boundary halo before these subroutines are called, which is enabled with the *exportvelocitydata()* subroutine.)

The other non-DPD thermostats are applied after particle velocities have been advanced during the second VV stage (which uses conservative interaction forces only apart from Stoyanov-Groot, which adds the pairwise Nosé-Hoover thermostatting forces): these are implemented with thermostat-specific subroutines, e.g. *lowe_correct()*, which apply the corrections to particle velocities using the lists of particle pairs constructed during force calculations, e.g. by *forces_lowe()*. Each entry in the lists includes a random number generated for the particle pair: after the lists for all processor cores are put together, the combined list is sorted by these random numbers to give a consistent order for the pairs for all processor cores to work through sequentially and apply corrections if either or both particles in a pair belong to the given processor core. If the two particles in a pair belong to different cores, the two cores will share information required to complete the velocity correction calculation (particle masses and velocities) via a blocked MPI communication.

Two barostats are available to apply volume changes for pressure control - Langevin and Berendsen - each of which can be used for isotropic constant pressure (NPT), constant normal pressure and surface volume (NP$_n$AT), and constant normal pressure and surface tension (NP$_n\gamma$T) ensembles, with semi-isotropic and fully anisotropic options for the latter. Both barostat types make use of instantaneous virial values obtained during force calculations that do *not* include contributions from pairwise thermostats, stored for each Cartesian component in *ivrl* and used to calculate the required pressure (either system-wide or by individual components). In the case of constant surface volume (NP$_n$AT) ensembles, the box size is adjusted only in the z-direction, while the surface tension for NP$_n\gamma$T ensembles is applied in the same direction: this quantity is measured and reported in both the *OUTPUT* and *CORREL* files.

The Langevin barostats are implemented using an iterative procedure [64]. The first VV stage includes modifications to the mid-step particle velocities to incorporate the piston velocity at the start of the timestep and the new particle positions are modified using scaling factors that depend on the mid-step piston velocity. During the second VV stage, the changes to particle velocities due to interaction forces and the thermostat (without initially applying the barostat) are kept in memory, before storing the mid-step piston velocity and estimating the value at the end of the timestep by using the piston force at the start of the timestep and the velocity at the beginning of the previous timestep ($t - \Delta t$). This estimate for the end-step piston velocity is used to calculate new particle velocities (also taking the previously calculated changes into account), before using the velocities to calculate the pressure and the end-step piston force. A new estimate for the end-step piston velocity is then calculated before the cycle repeats from calculating new particle velocities: these and the piston force and velocity will be considered converged once the mean squared change in particle velocity drops below a minimum value, *langepsilon*.

The Berendsen barostats are implemented by applying changes to the particle positions during the first VV stage, consisting of scaling factors applied to the positions before adjustments are made based on mid-step particle velocities. The scaling factors are calculated during the second VV stage of the previous timestep, which are based on the instantaneous pressures calculated after the end-step velocities have been obtained.

9.12 System initialisation

A DL_MESO_DPD simulation requires particle positions, velocities and forces to be defined, along with any bond data, which is carried out by the *start()* subroutine. Starting with the molecule definitions in the *FIELD* file, an array *molstart* is put together with the starting global particle numbers for each molecule in the system (initially without applying any system duplication). This is then followed by obtaining the required particle positions and velocities using one of three options:

- Creating a new configuration based on the contents of the *CONTROL* and *FIELD* files
- Reading a *CONFIG* file to use an initial configuration specified by the user
- Reading an *export* file either to restart a previous simulation or as a new simulation

The *initialize()* subroutine determines the positions of three broad types of particles: those in frozen particle walls, those involved in molecules and those not involved in molecules (e.g. solvent). Both the frozen particle walls and the non-molecular particles are assigned to face-centred cubic lattices, whose spacings in each Cartesian dimension are determined from the available volume and the number of particles, with each particle given a unique global particle number (index). In the case of the non-molecular particles, the volume in which the lattice has to fit is given in the *CONTROL* file, although the lattice may be incomplete depending on the total number of particles required. The one particle species for the frozen particle walls is readily assigned to the particles, while those for non-molecular particles are assigned to distribute the available species (based on their total numbers) as evenly as possible across the simulation box and randomly among particles in each processor core's subdomain. The required molecules are inserted into the simulation box, using the mini-configuration supplied in the *FIELD* file for each molecule type and applying random translations and rotations (as well as inversions if permitted): if any frozen particle walls or hard surfaces are included, the particle positions of each inserted molecule are checked to ensure no bonds cross these boundaries and a new set of coordinates calculated if they do. A random number generator with the same state across all processor cores is used to ensure the molecules are correctly assigned and to avoid requiring communications among the cores: all cores work through all of the molecules in the system, with each core only assigning particles that are found inside its subdomain.

If a *CONFIG* file is supplied, the *read_config()* subroutine is used to read the contents of the file and assign particles to each processor core based on whether or not they are within its subdomain. If system duplication is requested in the *CONTROL* file, this is applied in this subroutine: care is taken to ensure the particles in molecules that cross between duplicated unit cells are numbered correctly for specifying bond data. If a frozen particle wall is specified in the *CONTROL* and *FIELD* files, this is also added to the system (after any duplication) with corresponding increases in system volume and total number of particles.

If a previous simulation is restarted or used as the basis for a new simulation, the *read_export()* subroutine is used to read the contents of the supplied *export* file and assign particles to each processor core based on whether or not they are within its subdomain. No system duplication or additional frozen particle walls can be applied in this subroutine, although positions, velocities and forces are assigned for all particles. If the restart option selected in the *CONTROL* file requests a temperature rescale, the particle velocities are adjusted by a scaling factor to give the required system temperature (compared with the actual value given in the *export* file).

Once at least the particle positions, species and molecule types have been assigned to all processor cores, the *molstart* array is adjusted for any system duplication. If the system is being initialised from scratch by DL_MESO_DPD or the *CONFIG* file does not include particle velocities, these are set randomly by the *initialvelocity()* subroutine to give the required system temperature. Any electrostatic interactions are set up by the *ewald_reciprocal_map()* or *spme_reciprocal_map()* subroutines, along with the *ewald_frozen()* subroutine to apply corrective forces, virials, stresses and potentials to remove interactions for pairs of charged frozen particles. A sample of the initial configuration on processor core 0 is then printed to the *OUTPUT* file (or standard output) before the *write_history_header()* subroutine is called to prepare a new *HISTORY* file (if requested). The *plcfor_initial()* subroutine is called later to calculate any missing particle forces as well as the potential energy, stress tensor etc. before the main simulation starts.

Two subroutines are included in *start_module.F90* to assist with system initialisation. The *sort_beads()* subroutine re-orders the particles in each processor core to place the frozen particles at the beginning of the local arrays for positions, velocities and forces. The *assign_bonds()* subroutine uses the *molstart* array to put together the required bond, angle and dihedral book-keeping tables: if the `global bonds` approach is used, all processor cores are assigned all bonds/angles/dihedrals in the system, while the default (local) approach only assigns bonds/angles/dihedrals for a given processor core if the 'owning' particle exists in its subdomain.

9.13 Reading input files

Up to five input files can be read by DL_MESO_DPD at the start of a simulation: *CONTROL* with simulation parameters, *FIELD* with particle and interaction information, *CONFIG* with initial conditions, *export* with a simulation snapshot for restarting a previous simulation, and *REVIVE* with statistical accumulator and random number generator states for simulation restart. The first three of these are (human-readable) text files, while *export* and *REVIVE* are written in binary (see below).

The *CONTROL* and *FIELD* files are read by a single processor core (numbered 0), which then broadcasts the results to all other cores. Each of these files is read at least twice: an initial scan of each file is carried out - *scan_control()* and *scan_field()* - to find the most important simulation parameters and gauge how large arrays for particle information, interaction parameters, bond connectivity etc. need to be. (If any vital information is missing, DL_MESO_DPD will stop with an error message.) Once all arrays have been allocated in memory, each of the files is then read in full - *read_control()* and *read_field()* - to enter the values into memory ready for initialising and running the simulation.

A different approach is taken with the *CONFIG* file, which is often substantially larger than other input files. After an initial scan for information about the simulation box size and the amount of data available per particle - *scan_config()* - carried out by processor core 0, the main reading subroutine - *read_config()* - divides the file as evenly as possible (based on the size in bytes) among the available processor cores. Each core reads in data for approximately equal numbers of particles and assigns the data to arrays in derived data types, separating particles out based on whether or not each one belongs to a molecule. If the `nfold` option in *CONTROL* is invoked, the particles are duplicated with their positions adjusted to fit inside the expanded simulation box. (The position adjustments for molecules are carried out to ensure the correct particles are joined together by bonds.) MPI communications are then carried out by all processor cores to send their particle data to the appropriate cores based on the particles' positions.

Reading of binary *export* files is carried out using a similar basic approach to *CONFIG* files. After an initial scan for simulation box size by processor core 0 - *scan_export()* - the reading of the file in *read_export()* starts with the same core reading and broadcasting the shift in positions due to Lees-Edwards shearing boundaries as well as a velocity scaling factor (as required for the `restart scale` option in *CONTROL*) and the total number of particles in the file. The subroutine then divides up the particles among the available processor cores, with each core reading its section of the block of integers with global particle numbers, species and molecule type numbers, followed by the particle data itself. This information is then stored in arrays with a similar derived data type to that used in *read_config()*, before the particle data is then communicated to the appropriate cores based on positions.

The *REVIVE* file is only read by *read_revive()* if the `restart` option is included in the *CONTROL* file, i.e. resuming a previous simulation instead of starting a new one with a previous configuration. If the file is read by DL_MESO_DPD, processor core 0 reads in nearly all of the simulation state data in the file before broadcasting it to all other cores. The exception to this read/broadcast model are the available random number generator states, which are each individually read by a processor core and overwrite those generated when DL_MESO_DPD starts up.

9.14 Writing output files

The *OUTPUT*, *CORREL* and *Stress_*.d* files are ANSI text files all written by a single processor core (number 0). These consist of information available among all processor cores, including values broadcast by processor core 0 to others and properties found by global summation among all cores. Other output files that rely on specific information from each core - primarily *HISTORY* and *export* - are written in binary by a set of processor cores designated to do so autonomously from other cores and simultaneously using MPI-IO (or stream I/O in serial) with the endianness of the computer used to run the simulation.

The *OUTPUT* file includes information read in from input files and derived from simulation setup (printed by the subroutines *sysdef()* and *start()* respectively), system-wide information about the simulation at periodic intervals - written by *write_output_summary()* - and a summary of the calculation at the end with timings, average properties and fluctuations written by the subroutine *write_output_result()*. This file also shows any error or warning messages printed by *error()* to help users diagnose any issues with the simulation. All of this information can be redirected to the standard output (typically the screen for standalone computers) if the `l_scr` option in the *CONTROL* file is invoked.

Statistical properties for the system - energies, pressure, temperature etc. - are written to the *CORREL* file by the *write_correl()* subroutine as tabulated columns showing their variation as a function of time. This file is optional and the frequency of writing is specified in the *CONTROL* file: the file is only written to after equilibration has come to an end. Since the number of properties can vary from simulation to simulation based on the specified interactions and ensemble, the number of columns of data in a *CORREL* file can vary: the first line in this file consists of column headers to identify each property. Some data not printed in the *OUTPUT* file are included here, e.g. time progression of pressure tensors, bond lengths.

If requested in the *CONTROL* file, the subroutine *write_stress()* will write pressure tensors to *Stress_*.d* files, separated into interaction potential (Stress_pot.d), dissipative (Stress_diss.d), random (Stress_rn.d) and kinetic (Stress_kin.d) components. These files follow a similar format to *CORREL* files: tabulated columns for time, the nine tensor components and system volume (required to calculate stress tensors) with a header in the first line to identify each column. The user can specify which files are created as well as the frequency and the first timestep for writing to these files in the *CONTROL* file.

Writing to *export* and *HISTORY* files is reliant upon the creation of groups of processor cores. Each group gathers together particle data for one processor core to write to a file using MPI-IO to enable multiple simultaneous write operations. The *init_output_groups()* subroutine determines how many gathering groups of processor cores are required based on the total number of particles and a maximum amount of data to be shared among each group (set to 32 MiB) - albeit reset to a minimum of 4 or the total number of processor cores (whichever is smaller) - before working out how many processor cores are needed per group and assigns the available cores to each group. The first numbered core in each group is designated as the root to receive gathered particle data and write to files, and MPI communicators are created to enable each group to gather together their data autonomously to the others. The root cores for these groups are themselves put together as another group for writing to files and a set of MPI communicators are created to enable these cores to write concurrently to each file using MPI-IO. (If running in serial, a single data gathering group and a single file writing group are created with the single processor core in both: no MPI communicators are actually created in this case.) The information about the gathering and writing groups and their associated MPI communicators are stored in *group_info*, a derived data type in *write_module.F90* used by the various file-writing routines.

The subroutine *gather_write_data()* is used to gather together particle data (positions, velocities, forces, global identifying numbers etc.) among each data gathering group of processor cores. Each gathering group determines the total number of particles they hold to determine the sizes of arrays that the root core needs to allocate. Regardless of which file is being written to, all of the particle data is then gathered onto the root core prior to *write_history()* and/or *write_export()* being called to write the data to the *HISTORY* and/or *export* files respectively (depending on which are required at the current timestep). These subroutines use the root cores for each gathering group to write the data to the files concurrently with MPI-IO (or with stream I/O when running in serial): all of the cores involved in writing to a file share the numbers of particles they each hold among themselves, which are used to determine the byte number in the file at which each core starts to write. The data is written to the files as continuous stream of values - integers for particle numbers, double-precision real numbers for positions, velocities and forces - and no attempt is made to sort the particles numerically beforehand in order to save time in writing. A small amount of system-wide data - e.g. box dimensions, timestep number - are also written to the file(s) by processor core 0, which happens to be the root core for one of the gathering groups. In the case of the *HISTORY* file, the total file size, number of trajectory frames and timestep number for the last frame in the header - initially written by the *write_history_header()* subroutine - are updated as additional trajectory frames are appended to the end of the file, while the *export* file is overwritten each time.

The *write_config()* subroutine follows a slightly different strategy when creating a CFGINI file in the text-based *CONFIG* format. After processor core 0 opens the file and writes the short header, each particle in the simulation box is assigned to one of the file writing processor cores based on its global identifying number before its data (particle number, position, veloocity and force) are sent to that core. Each file writing core then sorts its received particle data based on particle number, before rendering the data as a string to write concurrently to the file. While this gather/write process typically takes longer than that for *HISTORY* and *export* files, it is only intended to be carried out once per simulation.

The *write_revive()* subroutine predominately uses processor core 0 to write simulation data at the current timestep intended for a later restart to the *REVIVE* file, including the current timestep number and statistical stacks for calculating rolling and final average values of system properties. The exceptions to this are the random number generator states: each processor core writes its own generator state to the *REVIVE* file using MPI-IO, with the starting position in the file determined using the core number.

CHAPTER

TEN

DL_MESO_DPD CODE DESCRIPTION

This chapter lists and describes the subroutines, functions, variables, datatypes etc. in DL_MESO_DPD, based on output generated using Doxygen with annotations in the code.

10.1 dlmesodpd.F90

10.1.1 Summary

Main program for DL_MESO_DPD: version with OpenMP multithreading available as dlmesodpd_omp.F90.

10.1.2 Function/Subroutine Documentation

dlmesodpd()

```
program dlmesodpd
```

Main DL_MESO_DPD program: starting off DPD calculation, reading input files, printing first messages (including version/revision numbers, how many processors etc.), setting up initial (restarted) configuration, launching calculation, printing final results and messages (including prompt to user to cite DL_MESO article and/or website) and freeing up memory afterwards. (Version with OpenMP multithreading also adds initial message indicating how many threads are in use.)

10.2 constants.F90

10.2.1 Summary

Module with commonly-used constants and parameters.

10.2.2 Variables

- `integer, parameter` *dp*

  Real data kind for double precision.

- `integer, parameter` *si*

  Integer kind for standard integers.

- `integer, parameter` *li*

  Integer kind for long integers.

- `integer, parameter` *mxword*

  Longest string (word) used for parsing numbers.
- `integer, parameter` *nread*

  File input channel for reading input files.
- `integer, parameter` *nprint*

  File output channel for writing OUTPUT file.
- `integer, parameter` *nrtin*

  File input channel for reading restart data (export, REVIVE)
- `integer, parameter` *nrtout*

  File output channel for writing restart data (export, REVIVE)
- `integer, parameter` *nsave*

  File output channel for writing statistical properties (CORREL)
- `integer, parameter` *nhist*

  File output channel for writing trajectory data (HISTORY)
- `integer, parameter` *nstress*

  File output channel for writing stress tensors (Stress_*.d)
- `integer, parameter` *csize*

  Number of entries (dimensions) per particle property.
- `integer, parameter` *statsize*

  Number of statistical properties for averaging.
- `integer, parameter` *stksize*

  Number of instantaneous properties for statistical stacks.
- `real(kind=dp), parameter` *pi*

  Value of pi.
- `real(kind=dp), parameter` *rtpi*

  Value of square root of pi.
- `real(kind=dp), parameter` *sin60*

  Value of sine of 60 degrees (used for FFT solvers)
- `real(kind=dp), parameter` *cos72*

  Value of cosine of 72 degrees (used for FFT solvers)
- `real(kind=dp), parameter` *sin72*

  Value of sine of 72 degrees (used for FFT solvers)
- `real(kind=dp), parameter` *degrad*

  Parameter to convert degrees to radians.
- `real(kind=dp), parameter` *raddeg*

  Parameter to convert radians to degrees.
- `real(kind=dp), parameter` *fkt*

  Parameter to convert kinetic energy to temperature.

- `real(kind=dp), parameter` *rt12*

  Square root of 12 used for approximation of Gaussian random numbers.
- `real(kind=dp), parameter` *langepsilon*

  Convergence limit for iterative Langevin barostat.
- `character(len=3)` *version*

  Major current version number of DL_MESO.
- `integer` *revision*

  Revision number of DL_MESO.
- `character(len=14)` *date*

  Date (month and year) for current DL_MESO version.
- `character(len=4)` *year*

  Year for current DL_MESO version.

10.2.3 Variable Documentation

cos72

```
real(kind=dp), parameter constants::cos72 = 0.30901699437494742_dp
```

Value of the cosine of 72 degrees, $\cos\left(\frac{2\pi}{5}\right) = \frac{\sqrt{5}-1}{4}$, used in Fast Fourier Transform (FFT) solver.

csize

```
integer, parameter constants::csize = 4
```

Number of entries (dimensions) for each main particle property (positions, velocities, forces), value set to exploit cache when calling property from memory.

date

```
character(len=14) constants::date
```

String containing date (month and year) of current version of DL_MESO, used in printing messages in OUTPUT file (or to standard output/screen).

degrad

```
real(kind=dp), parameter constants::degrad = pi/180.0_dp
```

Factor to convert angles in degrees to radians, $\frac{\pi}{180}$.

dp

```
integer, parameter constants::dp = SELECTED_REAL_KIND (15, 307)
```

Fortran kind for a real data type that provides double precision (8 bytes in size).

fkt

```
real(kind=dp), parameter constants::fkt = 2.0_dp/3.0_dp
```

Factor to convert kinetic energy per particle to product of temperature and Boltzmann constant, $\frac{2}{3}$.

langepsilon

```
real(kind=dp), parameter constants::langepsilon = 1.0e-6_dp
```

Maximum squared change in particle speed between iterations of Langevin barostat required to indicate convergence.

li

```
integer, parameter constants::li = SELECTED_INT_KIND (12)
```

Fortran kind for an integer data type that provides long precision (8 bytes in size).

mxword

```
integer, parameter constants::mxword = 20
```

Largest number of characters assumed for parsing a string (word) to find any numbers in a text-based input file (CONTROL, FIELD, CONFIG).

nhist

```
integer, parameter constants::nhist = 10
```

Fortran data channel number used to write binary output file for trajectory data (HISTORY).

nprint

```
integer, parameter constants::nprint = 6
```

Fortran data channel number used to write simulation information to OUTPUT file (or to standard output/screen).

nread

```
integer, parameter constants::nread = 5
```

Fortran data channel number used to read in text-based input files (CONTROL, FIELD, CONFIG).

nrtin

```
integer, parameter constants::nrtin = 7
```

Fortran data channel number used to read in binary input files for simulation restart (export, REVIVE).

nrtout

```
integer, parameter constants::nrtout = 8
```

Fortran data channel number used to write binary output files for simulation restart (export, REVIVE).

nsave

```
integer, parameter constants::nsave = 9
```

Fortran data channel number used to write text output file for statistical properties (CORREL).

nstress

```
integer, parameter constants::nstress = 11
```

Fortran data channel number used to write text output files for separated stress tensors (Stress_*.d).

pi

```
real(kind=dp), parameter constants::pi = 3.14159265358979323_dp
```

Fundamental constant π.

raddeg

```
real(kind=dp), parameter constants::raddeg = 180.0_dp/pi
```

Factor to convert angles in radians to degrees, $\frac{180}{\pi}$.

revision

```
integer constants::revision
```

Integer with minor revision number for current version of DL_MESO, used in printing messages in OUTPUT file (or to standard output/screen).

rt12

```
real(kind=dp), parameter constants::rt12 = 3.464101615377546_dp
```

Standard deviation of uniform random number generator, $\sqrt{12}$, used to give approximations of Gaussian random numbers required for DPD random forces.

rtpi

```
real(kind=dp), parameter constants::rtpi = 1.772453850905516O3_dp
```

Square root of fundamental constant $\sqrt{\pi}$.

si

```
integer, parameter constants::si = SELECTED_INT_KIND (8)
```

Fortran kind for an integer data type that provides standard precision (4 bytes in size).

sin60

```
real(kind=dp), parameter constants::sin60 = 0.86602540378443865_dp
```

Value of the sine of 60 degrees, $\sin\left(\frac{\pi}{3}\right) = \frac{\sqrt{3}}{2}$, used in Fast Fourier Transform (FFT) solver.

sin72

```
real(kind=dp), parameter constants::sin72 = 0.95105651629515357_dp
```

Value of the sine of 72 degrees, $\sin\left(\frac{2\pi}{5}\right) = \frac{\sqrt{10+2\sqrt{5}}}{4}$, used in Fast Fourier Transform (FFT) solver.

statsize

```
integer, parameter constants::statsize = 51
```

Total number of statistical properties (energies per particle, system pressure, temperature etc.) collected for time-averaged values and fluctuations during simulation (after equilibration).

stksize

```
integer, parameter constants::stksize = 15
```

Total number of statistical properties (energies per particle, system pressure, temperature etc.) collected for rolling average values.

version

```
character(len=3) constants::version = '2.7'
```

String containing major version number for current version of DL_MESO, used in printing messages in OUTPUT file (or to standard output/screen).

year

```
character(len=4) constants::year
```

String containing year for current version of DL_MESO, used in printing messages in OUTPUT file (or to standard output/screen).

10.3 variables.F90

10.3.1 Summary

Module with globally accessible variables for DPD simulation.

10.3.2 Variables

- `integer` *idnode*

  Number (rank) of current processor.

- `integer` *nodes*

  Total number of processors.

- `integer` *tnum*

  Total number of OpenMP threads.

- `integer` *tnumuse*

  Total number of OpenMP threads in use.

- `character(len=12)` *exportname*

  Name of file for restart simulation configuration.

- `character(len=12)` *historyname*

  Name of file for simulation trajectories (default: HISTORY)

- `logical` *ltemp*

  Equilibration temperature rescale switch.

- `logical` *lconfzero*

  CONFIG file origin switch.

- `logical` *lcorr*

  CORREL file writing switch.
- `logical` *ltraj*

  Trajectory writing switch.
- `logical` *lstrs*

  Separated stress tensor writing switch.
- `logical, dimension(4)` *lstrss*

  Separated stress tensor file switches.
- `logical` *lbond*

  Bond switch.
- `logical` *langle*

  Angle switch.
- `logical` *ldihed*

  Dihedral switch.
- `logical` *lgbnd*

  Global bonds switch.
- `logical` *lvarfc*

  Variable forces switch.
- `logical` *lisoprs*

  Isotropic/semi-isotropic switch.
- `logical` *ligindex*

  CONFIG ignore indices switch.
- `logical` *lmb*

  Many-body DPD switch.
- `logical` *ldpol*

  Charge dipole switch.
- `logical` *lnfold*

  System duplication switch.
- `logical` *ldyn*

  Dynamic system switch.
- `logical` *lompcrit*

  OpenMP critical regions switch.
- `logical` *l_scr*

  Divert output switch.
- `integer` *nfoldx*

  Number of system duplications in x-direction.
- `integer` *nfoldy*

  Number of system duplications in y-direction.

- `integer` *nfoldz*

  Number of system duplications in z-direction.

- `integer` *nfold*

  Total number of system duplications.

- `integer` *levcfg*

  CONFIG file data level.

- `integer` *imcon*

  CONFIG file boundary condition key.

- `integer` *outsel*

  OUTPUT file printing key.

- `integer` *keytrj*

  HISTORY file data level.

- `integer` *engunit*

  Energy unit selection.

- `integer` *rndseed*

  Random number seed.

- `integer` *maxdim*

  Maximum number of particles per processor.

- `integer` *maxpair*

  Maximum number of particle pairs per processor.

- `integer` *maxbfbd*

  Maximum number of particles in boundary halo for inter-processor communications.

- `integer` *maxbuf*

  Maximum size of arrays for inter-processor communications.

- `integer` *mxmolsize*

  Maximum number of particles per molecule.

- `integer` *mxbonds*

  Maximum number of bonds per molecule.

- `integer` *mxangles*

  Maximum number of angles per molecule.

- `integer` *mxdiheds*

  Maximum number of dihedrals per molecule.

- `integer` *mxprm*

  Maximum number of parameters per interaction.

- `integer` *mxsprm*

  Maximum number of parameters per surface interaction.

- `real(kind=dp)` *dvar*

  Density variation multiplier.

- `character(len=80)` *text*

  CONTROL file header to name DPD simulation.
- `integer` *nrun*

  Total number of DPD simulation timesteps.
- `integer` *nsbpo*

  OUTPUT file printing frequency.
- `integer` *iscorr*

  CORREL file writing frequency.
- `integer` *ntraj*

  Trajectory file writing frequency.
- `integer` *straj*

  Trajectory file writing start.
- `integer` *ndump*

  Restart file writing frequency.
- `integer` *nstrs*

  Stress tensor file writing frequency.
- `integer` *sstrs*

  Stress tensor file writing start.
- `integer` *nsbts*

  Equilibration temperature rescale frequency.
- `integer` *nseql*

  Equilibration timesteps.
- `integer` *kres*

  Simulation restart key.
- `integer` *nspe*

  Number of particle species.
- `integer` *npot*

  Number of interaction potentials between pairs of species.
- `integer` *nmoldef*

  Number of defined molecule types.
- `integer` *nbonddef*

  Number of defined bond types (based on functional form and parameters)
- `integer` *nangdef*

  Number of defined angle types (based on functional form and parameters)
- `integer` *ndhddef*

  Number of defined dihedral types (based on functional form and parameters)
- `integer` *nstk*

  Statistical rolling average stack size.

- `integer` *nsyst*

  Total number of particles in simulation.

- `integer` *nusyst*

  Total number of particles outside of molecules in simulation.

- `integer` *nfsyst*

  Total number of frozen particles in simulation.

- `integer` *nfwsyst*

  Total number of particles in frozen bead walls.

- `integer` *nsystcell*

  Number of particles in unit cell.

- `integer` *nusystcell*

  Number of particles outside of molecules in unit cell.

- `integer` *nfsystcell*

  Number of frozen particles in unit cell.

- `integer` *nummol*

  Total number of molecules in simulation.

- `integer` *nummolcell*

  Number of molecules in unit cell.

- `integer` *numbond*

  Total number of bonds in simulation.

- `integer` *numang*

  Total number of angles in simulation.

- `integer` *numdhd*

  Total number of dihedrals in simulation.

- `integer` *numbondcell*

  Number of bonds in unit cell.

- `integer` *numangcell*

  Number of angles in unit cell.

- `integer` *numdhdcell*

  Number of dihedrals in unit cell.

- `integer` *nstep*

  Current simulation timestep number.

- `real(kind=dp)` *timfrc*

  Particle force calculation timer.

- `real(kind=dp)` *timstp*

  Timestep calculation timer.

- `real(kind=dp)` *temp*

  System temperature.

- `real(kind=dp)` *tstep*

  Timestep size.
- `real(kind=dp)` *rtstep*

  Reciprocal of timestep size.
- `real(kind=dp)` *rhalo*

  Specified size of boundary halo.
- `real(kind=dp)` *rhalox1*

  Boundary halo size in -x direction.
- `real(kind=dp)` *rhalox2*

  Boundary halo size in +x direction.
- `real(kind=dp)` *rhaloy1*

  Boundary halo size in -y direction.
- `real(kind=dp)` *rhaloy2*

  Boundary halo size in +y direction.
- `real(kind=dp)` *rhaloz1*

  Boundary halo size in -z direction.
- `real(kind=dp)` *rhaloz2*

  Boundary halo size in +z direction.
- `real(kind=dp)` *rcut*

  Maximum interaction cutoff distance.
- `real(kind=dp)` *rct2*

  Square of maximum interaction cutoff distance.
- `real(kind=dp)` *rrct2*

  Squared reciprocal of maximum interaction cutoff distance.
- `real(kind=dp)` *rtcut*

  Thermostat cutoff distance.
- `real(kind=dp)` *rrtcut*

  Reciprocal of thermostat cutoff distance.
- `real(kind=dp)` *rtct2*

  Square of thermostat cutoff distance.
- `real(kind=dp)` *rmbcut*

  Many-body DPD interaction cutoff distance.
- `real(kind=dp)` *rmbct2*

  Square of many-body DPD interaction cutoff distance.
- `real(kind=dp)` *rrmbcut*

  Reciprocal of many-body DPD interaction cutoff distance.
- `real(kind=dp)` *relec*

  Short-range (real-space) electrostatic interaction cutoff distance.

- `real(kind=dp)` *rel2*

  Square of short-range electrostatic interaction cutoff distance.

- `real(kind=dp)` *srfzcut*

  Surface interaction cutoff distance.

- `real(kind=dp)` *srfzct2*

  Square of surface interaction cutoff distance.

- `real(kind=dp)` *timjob*

  Maximum available time for DPD calculation (in seconds)

- `real(kind=dp)` *tclose*

  Calculation closedown time (in seconds)

- `real(kind=dp)` *volm*

  Total system volume.

- `real(kind=dp)` *dimx*

  Simulation box size in x-direction.

- `real(kind=dp)` *dimy*

  Simulation box size in y-direction.

- `real(kind=dp)` *dimz*

  Simulation box size in z-direction.

- `real(kind=dp)` *dimxcell*

  Unit cell box size in x-direction.

- `real(kind=dp)` *dimycell*

  Unit cell box size in y-direction given by CONFIG file.

- `real(kind=dp)` *dimzcell*

  Unit cell box size in z-direction given by CONFIG file.

- `integer` *npx*

  Total number of processors and subdomains in x-direction.

- `integer` *npy*

  Total number of processors and subdomains in y-direction.

- `integer` *npz*

  Total number of processors and subdomains in z-direction.

- `integer, dimension(6)` *map*

  Mapping of neighbouring processors for current processor (1 = -x, 2 = +x, 3 = -y, 4 = +y, 5 = -z, 6 = +z)

- `integer` *idx*

  Number of processor/subdomain in x-direction (between 0 and npx-1)

- `integer` *idy*

  Number of processor/subdomain in y-direction (between 0 and npy-1)

- `integer` *idz*

  Number of processor/subdomain in z-direction (between 0 and npz-1)

- `real(kind=dp)` *delx*

  Position of bottom left back corner of subdomain (x-dimension)
- `real(kind=dp)` *dely*

  Position of bottom left back corner of subdomain (y-dimension)
- `real(kind=dp)` *delz*

  Position of bottom left back corner of subdomain (z-dimension)
- `real(kind=dp)` *sidex*

  Size of subdomain in x-direction.
- `real(kind=dp)` *sidey*

  Size of subdomain in y-direction.
- `real(kind=dp)` *sidez*

  Size of subdomain in z-direction.
- `integer` *nbeads*

  Number of particles in current processor/subdomain.
- `integer` *nfbeads*

  Number of frozen particles in current processor/subdomain.
- `integer` *nlx*

  Number of interaction link cells in current subdomain (x-direction)
- `integer` *nly*

  Number of interaction link cells in current subdomain (y-direction)
- `integer` *nlz*

  Number of interaction link cells in current subdomain (z-direction)
- `integer` *nlx2*

  Number of interaction link cells in current subdomain and boundary halo (x-direction)
- `integer` *nly2*

  Number of interaction link cells in current subdomain and boundary halo (y-direction)
- `integer` *nlz2*

  Number of interaction link cells in current subdomain and boundary halo (z-direction)
- `integer` *mxpcell*

  Maximum number of particles per interaction link cell.
- `real(kind=dp)` *wdthx*

  Size of interaction link cell in x-direction.
- `real(kind=dp)` *wdthy*

  Size of interaction link cell in y-direction.
- `real(kind=dp)` *wdthz*

  Size of interaction link cell in z-direction.
- `integer, dimension(:), allocatable` *lct*

  Starting particles in interaction linked-cell lists.

- `integer, dimension(:), allocatable` *link*

  Array with subsequent particles in interaction linked-cell lists.
- `integer, dimension(:), allocatable` *lcell*

  Interaction link cell numbers for subdomain.
- `integer, dimension(:), allocatable` *lcell_neighbour*

  Neighbouring link cells for each interaction link cell in subdomain.
- `integer, dimension(:), allocatable` *lcell_therm*

  Link cell thermostat switches.
- `logical` *lnsmall*

  Small system interaction link cell switch.
- `integer` *nlewx*

  Number of electrostatic link cells in current subdomain (x-direction)
- `integer` *nlewy*

  Number of electrostatic link cells in current subdomain (y-direction)
- `integer` *nlewz*

  Number of electrostatic link cells in current subdomain (z-direction)
- `integer` *nlewx2*

  Number of electrostatic link cells in current subdomain and boundary halo (x-direction)
- `integer` *nlewy2*

  Number of electrostatic link cells in current subdomain and boundary halo (y-direction)
- `integer` *nlewz2*

  Number of electrostatic link cells in current subdomain and boundary halo (z-direction)
- `real(kind=dp)` *wdthewx*

  Size of electrostatic link cell in x-direction.
- `real(kind=dp)` *wdthewy*

  Size of electrostatic link cell in y-direction.
- `real(kind=dp)` *wdthewz*

  Size of electrostatic link cell in z-direction.
- `integer, dimension(:), allocatable` *lctew*

  Starting particles in electrostatic linked-cell lists.
- `integer, dimension(:), allocatable` *linkew*

  Array with subsequent particles in electrostatic linked-cell lists.
- `integer, dimension(:), allocatable` *lcellew*

  Electrostatic link cell numbers for subdomain.
- `integer, dimension(:), allocatable` *lcellew_neighbour*

  Neighbouring link cells for each electrostatic link cell in subdomain.
- `logical` *lnewsmall*

  Small system electrostatic link cell switch.

- `integer` *nlmbx*

  Number of many-body DPD link cells in current subdomain (x-direction)
- `integer` *nlmby*

  Number of many-body DPD link cells in current subdomain (y-direction)
- `integer` *nlmbz*

  Number of many-body DPD link cells in current subdomain (z-direction)
- `integer` *nlmbx2*

  Number of many-body DPD link cells in current subdomain and boundary halo (x-direction)
- `integer` *nlmby2*

  Number of many-body DPD link cells in current subdomain and boundary halo (y-direction)
- `integer` *nlmbz2*

  Number of many-body DPD link cells in current subdomain and boundary halo (z-direction)
- `real(kind=dp)` *wdthmbx*

  Size of many-body DPD link cell in x-direction.
- `real(kind=dp)` *wdthmby*

  Size of many-body DPD link cell in y-direction.
- `real(kind=dp)` *wdthmbz*

  Size of many-body DPD link cell in z-direction.
- `integer, dimension(:), allocatable` *lctmb*

  Starting particles in many-body DPD linked-cell lists.
- `integer, dimension(:), allocatable` *linkmb*

  Array with subsequent particles in many-body DPD linked-cell lists.
- `integer, dimension(:), allocatable` *lcellmb*

  Many-body DPD link cell numbers for subdomain.
- `integer, dimension(:), allocatable` *lcellmb_neighbour*

  Neighbouring link cells for each many-body DPD link cell in subdomain.
- `logical` *lnmbsmall*

  Small system many-body DPD link cell switch.
- `character(len=8), dimension(:), allocatable, save` *namspe*

  Names of particle species.
- `integer, dimension(:), allocatable` *ktype*

  Interaction potential types.
- `real(kind=dp), dimension(:), allocatable` *amass*

  Particle masses for species.
- `real(kind=dp), dimension(:), allocatable` *chge*

  Particle charges for species.
- `real(kind=dp), dimension(:,:), allocatable` *vvv*

  Interaction parameters.

- `integer, dimension(:), allocatable` *lfrzn*

  Frozen particle switch for species.

- `real(kind=dp), dimension(:), allocatable` *lsurf*

  Switch for species pairs counting as surface interactions.

- `real(kind=dp), dimension(2)` *clr*

  Long-range corrections for potential energy and virial.

- `real(kind=dp), dimension(:,:), allocatable, target` *cfxfyfz*

  Corrective forces on frozen particles to remove reciprocal-space Ewald sum contributions.

- `real(kind=dp), dimension(:), pointer` *fcfx*

  Corrective electrostatic forces on frozen particles (x-component, pointer)

- `real(kind=dp), dimension(:), pointer` *fcfy*

  Corrective electrostatic forces on frozen particles (y-component, pointer)

- `real(kind=dp), dimension(:), pointer` *fcfz*

  Corrective electrostatic forces on frozen particles (z-component, pointer)

- `real(kind=dp), dimension(36)` *strcfz*

  Corrective stress tensor to remove reciprocal-space Ewald sum contributions for frozen particles.

- `real(kind=dp), dimension(3)` *vrlcfz*

  Corrective virial to remove reciprocal-space Ewald sum contributions for frozen particles.

- `real(kind=dp)` *potcfz*

  Corrective potential energy to remove reciprocal-space Ewald sum contributions for frozen particles.

- `integer` *itype*

  Selected thermostat and integration type.

- `real(kind=dp), dimension(:), allocatable` *gamma*

  Dissipative force parameters/collision frequencies.

- `real(kind=dp), dimension(:), allocatable` *sigma*

  Random force parameters/thermostat probabilities.

- `real(kind=dp)` *alphasg*

  Stoyanov-Groot temperature coupling parameter.

- `real(kind=dp), dimension(:), allocatable, save` *pldxyz*

  Alternative thermostat particle pair vectors.

- `integer, dimension(:), allocatable, save` *plparti*

  Alternative thermostat first local particle numbers.

- `integer, dimension(:), allocatable, save` *plpartj*

  Alternative thermostat second local particle numbers.

- `integer, dimension(:), allocatable, save` *plproci*

  Alternative thermostat first processor/subdomain numbers.

- `integer, dimension(:), allocatable, save` *plprocj*

  Alternative thermostat second processor/subdomain numbers.

- `integer, dimension(:), allocatable, save` *plbound*

  Alternative thermostat boundary thermostatting keys.
- `integer, dimension(:), allocatable, save` *plintij*

  Alternative thermostat particle pair identifiers.
- `integer` *npair*

  Alternative thermostat pair list size.
- `integer` *btype*

  Selected barostat and ensemble.
- `real(kind=dp)` *prszero*

  Target system pressure for barostat.
- `real(kind=dp)` *abaro*

  First barostat parameter.
- `real(kind=dp)` *bbaro*

  Second barostat parameter.
- `real(kind=dp)` *cbaro*

  Third barostat parameter.
- `real(kind=dp)` *dbaro*

  Fourth barostat parameter.
- `real(kind=dp)` *upx*

  Barostat piston velocity (x-component) at current timestep
- `real(kind=dp)` *upy*

  Barostat piston velocity (y-component) at current timestep
- `real(kind=dp)` *upz*

  Barostat piston velocity (z-component) at current timestep
- `real(kind=dp)` *up1x*

  Barostat piston velocity (x-component) at previous timestep
- `real(kind=dp)` *up1y*

  Barostat piston velocity (y-component) at previous timestep
- `real(kind=dp)` *up1z*

  Barostat piston velocity (z-component) at previous timestep
- `real(kind=dp)` *psmass*

  Barostat piston mass.
- `real(kind=dp)` *rpsmass*

  Reciprocal of barostat piston mass.
- `real(kind=dp)` *fpx*

  Barostat piston force (x-component)
- `real(kind=dp)` *fpy*

  Barostat piston force (y-component)

- `real(kind=dp)` *fpz*

  Barostat piston force (z-component)
- `real(kind=dp), dimension(3)` *ivrl*

  Instantaneous system virial.
- `real(kind=dp)` *sigmalang*

  Barostat random parameter.
- `integer` *etype*

  Selected electrostatic model and application method.
- `real(kind=dp)` *gammaelec*

  Permittivity coefficient.
- `real(kind=dp)` *bjerelec*

  Bjerrum length.
- `real(kind=dp)` *qchg*

  Total system charge.
- `real(kind=dp)` *alphaew*

  Ewald real-space convergence coefficient.
- `real(kind=dp)` *ralphaew*

  Reciprocal of Ewald real-space convergence coefficient.
- `integer` *kmax1*

  Maximum reciprocal-space vector or k-vector (x-component)
- `integer` *kmax2*

  Maximum reciprocal-space vector or k-vector (y-component)
- `integer` *kmax3*

  Maximum reciprocal-space vector or k-vector (z-component)
- `real(kind=dp)` *engsic*

  Ewald sum self-interaction correction term to potential.
- `real(kind=dp)` *qfixv*

  Ewald sum charged system correction term to potential.
- `real(kind=dp)` *betaew*

  Charge smearing parameter.
- `real(kind=dp)` *chglen*

  Charge smearing length.
- `real(kind=dp), dimension(:,:), allocatable` *rkxyz*

  List of reciprocal space vectors within range for maximum k-vector.
- `integer, dimension(:,:), allocatable` *kxyz*

  List of available reciprocal-space or k-vectors.
- `integer` *nlistew*

  Number of available reciprocal-space or k-vectors.

- `integer` *mxspl*

  B-spline interpolation order for SPME.
- `real(kind=dp)` *vgapx*

  Vacuum gap for slab geometries (x-component)
- `real(kind=dp)` *vgapy*

  Vacuum gap for slab geometries (y-component)
- `real(kind=dp)` *vgapz*

  Vacuum gap for slab geometries (z-component)
- `real(kind=dp)` *ewprec*

  Relative precision (maximum error) of Ewald sum or SPME calculations.
- `real(kind=dp), dimension(:), allocatable` *aabond*

  First interaction parameters for bond interactions.
- `real(kind=dp), dimension(:), allocatable` *bbbond*

  Second interaction parameters for bond interactions.
- `real(kind=dp), dimension(:), allocatable` *ccbond*

  Third interaction parameters for bond interactions.
- `real(kind=dp), dimension(:), allocatable` *ddbond*

  Fourth interaction parameters for bond interactions.
- `real(kind=dp), dimension(:), allocatable` *aaang*

  First interaction parameters for angle interactions.
- `real(kind=dp), dimension(:), allocatable` *bbang*

  Second interaction parameters for angle interactions.
- `real(kind=dp), dimension(:), allocatable` *ccang*

  Third interaction parameters for angle interactions.
- `real(kind=dp), dimension(:), allocatable` *ddang*

  Fourth interaction parameters for angle interactions.
- `real(kind=dp), dimension(:), allocatable` *aadhd*

  First interaction parameters for dihedral interactions.
- `real(kind=dp), dimension(:), allocatable` *bbdhd*

  Second interaction parameters for dihedral interactions.
- `real(kind=dp), dimension(:), allocatable` *ccdhd*

  Third interaction parameters for dihedral interactions.
- `real(kind=dp), dimension(:), allocatable` *dddhd*

  Fourth interaction parameters for dihedral interactions.
- `integer, dimension(:), allocatable` *bdtype*

  Types (functional forms) of bond interactions.
- `integer, dimension(:), allocatable` *angtype*

  Types (functional forms) of angle interactions.

- `integer, dimension(:), allocatable` *dhdtype*

  Types (functional forms) of dihedral interactions.

- `logical, dimension(:), allocatable` *moliso*

  Molecular isomer switch.

- `integer, dimension(:), allocatable` *nspec*

  Number of particles for each species not included in molecules.

- `integer, dimension(:), allocatable` *nspecmol*

  Number of particles for each species included in molecules.

- `integer, dimension(:), allocatable` *nmol*

  Number of molecules for each type.

- `integer, dimension(:), allocatable` *nbdmol*

  Number of particles per molecule type.

- `character(len=8), dimension(:), allocatable, save` *nammol*

  Names of molecule types.

- `integer, dimension(:,:), allocatable, save` *mlstrtspe*

  Species for each particle in a specified molecule type.

- `integer, dimension(:), allocatable, save` *nbond*

  Number of bonds for each molecule type.

- `integer, dimension(:), allocatable, save` *nangle*

  Number of angles for each molecule type.

- `integer, dimension(:), allocatable, save` *ndihed*

  Number of dihedrals for each molecule type.

- `integer, dimension(:,:), allocatable, save` *bdinp1*

  Bond connectivity data for inserting molecule in system (first particle index)

- `integer, dimension(:,:), allocatable, save` *bdinp2*

  Bond connectivity data for inserting molecule in system (second particle index)

- `integer, dimension(:,:), allocatable, save` *bdinp3*

  Bond connectivity data for inserting molecule in system (bond type)

- `integer, dimension(:,:), allocatable, save` *anginp1*

  Angle connectivity data for inserting molecule in system (first particle index)

- `integer, dimension(:,:), allocatable, save` *anginp2*

  Angle connectivity data for inserting molecule in system (second particle index)

- `integer, dimension(:,:), allocatable, save` *anginp3*

  Angle connectivity data for inserting molecule in system (third particle index)

- `integer, dimension(:,:), allocatable, save` *anginp4*

  Angle connectivity data for inserting molecule in system (angle type)

- `integer, dimension(:,:), allocatable, save` *dhdinp1*

  Dihedral connectivity data for inserting molecule in system (first particle index)

- `integer, dimension(:,:), allocatable, save` *dhdinp2*

  Dihedral connectivity data for inserting molecule in system (second particle index)
- `integer, dimension(:,:), allocatable, save` *dhdinp3*

  Dihedral connectivity data for inserting molecule in system (third particle index)
- `integer, dimension(:,:), allocatable, save` *dhdinp4*

  Dihedral connectivity data for inserting molecule in system (fourth particle index)
- `integer, dimension(:,:), allocatable, save` *dhdinp5*

  Dihedral connectivity data for inserting molecule in system (dihedral type)
- `integer, dimension(:), allocatable, save` *molstart*

  Starting global particle indices for molecules inserted in system.
- `real(kind=dp), dimension(:), allocatable, save` *mlszx*

  Maximum extent of molecule inserted in system (x-component)
- `real(kind=dp), dimension(:), allocatable, save` *mlszy*

  Maximum extent of molecule inserted in system (y-component)
- `real(kind=dp), dimension(:), allocatable, save` *mlszz*

  Maximum extent of molecule inserted in system (z-component)
- `real(kind=dp), dimension(:,:), allocatable, save` *mlstrtxxx*

  Relative positions of particles in molecule inserted in system (x-component)
- `real(kind=dp), dimension(:,:), allocatable, save` *mlstrtyyy*

  Relative positions of particles in molecule inserted in system (y-component)
- `real(kind=dp), dimension(:,:), allocatable, save` *mlstrtzzz*

  Relative positions of particles in molecule inserted in system (z-component)
- `integer, dimension(:,:), allocatable, save` *bndtbl*

  Bond book-keeping table.
- `integer, dimension(:,:), allocatable, save` *angtbl*

  Angle book-keeping table.
- `integer, dimension(:,:), allocatable, save` *dhdtbl*

  Dihedral book-keeping table.
- `integer` *nbonds*

  Number of bonds in subdomain book-keeping table.
- `integer` *nangles*

  Number of angles in subdomain book-keeping table.
- `integer` *ndiheds*

  Number of dihedrals in subdomain book-keeping table.
- `integer, dimension(:,:), allocatable, save` *lblclst*

  Look-up list of global and local particle indices.
- `integer` *nlist*

  Number of particles in global/local index look-up list.

- `real(kind=dp), dimension(:, :), allocatable` *rhomb*

  Local densities for many-body DPD.
- `integer` *srftype*

  Selected surface type.
- `integer` *srfx*

  Switch for surface orthogonal to x-axis.
- `integer` *srfy*

  Switch for surface orthogonal to y-axis.
- `integer` *srfz*

  Switch for surface orthogonal to z-axis.
- `logical, dimension(6)` *srflgc*

  Switches for surfaces in current processor/subdomain.
- `real(kind=dp), dimension(:, :), allocatable` *vvsrf*

  Surface interaction parameters.
- `integer, dimension(:), allocatable` *srfktype*

  Surface interaction potential types.
- `real(kind=dp)` *srfpos*

  Distance between simulation box boundary and reflecting wall surfaces.
- `real(kind=dp)` *srfxps1*

  Position of reflecting wall surface in processor with surface 1 (-x)
- `real(kind=dp)` *srfxps2*

  Position of reflecting wall surface in processor with surface 2 (+x)
- `real(kind=dp)` *srfyps1*

  Position of reflecting wall surface in processor with surface 3 (-y)
- `real(kind=dp)` *srfyps2*

  Position of reflecting wall surface in processor with surface 4 (+y)
- `real(kind=dp)` *srfzps1*

  Position of reflecting wall surface in processor with surface 5 (-z)
- `real(kind=dp)` *srfzps2*

  Position of reflecting wall surface in processor with surface 6 (+z)
- `integer` *frzwspe*

  Particle species for frozen bead wall.
- `integer` *npxfwx*

  Number of particles in x-axis frozen bead walls (x-direction)
- `integer` *npxfwy*

  Number of particles in x-axis frozen bead walls (y-direction)
- `integer` *npxfwz*

  Number of particles in x-axis frozen bead walls (z-direction)

- `integer` *npyfwx*

  Number of particles in y-axis frozen bead walls (x-direction)
- `integer` *npyfwy*

  Number of particles in y-axis frozen bead walls (y-direction)
- `integer` *npyfwz*

  Number of particles in y-axis frozen bead walls (z-direction)
- `integer` *npzfwx*

  Number of particles in z-axis frozen bead walls (x-direction)
- `integer` *npzfwy*

  Number of particles in z-axis frozen bead walls (y-direction)
- `integer` *npzfwz*

  Number of particles in z-axis frozen bead walls (z-direction)
- `real(kind=dp)` *frzwdens*

  Density of particles in frozen bead walls.
- `real(kind=dp)` *frzwxwid*

  Width of frozen bead wall orthogonal to x-axis.
- `real(kind=dp)` *frzwywid*

  Width of frozen bead wall orthogonal to y-axis.
- `real(kind=dp)` *frzwzwid*

  Width of frozen bead wall orthogonal to z-axis.
- `logical` *lfrzx*

  Switch for x-axis frozen bead walls.
- `logical` *lfrzy*

  Switch for y-axis frozen bead walls.
- `logical` *lfrzz*

  Switch for z-axis frozen bead walls.
- `logical` *lfrzwall*

  Switch for frozen bead walls.
- `real(kind=dp)` *shrvx*

  Velocity of Lees-Edwards shearing boundary (x-component)
- `real(kind=dp)` *shrvy*

  Velocity of Lees-Edwards shearing boundary (y-component)
- `real(kind=dp)` *shrvz*

  Velocity of Lees-Edwards shearing boundary (z-component)
- `real(kind=dp)` *shrdx*

  Displacement of Lees-Edwards shearing boundary (x-component)
- `real(kind=dp)` *shrdy*

  Displacement of Lees-Edwards shearing boundary (y-component)

- `real(kind=dp)` *shrdz*

  Displacement of Lees-Edwards shearing boundary (z-component)

- `integer` *nshrs*

  Number of timesteps before applying Lees-Edwards shearing boundary.

- `real(kind=dp)` *bdfrcx*

  Body force on particles (x-component)

- `real(kind=dp)` *bdfrcy*

  Body force on particles (y-component)

- `real(kind=dp)` *bdfrcz*

  Body force on particles (z-component)

- `real(kind=dp)` *elecx*

  Electric field on charged particles (x-component)

- `real(kind=dp)` *elecy*

  Electric field on charged particles (y-component)

- `real(kind=dp)` *elecz*

  Electric field on charged particles (z-component)

- `real(kind=dp), dimension(:,:), allocatable, target` *fxfyfz*

  Particle forces.

- `real(kind=dp), dimension(:), pointer` *fxx*

  Particle force (x-component, pointer)

- `real(kind=dp), dimension(:), pointer` *fyy*

  Particle force (y-component, pointer)

- `real(kind=dp), dimension(:), pointer` *fzz*

  Particle force (z-component, pointer)

- `real(kind=dp), dimension(:,:), allocatable, target` *vfxfyfz*

  Variable (recalculated dissipative or Stoyanov-Groot temperature-dependent) particle forces.

- `real(kind=dp), dimension(:), pointer` *fvx*

  Variable particle force (x-component, pointer)

- `real(kind=dp), dimension(:), pointer` *fvy*

  Variable particle force (y-component, pointer)

- `real(kind=dp), dimension(:), pointer` *fvz*

  Variable particle force (z-component, pointer)

- `real(kind=dp), dimension(:,:), allocatable, target` *vxvyvz*

  Particle velocities.

- `real(kind=dp), dimension(:), pointer` *vxx*

  Particle velocity (x-component, pointer)

- `real(kind=dp), dimension(:), pointer` *vyy*

  Particle velocity (y-component, pointer)

- `real(kind=dp), dimension(:), pointer` *vzz*

  Particle velocity (z-component, pointer)
- `real(kind=dp), dimension(:,:), allocatable, target` *xxyyzz*

  Particle positions (coordinates)
- `real(kind=dp), dimension(:), pointer` *xxx*

  Particle position (x-component, pointer)
- `real(kind=dp), dimension(:), pointer` *yyy*

  Particle position (y-component, pointer)
- `real(kind=dp), dimension(:), pointer` *zzz*

  Particle position (z-component, pointer)
- `integer, dimension(:), allocatable, save` *lab*

  Particle global index.
- `integer, dimension(:), allocatable, save` *ltp*

  Particle species.
- `integer, dimension(:), allocatable, save` *ltm*

  Particle molecule type.
- `integer, dimension(:), allocatable, save` *lmp*

  Owning processor for particle.
- `integer, dimension(:), allocatable, save` *loc*

  Particle local index on owning processor.
- `character(len=8), dimension(:), allocatable, save` *atmnam*

  Particle species name.
- `character(len=8), dimension(:), allocatable, save` *molnam*

  Particle molecule name.
- `real(kind=dp), dimension(:), allocatable, save` *weight*

  Particle mass.
- `real(kind=dp)` *pe*

  Total potential (interaction) energy.
- `real(kind=dp)` *vir*

  Total virial.
- `real(kind=dp)` *be*

  Bond energy.
- `real(kind=dp)` *ae*

  Angle energy.
- `real(kind=dp)` *de*

  Dihedral energy.
- `real(kind=dp)` *ee*

  Electrostatic energy.

- `real(kind=dp)` *se*

  Surface energy.

- `real(kind=dp)` *bdlng*

  Summed bond lengths.

- `real(kind=dp)` *bdlmin*

  Minimum bond length.

- `real(kind=dp)` *bdlmax*

  Maximum bond length.

- `real(kind=dp)` *bdang*

  Summed bond angles.

- `real(kind=dp)` *bddhd*

  Summed bond dihedrals.

- `real(kind=dp), dimension(3)` *tke*

  Kinetic energy separated into x-, y- and z-components.

- `real(kind=dp), dimension(36)` *stress*

  Stress tensor separated into conservative, dissipative, random and kinetic contributions.

- `real(kind=dp)` *stppe*

  Total potential energy per particle at current timestep.

- `real(kind=dp)` *stpvir*

  Total virial per particle at current timestep.

- `real(kind=dp)` *stptke*

  Total kinetic energy per particle at current timestep.

- `real(kind=dp)` *stpte*

  Total energy per particle at current timestep.

- `real(kind=dp)` *stpprs*

  System pressure at current timestep.

- `real(kind=dp)` *stpvlm*

  System volume at current timestep.

- `real(kind=dp)` *stpzts*

  System surface tension in z-direction at current timestep.

- `real(kind=dp)` *stpttp*

  System temperature at current timestep.

- `real(kind=dp)` *stptpx*

  Partial temperature in x-direction at current timestep.

- `real(kind=dp)` *stptpy*

  Partial temperature in y-direction at current timestep.

- `real(kind=dp)` *stptpz*

  Partial temperature in z-direction at current timestep.

- `real(kind=dp)` *stpbe*

  Total bond energy per particle at current timestep.
- `real(kind=dp)` *stpae*

  Total angle energy per particle at current timestep.
- `real(kind=dp)` *stpde*

  Total dihedral energy per particle at current timestep.
- `real(kind=dp)` *stpee*

  Total electrostatic energy per particle at current timestep.
- `real(kind=dp)` *stpse*

  Total surface energy per particle at current timestep.
- `real(kind=dp)` *stpbdl*

  Mean bond length at current timestep.
- `real(kind=dp)` *stpbdmx*

  Maximum bond length at current timestep.
- `real(kind=dp)` *stpbdmn*

  Minimum bond length at current timestep.
- `real(kind=dp)` *stpang*

  Mean bond angle at current timestep.
- `real(kind=dp)` *stpdhd*

  Mean bond dihedral at current timestep.
- `real(kind=dp), dimension(stksize)` *rav*

  Current rolling averages of system properties using values from statistical stacks.
- `real(kind=dp), dimension(statsize)` *ave*

  Current all-timestep averages of system properties.
- `real(kind=dp), dimension(statsize)` *flc*

  Current all-timestep fluctuations (variances) of system properties.
- `real(kind=dp), dimension(:), allocatable` *stkpe*

  Statistical stack of potential energy per particle values.
- `real(kind=dp), dimension(:), allocatable` *stktkex*

  Statistical stack of x-component kinetic energy per particle values.
- `real(kind=dp), dimension(:), allocatable` *stktkey*

  Statistical stack of y-component kinetic energy per particle values.
- `real(kind=dp), dimension(:), allocatable` *stktkez*

  Statistical stack of z-component kinetic energy per particle values.
- `real(kind=dp), dimension(:), allocatable` *stkbe*

  Statistical stack of bond energy per particle values.
- `real(kind=dp), dimension(:), allocatable` *stkae*

  Statistical stack of angle energy per particle values.

- `real(kind=dp), dimension(:), allocatable` *stkde*

  Statistical stack of dihedral energy per particle values.
- `real(kind=dp), dimension(:), allocatable` *stkee*

  Statistical stack of electrostatic energy per particle values.
- `real(kind=dp), dimension(:), allocatable` *stkse*

  Statistical stack of surface energy per particle values.
- `real(kind=dp), dimension(:), allocatable` *stkvir*

  Statistical stack of virial per particle values.
- `real(kind=dp), dimension(:), allocatable` *stkvlm*

  Statistical stack of system volume values.
- `real(kind=dp), dimension(:), allocatable` *stkzts*

  Statistical stack of surface tension in z-direction values.
- `integer` *nav*

  Current number of timesteps for statistical sampling.
- `real(kind=dp), dimension(11)` *zum*

  Summed values of statistical stacked properties.
- `real(kind=dp), dimension(:,:), allocatable, target, save` *commsinbuf*

  Communication buffers for receiving data.
- `real(kind=dp), dimension(:,:), allocatable, target, save` *commsoutbuf*

  Communication buffers for sending data.

10.3.3 Variable Documentation

aaang

```
real(kind=dp), dimension (:), allocatable variables::aaang
```

Array with first angle interaction parameters given for each angle type specified in FIELD file.

aabond

```
real(kind=dp), dimension (:), allocatable variables::aabond
```

Array with first bond interaction parameters given for each bond type specified in FIELD file.

aadhd

```
real(kind=dp), dimension (:), allocatable variables::aadhd
```

Array with first dihedral interaction parameters given for each dihedral type specified in FIELD file.

abaro

```
real(kind=dp) variables::abaro
```

First barostat parameter: either barostat relaxation time τ_p for Langevin barostat, or ratio of compressibility to relaxation time $\frac{\beta}{\tau_p}$ for Berendesen barostat.

ae

```
real(kind=dp) variables::ae
```

Total potential energy for system resulting from angle interactions.

alphaew

```
real(kind=dp) variables::alphaew
```

Real-space convergence coefficient for electrostatic interactions with Ewald sums, α.

alphasg

```
real(kind=dp) variables::alphasg
```

Temperature-dependent force coupling parameter used for Stoyanov-Groot thermostat, α^T.

amass

```
real(kind=dp), dimension (:), allocatable variables::amass
```

Masses for particles of available species (in DPD mass units).

anginp1

```
integer, dimension (:,:), allocatable, save variables::anginp1
```

Array with first (relative) particle number in molecule included in current angle, used to set up angle tables for calculations.

anginp2

```
integer, dimension (:,:), allocatable, save variables::anginp2
```

Array with second (relative) particle number in molecule included in current angle, used to set up angle tables for calculations.

anginp3

```
integer, dimension (:,:), allocatable, save variables::anginp3
```

Array with third (relative) particle number in molecule included in current angle, used to set up angle tables for calculations.

anginp4

```
integer, dimension (:,:), allocatable, save variables::anginp4
```

Array with identified angle type for current angle, used to set up angle tables for calculations.

angtbl

```
integer, dimension (:,:), allocatable, save variables::angtbl
```

Array listing available angles in processor's subdomain for interaction calculations: first index is entry in angle table, second index is either global particle numbers involved in angle (1, 2, 3) or angle type (4).

angtype

```
integer, dimension (:), allocatable variables::angtype
```

Array with flags indicating the functional forms for each angle type specified in FIELD file: 1 = harmonic, 2 = harmonic cosine, 3 = cosine

atmnam

```
character(len=8), dimension (:), allocatable, save variables::atmnam
```

Species names (up to 8 characters) of particles

ave

```
real(kind=dp), dimension (statsize) variables::ave
```

Current average values of system properties obtained from properties over all timesteps after equilibration: 1 = total energy per particle, 2 = total potential energy per particle, 3 = total electrostatic energy per particle, 4 = total bond energy per particle, 5 = total angle energy per particle, 6 = total dihedral energy per particle, 7 = total virial per particle, 8 = total kinetic energy per particle, 9 = system pressure, 10 = system volume, 11 = surface tension in z-direction, 12 = system temperature, 13 = partial temperature in x-direction, 14 = partial temperature in y-direction, 15 = partial temperature in z-direction, 16-51 = pressure tensors separated out into conservative (potential), dissipative, random and kinetic contributions for all tensor components.

bbang

```
real(kind=dp), dimension (:), allocatable variables::bbang
```

Array with second angle interaction parameters given for each angle type specified in FIELD file.

bbaro

```
real(kind=dp) variables::bbaro
```

Second barostat parameter: either barostat dissipative parameter γ_p for Langevin barostat, or target surface tension γ_0 for Berendsen barostat.

bbbond

```
real(kind=dp), dimension (:), allocatable variables::bbbond
```

Array with second bond interaction parameters given for each bond type specified in FIELD file.

bbdhd

```
real(kind=dp), dimension (:), allocatable variables::bbdhd
```

Array with second dihedral interaction parameters given for each dihedral type specified in FIELD file.

bdang

```
real(kind=dp) variables::bdang
```

Total angles between pairs of bonds in molecules.

bddhd

```
real(kind=dp) variables::bddhd
```

Total dihedrals between pairs of bond planes in molecules.

bdfrcx

```
real(kind=dp) variables::bdfrcx
```

Constant acceleration acting on all moving particles (x-component).

bdfrcy

```
real(kind=dp) variables::bdfrcy
```

Constant acceleration acting on all moving particles (y-component).

bdfrcz

```
real(kind=dp) variables::bdfrcz
```

Constant acceleration acting on all moving particles (z-component).

bdinp1

```
integer, dimension (:,:), allocatable, save variables::bdinp1
```

Array with first (relative) particle number in molecule included in current bond, used to set up bond tables for calculations.

bdinp2

```
integer, dimension (:,:), allocatable, save variables::bdinp2
```

Array with second (relative) particle number in molecule included in current bond, used to set up bond tables for calculations.

bdinp3

```
integer, dimension (:,:), allocatable, save variables::bdinp3
```

Array with identified bond type for current bond, used to set up bond tables for calculations.

bdlmax

```
real(kind=dp) variables::bdlmax
```

Maximum length of bonds between pairs of particles in molecules.

bdlmin

```
real(kind=dp) variables::bdlmin
```

Minimum length of bonds between pairs of particles in molecules.

bdlng

```
real(kind=dp) variables::bdlng
```

Total lengths of bonds between pairs of particles in molecules.

bdtype

```
integer, dimension (:), allocatable variables::bdtype
```

Array with flags indicating the functional forms for each bond type specified in FIELD file: 1 = harmonic spring, 2 = finitely extensible non-linear elastic (FENE), 3 = Marko/Siggia worm-like chain (WLC), 4 = Morse anharmonic.

be

```
real(kind=dp) variables::be
```

Total potential energy for system resulting from bond interactions.

betaew

```
real(kind=dp) variables::betaew
```

Parameter used in calculating electrostatic interactions with charge smearing (usually related to reciprocal of charge smearing length).

bjerelec

```
real(kind=dp) variables::bjerelec
```

Bjerrum length used for electrostatic interactions, l_B (related to permittivity coefficient)

bndtbl

```
integer, dimension (:,:), allocatable, save variables::bndtbl
```

Array listing available bonds in processor's subdomain for interaction calculations: first index is entry in bond table, second index is either global particle numbers involved in bond (1, 2) or bond type (3).

btype

```
integer variables::btype
```

Flag indicating barostat and ensemble type specified in CONTROL file: 0 = no barostat, 1 = Lsngevin barostat with constant pressure (NPT) ensemble, 2 = Langevin barostat with constant surface area (NP_nAT) ensemble, 3 = Langevin barostat with constant surface tension ($NP_n\gamma T$) ensemble, 4 = Berendsen barostat with constant pressure (NPT) ensemble, 5 = Berendsen barostat with constant surface area (NP_nAT) ensemble, 6 = Berendsen barostat with constant surface tension ($NP_n\gamma T$) ensemble.

cbaro

```
real(kind=dp) variables::cbaro
```

Third barostat parameter: target surface tension γ_0 for Langevin barostat.

ccang

```
real(kind=dp), dimension (:), allocatable variables::ccang
```

Array with third angle interaction parameters given for each angle type specified in FIELD file.

ccbond

```
real(kind=dp), dimension (:), allocatable variables::ccbond
```

Array with third bond interaction parameters given for each bond type specified in FIELD file.

ccdhd

```
real(kind=dp), dimension (:), allocatable variables::ccdhd
```

Array with third dihedral interaction parameters given for each dihedral type specified in FIELD file.

cfxfyfz

```
real(kind=dp), dimension (:,:), allocatable, target variables::cfxfyfz
```

Forces used to correct reciprocal-space Ewald sum contributions to remove the effects due to frozen-frozen particle pairs (already excluded in real space): first index of array is coordinate, second index is local particle number. These forces only need calculating once for constant volume (NVT) ensembles but must be recalculated when using a barostat due to particle position changes.

chge

```
real(kind=dp), dimension (:), allocatable variables::chge
```

Charges (valencies) for particles of available species.

chglen

```
real(kind=dp) variables::chglen
```

Lengthscale for charge smearing used in electrostatic interactions.

clr

```
real(kind=dp), dimension (2) variables::clr
```

Corrections to potential energy (1) and virial (2) due to long-range effects of given interaction potentials (particularly Lennard-Jones).

commsinbuf

```
real(kind=dp), dimension(:,:), allocatable, target, save variables::commsinbuf
```

Array used for inter-processor communications to receive particle data from other processors: first index is number of values being received per buffer, second index is buffer number (0 for serial calculations, up to 4 for parallel calculations with Lees-Edwards boundary conditions, 1 for all other parallel calculations).

commsoutbuf

```
real(kind=dp), dimension(:,:), allocatable, target, save variables::commsoutbuf
```

Array used for inter-processor communications to send particle data to other processors: first index is number of values being sent per buffer, second index is buffer number (0 for serial calculations, up to 4 for parallel calculations with Lees-Edwards boundary conditions, 1 for all other parallel calculations).

dbaro

```
real(kind=dp) variables::dbaro
```

Fourth barostat parameter (not currently used).

ddang

```
real(kind=dp), dimension (:), allocatable variables::ddang
```

Array with fourth angle interaction parameters given for each angle type specified in FIELD file.

ddbond

```
real(kind=dp), dimension (:), allocatable variables::ddbond
```

Array with fourth bond interaction parameters given for each bond type specified in FIELD file.

dddhd

```
real(kind=dp), dimension (:), allocatable variables::dddhd
```

Array with fourth dihedral interaction parameters given for each dihedral type specified in FIELD file.

de

```
real(kind=dp) variables::de
```

Total potential energy for system resulting from dihedral interactions.

delx

```
real(kind=dp) variables::delx
```

x-coordinate of bottom left back corner of current processor's subdomain relative to bottom left back corner of entire simulation box.

dely

```
real(kind=dp) variables::dely
```

y-coordinate of bottom left back corner of current processor's subdomain relative to bottom left back corner of entire simulation box.

delz

```
real(kind=dp) variables::delz
```

z-coordinate of bottom left back corner of current processor's subdomain relative to bottom left back corner of entire simulation box.

dhdinp1

```
integer, dimension (:,:), allocatable, save variables::dhdinp1
```

Array with first (relative) particle number in molecule included in current dihedral, used to set up dihedral tables for calculations.

dhdinp2

```
integer, dimension (:,:), allocatable, save variables::dhdinp2
```

Array with second (relative) particle number in molecule included in current dihedral, used to set up dihedral tables for calculations.

dhdinp3

```
integer, dimension (:,:), allocatable, save variables::dhdinp3
```

Array with third (relative) particle number in molecule included in current dihedral, used to set up dihedral tables for calculations.

dhdinp4

```
integer, dimension (:,:), allocatable, save variables::dhdinp4
```

Array with fourth (relative) particle number in molecule included in current dihedral, used to set up dihedral tables for calculations.

dhdinp5

```
integer, dimension (:,:), allocatable, save variables::dhdinp5
```

Array with identified dihedral type for current dihedral, used to set up dihedral tables for calculations.

dhdtbl

```
integer, dimension (:,:), allocatable, save variables::dhdtbl
```

Array listing available dihedral in processor's subdomain for interaction calculations: first index is entry in dihedral table, second index is either global particle numbers involved in dihedral (1, 2, 3, 4) or dihedral type (5).

dhdtype

```
integer, dimension (:), allocatable variables::dhdtype
```

Array with flags indicating the functional forms for each dihedral type specified in FIELD file: 1 = cosine (torsion), 2 = harmonic improper, 3 = harmonic cosine.

dimx

```
real(kind=dp) variables::dimx
```

Simulation box size in x-direction.

dimxcell

```
real(kind=dp) variables::dimxcell
```

Simulation box size in x-direction for unit cell defined by CONFIG file.

dimy

```
real(kind=dp) variables::dimy
```

Simulation box size in y-direction.

dimycell

```
real(kind=dp) variables::dimycell
```

Simulation box size in y-direction for unit cell defined by CONFIG file.

dimz

```
real(kind=dp) variables::dimz
```

Simulation box size in z-direction.

dimzcell

```
real(kind=dp) variables::dimzcell
```

Simulation box size in z-direction for unit cell defined by CONFIG file.

dvar

```
real(kind=dp) variables::dvar
```

Density multiplier used during simulation setup to increase numbers of particles per processor, in boundary halo etc. for simulations with variable densities (e.g. when using many-body DPD).

ee

```
real(kind=dp) variables::ee
```

Total potential energy for system resulting from electrostatic interactions.

elecx

```
real(kind=dp) variables::elecx
```

Electric field acting on all charged particles (x-component).

elecy

```
real(kind=dp) variables::elecy
```

Electric field acting on all charged particles (y-component).

elecz

```
real(kind=dp) variables::elecz
```

Electric field acting on all charged particles (z-component).

engsic

```
real(kind=dp) variables::engsic
```

Correction made to system potential energy resulting from application of Ewald sum on charges to remove self-interaction contributions.

engunit

```
integer variables::engunit
```

Energy units used for interaction parameters: 0 = absolute values given in FIELD file, 1 = values scaled by temperature given in CONTROL.

etype

```
integer variables::etype
```

Flag indicating electrostatic (smearing) model and application specified in CONTROL file: 0 = no electrostatics, 1 = Ewald sum with point charges (no charge smearing), 2 = Ewald sum with linear charge smearing, 3 = Ewald sum with exact Slater charge smearing, 4 = Ewald sum with approximate Slater charge smearing, 5 = Ewald sum with Gaussian charge smearing, 6 = Ewald sum with Gaussian charge smearing and no real-space terms, 7 = Ewald sum with sinusoidal charge smearing, 8 = SPME with point charges (no charge smearing), 9 = SPME with linear charge smearing, 10 = SPME with exact Slater charge smearing, 11 = SPME with approximate Slater charge smearing, 12 = SPME with Gaussian charge smearing, 13 = SPME with Gaussian charge smearing and no real-space terms, 14 = SPME with sinusoidal charge smearing.

ewprec

```
real(kind=dp) variables::ewprec
```

Relative precision (maximum relative error) in potential energy of Ewald sum or SPME calculations for electrostatics: either calculated from Ewald parameters or specified in CONTROL file as maximum value for parameterisation.

exportname

```
character(len=12) variables::exportname
```

Name of file (default: export) with a pre-existing configuration used to restart simulation.

fcfx

```
real(kind=dp), dimension (:), pointer variables::fcfx
```

Array pointer for x-component of corrective electrostatic forces on frozen particles.

fcfy

```
real(kind=dp), dimension (:), pointer variables::fcfy
```

Array pointer for y-component of corrective electrostatic forces on frozen particles.

fcfz

```
real(kind=dp), dimension (:), pointer variables::fcfz
```

Array pointer for z-component of corrective electrostatic forces on frozen particles.

flc

```
real(kind=dp), dimension (statsize) variables::flc
```

Current fluctuations (variances) of system properties obtained from properties over all timesteps after equilibration: 1 = total energy per particle, 2 = total potential energy per particle, 3 = total electrostatic energy per particle, 4 = total bond energy per particle, 5 = total angle energy per particle, 6 = total dihedral energy per particle, 7 = total virial per particle, 8 = total kinetic energy per particle, 9 = system pressure, 10 = system volume, 11 = surface tension in z-direction, 12 = system temperature, 13 = partial temperature in x-direction, 14 = partial temperature in y-direction, 15 = partial temperature in z-direction, 16-51 = pressure tensors separated out into conservative (potential), dissipative, random and kinetic contributions for all tensor components.

fpx

```
real(kind=dp) variables::fpx
```

x-component of force acting on Langevin barostat piston.

fpy

```
real(kind=dp) variables::fpy
```

y-component of force acting on Langevin barostat piston.

fpz

```
real(kind=dp) variables::fpz
```

z-component of force acting on Langevin barostat piston.

frzwdens

```
real(kind=dp) variables::frzwdens
```

Density of particles in frozen bead walls as specified in FIELD file, used to calculate numbers of particles in walls.

frzwspe

```
integer variables::frzwspe
```

Particle species of frozen bead walls selected in FIELD file.

frzwxwid

```
real(kind=dp) variables::frzwxwid
```

Width of both frozen bead walls orthogonal to x-axis, added to system volume during setup.

frzwywid

```
real(kind=dp) variables::frzwywid
```

Width of both frozen bead walls orthogonal to y-axis, added to system volume during setup.

frzwzwid

```
real(kind=dp) variables::frzwzwid
```

Width of both frozen bead walls orthogonal to z-axis, added to system volume during setup.

fvx

```
real(kind=dp), dimension (:), pointer variables::fvx
```

Array pointer for x-component of variable forces acting on particles.

fvy

```
real(kind=dp), dimension (:), pointer variables::fvy
```

Array pointer for y-component of variable forces acting on particles.

fvz

```
real(kind=dp), dimension (:), pointer variables::fvz
```

Array pointer for z-component of variable forces acting on particles.

fxfyfz

```
real(kind=dp), dimension (:,:), allocatable, target variables::fxfyfz
```

Forces acting on particles: first index of array is coordinate, second index is local particle number.

fxx

```
real(kind=dp), dimension (:), pointer variables::fxx
```

Array pointer for x-component of forces acting on particles.

fyy

```
real(kind=dp), dimension (:), pointer variables::fyy
```

Array pointer for y-component of forces acting on particles.

fzz

```
real(kind=dp), dimension (:), pointer variables::fzz
```

Array pointer for z-component of forces acting on particles.

gamma

```
real(kind=dp), dimension (:), allocatable variables::gamma
```

DPD dissipative force parameters γ or Lowe-Andersen collision frequencies Γ for particle species pairs.

gammaelec

```
real(kind=dp) variables::gammaelec
```

Permittivity coefficient used for electrostatic interactions, Γ (related to Bjerrum length).

historyname

```
character(len=12) variables::historyname
```

Name of file (default: HISTORY) used to write simulation trajectories.

idnode

```
integer variables::idnode
```

Number (rank) to identify current processor, set between 0 and the total number of processors less one. (Always equal to 0 for serial running.)

idx

```
integer variables::idx
```

Processor/subdomain x-coordinate with a value between 0 and the number of processors/subdomains in the x-direction less one. (Value is always 0 when running in serial.)

idy

```
integer variables::idy
```

Processor/subdomain y-coordinate with a value between 0 and the number of processors/subdomains in the y-direction less one. (Value is always 0 when running in serial.)

idz

```
integer variables::idz
```

Processor/subdomain z-coordinate with a value between 0 and the number of processors/subdomains in the z-direction less one. (Value is always 0 when running in serial.)

imcon

```
integer variables::imcon
```

Boundary condition key in CONFIG file: 0 = no boundaries, 1 = cubic, 2 = orthorhombic cuboid.

iscorr

```
integer variables::iscorr
```

Frequency (number of timesteps) for writing system properties to CORREL file.

itype

```
integer variables::itype
```

Flag indicating thermostat and integration type specified in CONTROL file: 0 = DPD with MD Velocity Verlet, 1 = DPD with DPD Velocity Verlet, 2 = DPD with first-order Shardlow splitting, 3 = DPD with second-order Shardlow splitting, 4 = Lowe-Andersen, 5 = Peters, 6 = Stoyanov-Groot.

ivrl

```
real(kind=dp), dimension (3) variables::ivrl
```

Instantaneous virial (separated into Cartesian directions) for application of barostats, excluding contributions from thermostat.

keytrj

```
integer variables::keytrj
```

Particle data level for writing to HISTORY file: 0 = positions, 1 = positions and velocities, 2 = positions, velocities and forces.

kmax1

```
integer variables::kmax1
```

Maximum number of periodic images used in x-direction for reciprocal space part of Ewald sum.

kmax2

```
integer variables::kmax2
```

Maximum number of periodic images used in y-direction for reciprocal space part of Ewald sum.

kmax3

```
integer variables::kmax3
```

Maximum number of periodic images used in z-direction for reciprocal space part of Ewald sum.

kres

```
integer variables::kres
```

Key for restarting simulation using export and/or REVIVE files: 0 = no restart, 1 = restart with configuration in export file and statistical accumulators in REVIVE file, 2 = start new simulation using configuration in export file, 3 = start new simulation using configuration in export file and rescale particle velocities to match system temperature).

ktype

```
integer, dimension (:), allocatable variables::ktype
```

Array with types of interaction potentials for available pairs of particle species: 0 = Lennard-Jones, 1 = Weeks-Chandler-Andersen, 2 = Groot-Warren 'standard DPD', 3 = Warren's two-parameter many-body DPD

kxyz

```
integer, dimension (:,:), allocatable variables::kxyz
```

List of available reciprocal space vectors within range of maximum vector in terms of periodic image duplications: first index is Cartesian coordinates for reciprocal vector (1, 2, 3) and flag for inclusion in SPME reciprocal space calculations (4), second index counts the available vectors.

l_scr

```
logical variables::l_scr
```

Switch to divert simulation outputs from OUTPUT file to standard output (screen).

lab

```
integer, dimension (:), allocatable, save variables::lab
```

Global numbers (unique identifiers) for particles.

langle

```
logical variables::langle
```

Switch to apply bond angle interactions within molecules.

lblclst

```
integer, dimension (:,:), allocatable, save variables::lblclst
```

Array of global and local numbers for all particles included in molecules for current processor/subdomain, sorted by global particle numbers and used as a look-up table to find local numbers for particles involved in bonds, angles and dihedrals: first index gives entry number, second index gives global (1) and local (2) particle numbers.

lbond

```
logical variables::lbond
```

Switch to apply bonded interactions within molecules.

lcell

```
integer, dimension (:), allocatable variables::lcell
```

Array for link cells used for pairwise interactions and thermostats indicating numbers of link cells within processor's subdomain (not including cells in boundary halo).

lcell_neighbour

```
integer, dimension (:), allocatable variables::lcell_neighbour
```

Array for link cells used for pairwise interactions and thermostats indicating numbers of neighbouring link cells for cells within processor's subdomain (set up in advance to find neighbouring cells quickly).

lcell_therm

```
integer, dimension (:), allocatable variables::lcell_therm
```

Switches for applying thermostats for particles in neighbouring interaction link cells (used to identify if neighbouring cell traverses Lees-Edwards shearing boundary).

lcellew

```
integer, dimension (:), allocatable variables::lcellew
```

Array for link cells used for short-range electrostatic interactions indicating numbers of link cells within processor's subdomain (not including cells in boundary halo).

lcellew_neighbour

```
integer, dimension (:), allocatable variables::lcellew_neighbour
```

Array for link cells used for short-range electrostatic interactions indicating numbers of neighbouring link cells for cells within processor's subdomain (set up in advance to find neighbouring cells quickly).

lcellmb

```
integer, dimension (:), allocatable variables::lcellmb
```

Array for link cells used for many-body DPD localised density calculations indicating numbers of link cells within processor's subdomain (not including cells in boundary halo).

lcellmb_neighbour

```
integer, dimension (:), allocatable variables::lcellmb_neighbour
```

Array for link cells used for many-body DPD localised density calculations indicating numbers of neighbouring link cells for cells within processor's subdomain (set up in advance to find neighbouring cells quickly).

lconfzero

```
logical variables::lconfzero
```

Switch to use bottom left back corner as origin (0,0,0) for CONFIG file, as used by default in previous versions of DL_MESO_DPD, instead of box centre.

lcorr

```
logical variables::lcorr
```

Switch to write system properties to CORREL file.

lct

```
integer, dimension (:), allocatable variables::lct
```

Array giving first particle (local) number for each linked-cell list used for pairwise interactions and thermostats.

lctew

```
integer, dimension (:), allocatable variables::lctew
```

Array giving first particle (local) number for each linked-cell list used for short-range electrostatic interactions.

lctmb

```
integer, dimension (:), allocatable variables::lctmb
```

Array giving first particle (local) number for each linked-cell list used for many-body DPD localised density calculations.

ldihed

```
logical variables::ldihed
```

Switch to apply bond dihedral interactions within molecules.

ldpol

```
logical variables::ldpol
```

Switch for applying charge dipole corrections for slab geometries.

ldyn

```
logical variables::ldyn
```

Switch for dynamic systems (i.e. if a flow field is included) to write additional information in OUTPUT and CORREL files.

levcfg

```
integer variables::levcfg
```

Particle data level in CONFIG file: 0 = positions, 1 = positions and velocities, 2 = positions, velocities and forces.

lfrzn

```
integer, dimension (:), allocatable variables::lfrzn
```

Array with switch indicating if all particles of given species are designated as frozen: 0 = not frozen, 1 = frozen.

lfrzwall

```
logical variables::lfrzwall
```

Switch indicating use of at least one set of frozen bead walls.

lfrzx

```
logical variables::lfrzx
```

Switch indicating use of frozen bead walls orthogonal to x-axis.

lfrzy

```
logical variables::lfrzy
```

Switch indicating use of frozen bead walls orthogonal to y-axis.

lfrzz

```
logical variables::lfrzz
```

Switch indicating use of frozen bead walls orthogonal to z-axis.

lgbnd

```
logical variables::lgbnd
```

Switch to apply bond interactions using a replicated data approach (globally specified book-keeping tables and gathering particle positions globally).

ligindex

```
logical variables::ligindex
```

Switch to ignore particle indices provided in CONFIG file and assign indices based on sequential order in file.

link

```
integer, dimension (:), allocatable variables::link
```

Array giving subsequent particle (local) number for each linked-cell list used for pairwise interactions and thermostats, based on previous particle numbers (with 0 indicating end of list).

linkew

```
integer, dimension (:), allocatable variables::linkew
```

Array giving subsequent particle (local) number for each linked-cell list used for short-range electrostatic interactions, based on previous particle numbers (with 0 indicating end of list).

linkmb

```
integer, dimension (:), allocatable variables::linkmb
```

Array giving subsequent particle (local) number for each linked-cell list used for many-body DPD localised density calculations, based on previous particle numbers (with 0 indicating end of list).

lisoprs

```
logical variables::lisoprs
```

Switch for barostat to apply system pressure (semi-)isotropically.

lmb

```
logical variables::lmb
```

Switch for local density calculations used for many-body DPD interactions.

lmp

```
integer, dimension (:), allocatable, save variables::lmp
```

Numbers of processors currently owning particles.

lnewsmall

```
logical variables::lnewsmall
```

Switch to indicate a single electrostatic link cell in any direction (for small systems) and impose more stringent conditions to find interacting particle pairs without duplications.

lnfold

```
logical variables::lnfold
```

Switch for duplicating system presented in FIELD and CONFIG files ('nfold' option).

lnmbsmall

```
logical variables::lnmbsmall
```

Switch to indicate a single many-body DPD link cell in any direction (for small systems) and impose more stringent conditions to find interacting particle pairs without duplications.

lnsmall

```
logical variables::lnsmall
```

Switch to indicate a single interaction link cell in any direction (for small systems) and impose more stringent conditions to find interacting particle pairs without duplications.

loc

```
integer, dimension (:), allocatable, save variables::loc
```

Local numbers for particles on processors currently owning them.

lompcrit

```
logical variables::lompcrit
```

Switch to use critical regions for OpenMP multithreading and reduce memory usage for assigning forces to particles.

lstrs

```
logical variables::lstrs
```

Switch to write any separated stress tensors to Stress_*.d files.

lstrss

```
logical, dimension (4) variables::lstrss
```

Switches to determine which separated stress tensors (conservative/potential, dissipative, random, kinetic) to write to Stress_*.d files.

lsurf

```
real(kind=dp), dimension (:), allocatable variables::lsurf
```

Array with switch indicating if numbered pair of particle species contributes to surface energy (e.g. if one particle species is frozen and contained in a frozen bead wall).

ltemp

```
logical variables::ltemp
```

Switch to rescale particle velocities during equilibration to match specified temperature.

ltm

```
integer, dimension (:), allocatable, save variables::ltm
```

Molecule types (numbers) for particles.

ltp

```
integer, dimension (:), allocatable, save variables::ltp
```

Species (numbers) for particles.

ltraj

```
logical variables::ltraj
```

Switch to write simulation trajectories to HISTORY file.

lvarfc

```
logical variables::lvarfc
```

Switch to determine if second set of forces needs to be allocated in memory (for use with DPD Velocity Verlet or Stoyanov-Groot thermostat).

map

```
integer, dimension(6) variables::map
```

Array specifying neighbouring processors for current processor in negative x-direction (1), positive x-direction (2), negative y-direction (3), positive y-direction (4), negative z-direction (5) and positive z-direction (6). (If only one processor in a particular direction, the corresponding array indices points to the current processor: this is the case for all directions when running in serial.)

maxbfbd

```
integer variables::maxbfbd
```

Maximum number of particles found in each processor's boundary halo: initially estimated from assumed constant particle density and size of boundary halo, readjusted during equilibration and for many-body DPD calculations, used to define sizes of arrays for inter-processor communications (primarily export of data to boundary halos).

maxbuf

```
integer variables::maxbuf
```

Maximum number of values each processor will send or receive during inter-processor communications: calculated from maximum number of particles in boundary halo or maximum reciprocal space vectors (whichever is larger).

maxdim

```
integer variables::maxdim
```

Maximum number of particles each processor can hold: initially estimated from number of particles in system and assumption of equal division among processors, used to define sizes of arrays for particle positions, velocities and forces.

maxpair

```
integer variables::maxpair
```

Maximum number of possible particle pairs: estimated from number of particles in system, maximum thermostat cutoff distance and thermostat parameters, used to define sizes of arrays holding lists of particle pairs for non-DPD thermostats (e.g. Lowe-Andersen).

mlstrtspe

```
integer, dimension (:,:), allocatable, save variables::mlstrtspe
```

Array with species of each particle for each molecule type as given in FIELD file: first index is specified molecule type, second index is particle number in molecule (up to maximum number of particles per molecule).

mlstrtxxx

```
real(kind=dp), dimension (:,:), allocatable, save variables::mlstrtxxx
```

x-coordinate of position for each particle in a given type of molecule (as specified in FIELD file) relative to the molecule centre of mass: first index is molecule type, second index is (relative) particle number in molecule.

mlstrtyyy

```
real(kind=dp), dimension (:,:), allocatable, save variables::mlstrtyyy
```

y-coordinate of position for each particle in a given type of molecule (as specified in FIELD file) relative to the molecule centre of mass: first index is molecule type, second index is (relative) particle number in molecule.

mlstrtzzz

```
real(kind=dp), dimension (:,:), allocatable, save variables::mlstrtzzz
```

z-coordinate of position for each particle in a given type of molecule (as specified in FIELD file) relative to the molecule centre of mass: first index is molecule type, second index is (relative) particle number in molecule.

mlszx

```
real(kind=dp), dimension (:), allocatable, save variables::mlszx
```

Maximum absolute x-coordinate for each molecule type as given in FIELD file, used to insert molecules into system when starting simulation without initial or restart configurations.

mlszy

```
real(kind=dp), dimension (:), allocatable, save variables::mlszy
```

Maximum absolute y-coordinate for each molecule type as given in FIELD file, used to insert molecules into system when starting simulation without initial or restart configurations.

mlszz

```
real(kind=dp), dimension (:), allocatable, save variables::mlszz
```

Maximum absolute z-coordinate for each molecule type as given in FIELD file, used to insert molecules into system when starting simulation without initial or restart configurations.

moliso

```
logical, dimension (:), allocatable variables::moliso
```

Switches indicating if isomers for each molecule type are permitted when setting up simulations from scratch (without initial or restart configurations).

molnam

```
character(len=8), dimension (:), allocatable, save variables::molnam
```

Molecule names (up to 8 characters) for particles.

molstart

```
integer, dimension (:), allocatable, save variables::molstart
```

Array with first global particle numbers for each molecule in system, used to add molecules to system when starting simulation without initial or restart configurations and to set up bond, angle and dihedral tables.

mxangles

```
integer variables::mxangles
```

Maximum number of angles in a molecule based on FIELD file contents: used to define sizes of arrays used to create angle tables during simulation setup.

mxbonds

```
integer variables::mxbonds
```

Maximum number of bonds in a molecule based on FIELD file contents: used to define sizes of arrays used to create bond tables during simulation setup.

mxdiheds

```
integer variables::mxdiheds
```

Maximum number of dihedrals in a molecule based on FIELD file contents: used to define sizes of arrays used to create dihedral tables during simulation setup.

mxmolsize

```
integer variables::mxmolsize
```

Maximum number of particles in a molecule based on FIELD file contents: used to define sizes of arrays used to assign molecular data during simulation setup.

mxpcell

```
integer variables::mxpcell
```

Maximum number of particles per interaction link cell, used to determine sizes of arrays for lists of interacting particle pairs.

mxprm

```
integer variables::mxprm
```

Largest number of parameters needed to define the interaction types specified in FIELD file.

mxspl

```
integer variables::mxspl
```

Number of points in each direction to interpolate charges onto grid using B-splines for Smooth Particle Mesh Ewald (SPME) calculation.

mxsprm

```
integer variables::mxsprm
```

Largest number of parameters needed to define the surfaceinteraction types specified in FIELD file.

nammol

```
character(len=8), dimension (:), allocatable, save variables::nammol
```

Names (up to 8 characters) for types of molecules as defined in FIELD.

namspe

```
character(len=8), dimension (:), allocatable, save variables::namspe
```

Names (up to 8 characters) for particle species as defined in FIELD.

nangdef

```
integer variables::nangdef
```

Total number of defined types of angle based on their functional forms and parameters, as given in FIELD file.

nangle

```
integer, dimension (:), allocatable, save variables::nangle
```

Total numbers of angle for each molecule type as defined in FIELD file

nangles

```
integer variables::nangles
```

Total number of entries in angle book-keeping table (angtbl) for current processor/subdomain.

nav

```
integer variables::nav
```

Current number of timesteps after equilibration, used to weight statistical properties when calculating time-averaged values and fluctuations.

nbdmol

```
integer, dimension (:), allocatable variables::nbdmol
```

Number of particles per molecule for each molecule type.

nbeads

```
integer variables::nbeads
```

Total number of particles in subdomain of current processor, excluding those in boundary halo.

nbond

```
integer, dimension (:), allocatable, save variables::nbond
```

Total numbers of bonds for each molecule type as defined in FIELD file.

nbonddef

```
integer variables::nbonddef
```

Total number of defined types of bond based on their functional forms and parameters, as given in FIELD file.

nbonds

```
integer variables::nbonds
```

Total number of entries in bond book-keeping table (bndtbl) for current processor/subdomain.

ndhddef

```
integer variables::ndhddef
```

Total number of defined types of dihedral based on their functional forms and parameters, as given in FIELD file.

ndihed

```
integer, dimension (:), allocatable, save variables::ndihed
```

Total numbers of dihedrals for each molecule type as defined in FIELD file.

ndiheds

```
integer variables::ndiheds
```

Total number of entries in dihedral book-keeping table (dhdtbl) for current processor/subdomain.

ndump

```
integer variables::ndump
```

Frequency for creating simulation restart data in export and REVIVE files (default: 1000).

nfbeads

```
integer variables::nfbeads
```

Total number of frozen particles in subdomain of current processor, excluding those in boundary halo.

nfold

```
integer variables::nfold
```

Total number of duplications of system given in CONFIG file when using 'nfold' option.

nfoldx

```
integer variables::nfoldx
```

Integer number of duplications in x-direction for system given in CONFIG file when using *nfold* option.

nfoldy

```
integer variables::nfoldy
```

Integer number of duplications in y-direction for system given in CONFIG file when using ‘nfold’ option.

nfoldz

```
integer variables::nfoldz
```

Integer number of duplications in z-direction for system given in CONFIG file when using ‘nfold’ option.

nfsyst

```
integer variables::nfsyst
```

Total number of frozen particles in simulation (can include particles in molecules if these specify particle species that are frozen).

nfsystcell

```
integer variables::nfsystcell
```

Total number of frozen particles in unit cell defined by FIELD and CONFIG files.

nfwsyst

```
integer variables::nfwsyst
```

Total number of particles in frozen bead walls, calculated from wall selections in CONTROL file, system volume (in CONTROL or CONFIG), wall density and thickness in FIELD.

nlewx

```
integer variables::nlewx
```

Total number of link cells in x-direction used for short-range electrostatic interactions in current subdomain (excluding boundary halo).

nlewx2

```
integer variables::nlewx2
```

Total number of link cells in x-direction used for short-range electrostatic interactions in current subdomain and boundary halo.

nlewy

```
integer variables::nlewy
```

Total number of link cells in y-direction used for short-range electrostatic interactions in current subdomain (excluding boundary halo).

nlewy2

```
integer variables::nlewy2
```

Total number of link cells in y-direction used for short-range electrostatic interactions in current subdomain and boundary halo.

nlewz

```
integer variables::nlewz
```

Total number of link cells in z-direction used for short-range electrostatic interactions in current subdomain (excluding boundary halo).

nlewz2

```
integer variables::nlewz2
```

Total number of link cells in z-direction used for short-range electrostatic interactions in current subdomain and boundary halo.

nlist

```
integer variables::nlist
```

Total number of entries in global/local particle number look-up array.

nlistew

```
integer variables::nlistew
```

Total number of reciprocal space vectors to work through when calculating Ewald sum (size of reciprocal space vector lists).

nlmbx

```
integer variables::nlmbx
```

Total number of link cells in x-direction used for many-body DPD localised density calculations in current subdomain (excluding boundary halo).

nlmbx2

```
integer variables::nlmbx2
```

Total number of link cells in x-direction used for many-body DPD localised density calculations in current subdomain and boundary halo.

nlmby

```
integer variables::nlmby
```

Total number of link cells in y-direction used for many-body DPD localised density calculations in current subdomain (excluding boundary halo).

nlmby2

```
integer variables::nlmby2
```

Total number of link cells in y-direction used for many-body DPD localised density calculations in current subdomain and boundary halo.

nlmbz

```
integer variables::nlmbz
```

Total number of link cells in z-direction used for many-body DPD localised density calculations in current subdomain (excluding boundary halo).

nlmbz2

```
integer variables::nlmbz2
```

Total number of link cells in z-direction used for many-body DPD localised density calculations in current subdomain and boundary halo.

nlx

```
integer variables::nlx
```

Total number of link cells in x-direction used for pairwise interactions and thermostats in current subdomain (excluding boundary halo).

nlx2

```
integer variables::nlx2
```

Total number of link cells in x-direction used for pairwise interactions and thermostats in current subdomain and boundary halo.

nly

```
integer variables::nly
```

Total number of link cells in y-direction used for pairwise interactions and thermostats in current subdomain (excluding boundary halo).

nly2

```
integer variables::nly2
```

Total number of link cells in y-direction used for pairwise interactions and thermostats in current subdomain and boundary halo.

nlz

```
integer variables::nlz
```

Total number of link cells in z-direction used for pairwise interactions and thermostats in current subdomain (excluding boundary halo).

nlz2

```
integer variables::nlz2
```

Total number of link cells in z-direction used for pairwise interactions and thermostats in current subdomain and boundary halo.

nmol

```
integer, dimension (:), allocatable variables::nmol
```

Total number of molecules in system for each molecule type.

nmoldef

```
integer variables::nmoldef
```

Total number of defined molecule types specified in FIELD file.

nodes

```
integer variables::nodes
```

Total number of available processors for current calculation. (Always equal to 1 for serial running.)

npair

```
integer variables::npair
```

Number of particle pairs in list (for current processor/subdomain) for alternative pairwise thermostats.

npot

```
integer variables::npot
```

Total number of possible pairs of particle species and pairwise interactions, calculated from number of particle species.

npx

```
integer variables::npx
```

Total number of processors and subdomains (equal divisions of simulation box) in x-direction. (Value is always 1 when running in serial.)

npxfwx

```
integer variables::npxfwx
```

Number of frozen particles in x-direction for frozen wall orthogonal to x-axis.

npxfwy

```
integer variables::npxfwy
```

Number of frozen particles in y-direction for frozen wall orthogonal to x-axis.

npxfwz

```
integer variables::npxfwz
```

Number of frozen particles in z-direction for frozen wall orthogonal to x-axis.

npy

```
integer variables::npy
```

Total number of processors and subdomains (equal divisions of simulation box) in y-direction. (Value is always 1 when running in serial.)

npyfwx

```
integer variables::npyfwx
```

Number of frozen particles in x-direction for frozen wall orthogonal to y-axis.

npyfwy

```
integer variables::npyfwy
```

Number of frozen particles in y-direction for frozen wall orthogonal to y-axis.

npyfwz

```
integer variables::npyfwz
```

Number of frozen particles in z-direction for frozen wall orthogonal to y-axis.

npz

```
integer variables::npz
```

Total number of processors and subdomains (equal divisions of simulation box) in z-direction. (Value is always 1 when running in serial.)

npzfwx

```
integer variables::npzfwx
```

Number of frozen particles in x-direction for frozen wall orthogonal to z-axis.

npzfwy

```
integer variables::npzfwy
```

Number of frozen particles in y-direction for frozen wall orthogonal to z-axis.

npzfwz

```
integer variables::npzfwz
```

Number of frozen particles in z-direction for frozen wall orthogonal to z-axis.

nrun

```
integer variables::nrun
```

Total number of timesteps requested in CONTROL file for DPD simulation.

nsbpo

```
integer variables::nsbpo
```

Frequency (number of timesteps) for printing simulation outputs to OUTPUT file.

nsbts

```
integer variables::nsbts
```

Frequency (number of timesteps) for rescaling particle velocities to match required temperature during equilibration.

nseql

```
integer variables::nseql
```

Number of timesteps for system equilibration (including temperature rescaling, excluding statistical accumulation of properties).

nshrs

```
integer variables::nshrs
```

Starting timestep for applying Lees-Edwards shearing boundary (change in particle velocity, displacement of particles crossing boundary).

nspe

```
integer variables::nspe
```

Total number of particle species as specified in FIELD file.

nspec

```
integer, dimension (:), allocatable variables::nspec
```

Total numbers of particles for each species not included in any molecules.

nspecmol

```
integer, dimension (:), allocatable variables::nspecmol
```

Total numbers of particles for each species included in all molecules.

nstep

```
integer variables::nstep
```

Number of current simulation timestep (should be no larger than total number of timesteps for simulation to continue).

nstk

```
integer variables::nstk
```

Number of timesteps to store system properties for calculating rolling averages.

nstrs

```
integer variables::nstrs
```

Frequency (number of timesteps) for writing separated stress tensors to Stress_*.d files.

nsyst

```
integer variables::nsyst
```

Total number of particles in simulation, including frozen particles and those contained in defined molecules.

nsystcell

```
integer variables::nsystcell
```

Total number of particles in unit cell defined by FIELD and CONFIG files.

ntraj

```
integer variables::ntraj
```

Frequency (number of timesteps) for writing simulation trajectories to HISTORY file.

numang

```
integer variables::numang
```

Total number of angles for all molecules in simulation.

numangcell

```
integer variables::numangcell
```

Total number of angles for all molecules in unit cell defined by FIELD file.

numbond

```
integer variables::numbond
```

Total number of bonds for all molecules in simulation.

numbondcell

```
integer variables::numbondcell
```

Total number of bonds for all molecules in unit cell defined by FIELD file.

numdhd

```
integer variables::numdhd
```

Total number of dihedrals for all molecules in simulation.

numdhdcell

```
integer variables::numdhdcell
```

Total number of dihedrals for all molecules in unit cell defined by FIELD file.

nummol

```
integer variables::nummol
```

Total number of molecules (of all defined types) in simulation.

nummolcell

```
integer variables::nummolcell
```

Total number of molecules (of all defined types) in unit cell defined by FIELD file.

nusyst

```
integer variables::nusyst
```

Total number of particles in simulation not contained in defined molecules (can include frozen particles).

nusystcell

```
integer variables::nusystcell
```

Total number of particles not contained in defined molecules in unit cell defined by FIELD and CONFIG files.

outsel

```
integer variables::outsel
```

System property key for printing to OUTPUT file, selecting which/how many properties are written.

pe

```
real(kind=dp) variables::pe
```

Total potential energy for system resulting from all interactions (except thermostat).

plbound

```
integer, dimension (:), allocatable, save variables::plbound
```

Boundary thermostatting key of selected pairs for alternative pairwise thermostats, used to determine if particle pair crosses Lees-Edwards shearing boundary and requires adjustment of relative velocity.

pldxyz

```
real(kind=dp), dimension (:), allocatable, save variables::pldxyz
```

Vectors between selected particle pairs for alternative pairwise thermostat (i.e. Lowe-Andersen, Peters, Stoyanov-Groot), given as triples with x-, y- and z-components contiguous in memory.

plintij

```
integer, dimension (:), allocatable, save variables::plintij
```

Particle interaction type (pair of species) for selected pairs used for Peters thermostat.

plparti

```
integer, dimension (:), allocatable, save variables::plparti
```

Local particle indices for first particles in selected pairs for alternative pairwise thermostats.

plpartj

```
integer, dimension (:), allocatable, save variables::plpartj
```

Local particle indices for second particles in selected pairs for alternative pairwise thermostats.

plproci

```
integer, dimension (:), allocatable, save variables::plproci
```

Processor/subdomain number for first particles in selected pairs for alternative pairwise thermostats.

plprocj

```
integer, dimension (:), allocatable, save variables::plprocj
```

Processor/subdomain number for second particles in selected pairs for alternative pairwise thermostats.

potcfz

```
real(kind=dp) variables::potcfz
```

Potential energy contribution used to corrective overall potential energy due to corrective electrostatic forces on frozen particles for reciprocal-space Ewald sum calculations.

prszero

```
real(kind=dp) variables::prszero
```

Target system or normal pressure for barostat specified in CONTROL file.

psmass

```
real(kind=dp) variables::psmass
```

Mass of Langevin barostat piston, W_g.

qchg

```
real(kind=dp) variables::qchg
```

Net charge on system (used to calculate corrections for Ewald sum).

qfixv

real(kind=dp) variables::qfixv

Correction made to system potential energy resulting from application of Ewald sum for systems with net overall charge.

ralphaew

```
real(kind=dp) variables::ralphaew
```

Reciprocal of real-space convergence coefficient for electrostatic interactions with Ewald sums.

rav

```
real(kind=dp), dimension (stksize) variables::rav
```

Current rolling average values of system properties obtained from properties in statistical stacks: 1 = total energy per particle, 2 = total potential energy per particle, 3 = total electrostatic energy per particle, 4 = total bond energy per particle, 5 = total angle energy per particle, 6 = total dihedral energy per particle, 7 = total virial per particle, 8 = total kinetic energy per particle, 9 = system pressure, 10 = system volume, 11 = surface tension in z-direction, 12 = system temperature, 13 = partial temperature in x-direction, 14 = partial temperature in y-direction, 15 = partial temperature in z-direction.

rct2

```
real(kind=dp) variables::rct2
```

Square of maximum cutoff distance for pairwise interactions (excluding many-body DPD and Ewald real-space).

rcut

```
real(kind=dp) variables::rcut
```

Maximum cutoff distance for pairwise interactions (other than many-body DPD localised density calculations and Ewald sum real-space cutoff), either specified in CONTROL or derived from interaction parameters in FIELD.

rel2

```
real(kind=dp) variables::rel2
```

Square of cutoff distance for short-range electrostatic interactions used for real-space Ewald sum contributions.

relec

```
real(kind=dp) variables::relec
```

Cutoff distance for short-range electrostatic interactions used for real-space Ewald sum contributions, r_e.

rhalo

```
real(kind=dp) variables::rhalo
```

Size of boundary halo specified in CONTROL file.

rhalox1

```
real(kind=dp) variables::rhalox1
```

Size of boundary halo in negative x-direction, derived from value specified in CONTROL.

rhalox2

```
real(kind=dp) variables::rhalox2
```

Size of boundary halo in positive x-direction, derived from value specified in CONTROL.

rhaloy1

```
real(kind=dp) variables::rhaloy1
```

Size of boundary halo in negative y-direction, derived from value specified in CONTROL.

rhaloy2

```
real(kind=dp) variables::rhaloy2
```

Size of boundary halo in positive y-direction, derived from value specified in CONTROL.

rhaloz1

```
real(kind=dp) variables::rhaloz1
```

Size of boundary halo in negative z-direction, derived from value specified in CONTROL.

rhaloz2

```
real(kind=dp) variables::rhaloz2
```

Size of boundary halo in positive z-direction, derived from value specified in CONTROL.

rhomb

```
real(kind=dp), dimension (:, :), allocatable variables::rhomb
```

Localised densities for many-body DPD interactions: first index is local particle number, second index is particle species.

rkxyz

```
real(kind=dp), dimension (:,:), allocatable variables::rkxyz
```

List of reciprocal space vectors within range of the maximum vector along with associated parameters: first index is Cartesian coordinates for reciprocal vector (1, 2, 3) or multipliers for reciprocal space calculations (4, 5), second index counts the available vectors.

rmbct2

```
real(kind=dp) variables::rmbct2
```

Square of cutoff distance for localised density calculations used for many-body DPD interactions.

rmbcut

```
real(kind=dp) variables::rmbcut
```

Cutoff distance for localised density calculations used for many-body DPD interactions, r_d.

rndseed

```
integer variables::rndseed
```

Seed for setup random number generators.

rpsmass

```
real(kind=dp) variables::rpsmass
```

Reciprocal of mass of Langevin barostat piston

rrct2

```
real(kind=dp) variables::rrct2
```

Square of the reciprocal of maximum cutoff distance for pairwise interactions (excluding many-body DPD and Ewald real-space).

rrmbcut

```
real(kind=dp) variables::rrmbcut
```

Reciprocal of cutoff distance for localised density calculations used for many-body DPD interactions.

rrtcut

```
real(kind=dp) variables::rrtcut
```

Reicprocal of cutoff distance for pairwise thermostat.

rtct2

```
real(kind=dp) variables::rtct2
```

Square of cutoff distance for pairwise thermostat

rtcut

```
real(kind=dp) variables::rtcut
```

Cutoff distance for pairwise thermostat (DPD or alternatives), r_c.

rtstep

```
real(kind=dp) variables::rtstep
```

Reciprocal of timestep size, $\frac{1}{\Delta t}$ (used for simulation setup and non-DPD thermostat calculations).

se

```
real(kind=dp) variables::se
```

Total potential energy for system resulting from surface interactions.

shrdx

```
real(kind=dp) variables::shrdx
```

x-component of boundary displacement at Lees-Edwards boundary along right/top/front box surface (negative value along left/bottom/back surface).

shrdy

```
real(kind=dp) variables::shrdy
```

y-component of boundary displacement at Lees-Edwards boundary along right/top/front box surface (negative value along left/bottom/back surface).

shrdz

```
real(kind=dp) variables::shrdz
```

z-component of boundary displacement at Lees-Edwards boundary along right/top/front box surface (negative value along left/bottom/back surface).

shrvx

```
real(kind=dp) variables::shrvx
```

x-component of shearing velocity at Lees-Edwards boundary on right/top/front box surface (negative value at left/bottom/back surface).

shrvy

```
real(kind=dp) variables::shrvy
```

y-component of shearing velocity at Lees-Edwards boundary on right/top/front box surface (negative value at left/bottom/back surface).

shrvz

```
real(kind=dp) variables::shrvz
```

z-component of shearing velocity at Lees-Edwards boundary on right/top/front box surface (negative value at left/bottom/back surface).

sidex

```
real(kind=dp) variables::sidex
```

Size of current processor's subdomain in x-direction.

sidey

```
real(kind=dp) variables::sidey
```

Size of current processor's subdomain in y-direction.

sidez

```
real(kind=dp) variables::sidez
```

Size of current processor's subdomain in z-direction.

sigma

```
real(kind=dp), dimension (:), allocatable variables::sigma
```

DPD random force parameters $\frac{\sigma}{\sqrt{\Delta t}}$ or Lowe-Andersen particle pair probabilities $\Gamma\Delta t$ for particle species pairs.

sigmalang

```
real(kind=dp) variables::sigmalang
```

Langevin barostat random parameter, σ_p.

srfktype

```
integer, dimension (:), allocatable variables::srfktype
```

Array with types of surface interaction potentials for particle species: 0 = Groot-Warren 'standard DPD', 1 = Weeks-Chandler-Andersen.

srflgc

```
logical, dimension(6) variables::srflgc
```

Flags to determine if surfaces exist in current processor/subdomain (indices: 1 = -x, 2 = +x, 3 = -y, 4 = +y, 5 = -z, 6 = +z).

srfpos

```
real(kind=dp) variables::srfpos
```

Distance between the specified boundaries of the simulation box and location of reflecting wall surfaces, specified in CONTROL file (default value of zero).

srftype

```
integer variables::srftype
```

Flag indicating surface type specified in CONTROL file: 0 = no surfaces, 1 = Lees-Edwards periodic shearing boundaries, 2 = hard walls with specular reflections, 3 = hard walls with bounce-back reflections.

srfx

```
integer variables::srfx
```

Flag indicating surfaces orthogonal to x-axis of simulation box: 0 = no surface, 1 = surface.

srfxps1

```
real(kind=dp) variables::srfxps1
```

x-coordinate of reflecting wall on left surface of box (orthogonal to negative x-axis) for current processor/subdomain.

srfxps2

```
real(kind=dp) variables::srfxps2
```

x-coordinate of reflecting wall on right surface of box (orthogonal to positive x-axis) for current processor/subdomain.

srfy

```
integer variables::srfy
```

Flag indicating surfaces orthogonal to y-axis of simulation box: 0 = no surface, 1 = surface.

srfyps1

```
real(kind=dp) variables::srfyps1
```

y-coordinate of reflecting wall on bottom surface of box (orthogonal to negative y-axis) for current processor/subdomain.

srfyps2

```
real(kind=dp) variables::srfyps2
```

y-coordinate of reflecting wall on top surface of box (orthogonal to positive y-axis) for current processor/subdomain.

srfz

```
integer variables::srfz
```

Flag indicating surfaces orthogonal to z-axis of simulation box: 0 = no surface, 1 = surface.

srfzct2

```
real(kind=dp) variables::srfzct2
```

Square of cutoff distance for interactions between particles and hard surfaces.

srfzcut

```
real(kind=dp) variables::srfzcut
```

Cutoff distance for interactions between particles and hard surfaces, z_c.

srfzps1

```
real(kind=dp) variables::srfzps1
```

z-coordinate of reflecting wall on back surface of box (orthogonal to negative z-axis) for current processor/subdomain.

srfzps2

```
real(kind=dp) variables::srfzps2
```

z-coordinate of reflecting wall on front surface of box (orthogonal to positive z-axis) for current processor/subdomain.

sstrs

```
integer variables::sstrs
```

Starting timestep to write separated stress tensors to Stress_*.d files.

stkae

```
real(kind=dp), dimension (:), allocatable variables::stkae
```

Array of collected angle energy per particle values (size given in CONTROL file), used as a statistical stack to calculate rolling average value.

stkbe

```
real(kind=dp), dimension (:), allocatable variables::stkbe
```

Array of collected bond energy per particle values (size given in CONTROL file), used as a statistical stack to calculate rolling average value.

stkde

```
real(kind=dp), dimension (:), allocatable variables::stkde
```

Array of collected dihedral energy per particle values (size given in CONTROL file), used as a statistical stack to calculate rolling average value.

stkee

```
real(kind=dp), dimension (:), allocatable variables::stkee
```

Array of collected electrostatic energy per particle values (size given in CONTROL file), used as a statistical stack to calculate rolling average value.

stkpe

```
real(kind=dp), dimension (:), allocatable variables::stkpe
```

Array of collected potential energy per particle values (size given in CONTROL file), used as a statistical stack to calculate rolling average value.

stkse

```
real(kind=dp), dimension (:), allocatable variables::stkse
```

Array of collected surface energy per particle values (size given in CONTROL file), used as a statistical stack to calculate rolling average value.

stktkex

```
real(kind=dp), dimension (:), allocatable variables::stktkex
```

Array of collected x-component kinetic energy per particle values (size given in CONTROL file), used as a statistical stack to calculate rolling average value.

stktkey

```
real(kind=dp), dimension (:), allocatable variables::stktkey
```

Array of collected y-component kinetic energy per particle values (size given in CONTROL file), used as a statistical stack to calculate rolling average value.

stktkez

```
real(kind=dp), dimension (:), allocatable variables::stktkez
```

Array of collected z-component kinetic energy per particle values (size given in CONTROL file), used as a statistical stack to calculate rolling average value.

stkvir

```
real(kind=dp), dimension (:), allocatable variables::stkvir
```

Array of collected virial per particle values (size given in CONTROL file), used as a statistical stack to calculate rolling average value.

stkvlm

```
real(kind=dp), dimension (:), allocatable variables::stkvlm
```

Array of collected system volume values (size given in CONTROL file), used as a statistical stack to calculate rolling average value.

stkzts

```
real(kind=dp), dimension (:), allocatable variables::stkzts
```

Array of collected surface tension in z-direction values (size given in CONTROL file), used as a statistical stack to calculate rolling average value.

stpae

```
real(kind=dp) variables::stpae
```

Total potential energy per particle resulting from all angle interactions at current timestep.

stpang

```
real(kind=dp) variables::stpang
```

Mean angle between pairs of bonds in molecules at current timestep.

stpbdl

```
real(kind=dp) variables::stpbdl
```

Mean length of bonds between pairs of particles in molecules at current timestep.

stpbdmn

```
real(kind=dp) variables::stpbdmn
```

Minimum length of bonds between pairs of particles in molecules at current timestep.

stpbdmx

```
real(kind=dp) variables::stpbdmx
```

Maximum length of bonds between pairs of particles in molecules at current timestep.

stpbe

```
real(kind=dp) variables::stpbe
```

Total potential energy per particle resulting from all bond interactions at current timestep.

stpde

```
real(kind=dp) variables::stpde
```

Total potential energy per particle resulting from all dihedral interactions at current timestep.

stpdhd

```
real(kind=dp) variables::stpdhd
```

Mean dihedral between pairs of bond planes in molecules at current timestep.

stpee

```
real(kind=dp) variables::stpee
```

Total potential energy per particle resulting from all electrostatic interactions at current timestep.

stppe

```
real(kind=dp) variables::stppe
```

Total potential energy per particle resulting from all interactions (excluding thermostat) at current timestep.

stpprs

```
real(kind=dp) variables::stpprs
```

Pressure of system at current timestep.

stpse

```
real(kind=dp) variables::stpse
```

Total potential energy per particle resulting from all surface (wall) interactions at current timestep.

stpte

```
real(kind=dp) variables::stpte
```

Total energy per particle resulting from all interactions (excluding thermostat) and motion at current timestep.

stptke

```
real(kind=dp) variables::stptke
```

Total kinetic energy per particle resulting from particle motion at current timestep.

stptpx

```
real(kind=dp) variables::stptpx
```

Partial kinetic temperature of system in x-direction at current timestep.

stptpy

```
real(kind=dp) variables::stptpy
```

Partial kinetic temperature of system in y-direction at current timestep.

stptpz

```
real(kind=dp) variables::stptpz
```

Partial kinetic temperature of system in z-direction at current timestep.

stpttp

```
real(kind=dp) variables::stpttp
```

Kinetic temperature of system at current timestep.

stpvir

```
real(kind=dp) variables::stpvir
```

Total virial per particle resulting from all interactions (including thermostat) at current timestep.

stpvlm

```
real(kind=dp) variables::stpvlm
```

Volume of system at current timestep.

stpzts

```
real(kind=dp) variables::stpzts
```

Surface tension of system in z-direction at current timestep.

straj

```
integer variables::straj
```

Starting timestep to write simulation trajectories to HISTORY file.

strcfz

```
real(kind=dp), dimension (36) variables::strcfz
```

Stress tensor contributions used to correct overall stress tensors (separated into potential, dissipative, random and kinetic terms) due to corrective electrostatic forces on frozen particles for reciprocal-space Ewald sum calculations.

stress

```
real(kind=dp), dimension (36) variables::stress
```

Total stress tensor for system resulting from particle interactions and motion, separated out into conservative (potential), dissipative, random and kinetic terms for all tensor components.

tclose

```
real(kind=dp) variables::tclose
```

Time allocated in seconds to close down DL_MESO_DPD calculation early, including writing and closing output and restart files.

temp

```
real(kind=dp) variables::temp
```

Specified system temperature in DPD units, $k_B T$.

text

```
character(len=80) variables::text
```

Contents of first line of CONTROL file used to name DPD simulation.

timfrc

```
real(kind=dp) variables::timfrc
```

Accumulator for time (in seconds) taken to calculate particle forces during simulation.

timjob

```
real(kind=dp) variables::timjob
```

Maximum available walltime in seconds for DPD calculation, including any time needed to write restart files when closing down calculation early.

timstp

```
real(kind=dp) variables::timstp
```

Accumulator for time (in seconds) taken to carry out all calculations per timestep during simulation.

tke

```
real(kind=dp), dimension (3) variables::tke
```

Total kinetic energy for system resulting from particle motion, separated out into (1) x-, (2) y- and (3) z-components.

tnum

```
integer variables::tnum
```

Total number of available OpenMP threads for calculation multithreading. (Always equal to 1 when using version without OpenMP.)

tnumuse

```
integer variables::tnumuse
```

Total number of OpenMP threads currently being used for calculation multithreading: this number can be set to less than the total number of available threads.

tstep

```
real(kind=dp) variables::tstep
```

Specified timestep size in DPD units, Δt.

upx

```
real(kind=dp) variables::upx
```

x-component of Langevin barostat piston velocity or Berendsen volume scaling factor for current timestep.

upy

```
real(kind=dp) variables::upy
```

y-component of Langevin barostat piston velocity or Berendsen volume scaling factor for current timestep.

upz

```
real(kind=dp) variables::upz
```

z-component of Langevin barostat piston velocity or Berendsen volume scaling factor for current timestep.

up1x

```
real(kind=dp) variables::up1x
```

x-component of Langevin barostat piston velocity at previous timestep.

up1y

```
real(kind=dp) variables::up1y
```

y-component of Langevin barostat piston velocity at previous timestep.

up1z

```
real(kind=dp) variables::up1z
```

z-component of Langevin barostat piston velocity at previous timestep.

vfxfyfz

```
real(kind=dp), dimension (:,:), allocatable, target variables::vfxfyfz
```

Variable forces acting on particles separated from others - dissipative forces (recalculated) for DPD Velocity Verlet or temperature-dependent forces calculated for Stoyanov-Groot thermostat: first index of array is coordinate, second index is local particle number.

vgapx

```
real(kind=dp) variables::vgapx
```

Vacuum gap (additional system volume) in x-direction for Ewald reciprocal space calculations of slab geometries.

vgapy

```
real(kind=dp) variables::vgapy
```

Vacuum gap (additional system volume) in y-direction for Ewald reciprocal space calculations of slab geometries.

vgapz

```
real(kind=dp) variables::vgapz
```

Vacuum gap (additional system volume) in z-direction for Ewald reciprocal space calculations of slab geometries.

vir

```
real(kind=dp) variables::vir
```

Total virial for system resulting from all interactions (including thermostat).

volm

```
real(kind=dp) variables::volm
```

Total volume of simulation box, specified in either CONTROL or from CONFIG file.

vrlcfz

```
real(kind=dp), dimension (3) variables::vrlcfz
```

Virial contributions (separated into Cartesian directions) used to correct overall virial due to corrective electrostatic forces on frozen particles for reciprocal-space Ewald sum calculations.

vvsrf

```
real(kind=dp), dimension (:, :), allocatable variables::vvsrf
```

Array with parameters (and preparatory values) needed to calculate surface interactions with particles: first index is parameter number, second index is particle species.

vvv

```
real(kind=dp), dimension (:,:), allocatable variables::vvv
```

Array with parameters (and preparatory values) needed to calculate interactions between particle pairs: first index is parameter number, second index is numbered pair of particle species.

vxvyvz

```
real(kind=dp), dimension (:,:), allocatable, target variables::vxvyvz
```

Velocities of particles: first index of array is coordinate, second index is local particle number.

vxx

```
real(kind=dp), dimension (:), pointer variables::vxx
```

Array pointer for x-component of particle velocities.

vyy

```
real(kind=dp), dimension (:), pointer variables::vyy
```

Array pointer for y-component of particle velocities.

vzz

```
real(kind=dp), dimension (:), pointer variables::vzz
```

Array pointer for z-component of particle velocities.

wdthewx

```
real(kind=dp) variables::wdthewx
```

Size of link cell for short-range electrostatic interactions in x-direction.

wdthewy

```
real(kind=dp) variables::wdthewy
```

Size of link cell for short-range electrostatic interactions in y-direction.

wdthewz

```
real(kind=dp) variables::wdthewz
```

Size of link cell for short-range electrostatic interactions in z-direction.

wdthmbx

```
real(kind=dp) variables::wdthmbx
```

Size of link cell for many-body DPD localised density calculations in x-direction.

wdthmby

```
real(kind=dp) variables::wdthmby
```

Size of link cell for many-body DPD localised density calculations in y-direction.

wdthmbz

```
real(kind=dp) variables::wdthmbz
```

Size of link cell for many-body DPD localised density calculations in z-direction.

wdthx

```
real(kind=dp) variables::wdthx
```

Size of link cell for pairwise interactions and thermostats in x-direction.

wdthy

```
real(kind=dp) variables::wdthy
```

Size of link cell for pairwise interactions and thermostats in y-direction.

wdthz

```
real(kind=dp) variables::wdthz
```

Size of link cell for pairwise interactions and thermostats in z-direction.

weight

```
real(kind=dp), dimension (:), allocatable, save variables::weight
```

Masses of particles.

xxx

```
real(kind=dp), dimension (:), pointer variables::xxx
```

Array pointer for x-component of particle positions.

xxyyzz

```
real(kind=dp), dimension (:,:), allocatable, target variables::xxyyzz
```

Positions of particles: first index of array is coordinate, second index is local particle number.

yyy

```
real(kind=dp), dimension (:), pointer variables::yyy
```

Array pointer for y-component of particle positions.

zum

```
real(kind=dp), dimension (11) variables::zum
```

Summed up values of properties in statistical stack arrays used to calculate rolling average values: 1 = potential energy per particle, 2 = electrostatic energy per particle, 3 = bond energy per particle, 4 = angle energy per particle, 5 = dihedral energy per particle, 6 = virial per particle, 7 = system volume, 8 = surface tension in z-direction, 9 = x-component of kinetic energy per particle, 10 = y-component of kinetic energy per particle, 11 = z-component of kinetic energy per particle.

zzz

```
real(kind=dp), dimension (:), pointer variables::zzz
```

Array pointer for z-component of particle positions.

10.4 numeric_container.F90

10.4.1 Summary

Module with general-purpose functions and subroutines required for DPD simulations, including random number generators, scalar sum functions, error functions, periodic image adjustment and Fast Fourier Transforms (FFT). (OpenMP multithreaded version available with numeric_container_omp.F90.)

10.4.2 Functions/Subroutines

- `real(kind=dp) function` *duni()*

  Creates a double precision random number between 0 and 1 using the Universal Random Number Generator.

- `real(kind=dp) function` *mtrnd()*

  Creates a double precision random number between 0 and 1 using the Mersenne Twister Random Number Generator.

- `real(kind=dp) function` *sarurnd()*

  Creates a double precision random number between 0 and 1 using the Saru Random Number Generator with three seeds.

- `real(kind=dp) function` *gaussmp()*

  Creates a double precision Gaussian random number with zero mean and unity variance using the Mersenne Twister random number generator and the Marsaglia polar method.

- `integer function` *idcube()*

  Calculates a node number based on input coordinates and extents.

- `real(kind=dp) function` *sclsum()*

  Calculates the scalar sum of an array.

- `real(kind=dp) function` *erfcdp()*

  Calculates the complementary error function of an input value.

- `real(kind=dp) function` *erfdp()*

  Calculates the error function of an input value.

- `subroutine` *images()*

  Calculates the minimum distance between two particles in a periodic orthogonal box.

- `subroutine` *quicksort_integer_indexed()*

  Sorts integers in a provided array in numerical order using quicksort while recording the original positions of its values.

- `recursive subroutine` *qsort_integer()*

  Sorts integers in array into numerical order using quicksort, recording the original positions of its values.

- `integer function` *bitadd()*

  Carries out bitwise addition of two integers.

- `integer function` *bitmult()*

  Carries out bitwise multiplication of two integers.

- `subroutine` *create_local_id_mol_map()*

  Creates a list of molecule numbers for each particle based on local particle index.

- `subroutine` *fft3d()*

  Carries out Fast Fourier Transform (FFT) on a three-dimensional complex array.

- `subroutine` *sfft()*

  Carries out Fast Fourier Transform (FFT) on complex array in place using Singleton's mixed-radix algorithm.

10.4.3 Variables

- `integer, parameter` *nuni*

  Size of array for Universal Random Number Generator state.

- `integer, parameter` *nmt*

  Size of array for Mersenne Twister Random Number Generator state.

- `real(kind=dp), dimension(nuni)` *uni*

  Random number generator state for Universal Random Number Generator.

- `integer, dimension(0:nmt)` *mt*

  Random number generator state for Mersenne Twister Random Number Generator.

- `integer, parameter` *mmt*

  Number used to reset Mersenne Twister Random Number Generator array.
- `integer, parameter` *lmask*

  Positive integer limit needed for Mersenne Twister Random Number Generator.
- `integer, parameter` *umask*

  Negative integer limit needed for Mersenne Twister Random Number Generator.
- `integer, parameter` *tmaskb*

  Integer used to generate Mersenne Twister Random Number (second operation)
- `integer, parameter` *tmaskc*

  Integer used to generate Mersenne Twister Random Number (third operation)

10.4.4 Function/Subroutine Documentation

bitadd()

```
integer function numeric_container::bitadd (integer, intent(in) a,
                                            integer, intent(in) b
                                           )
```

Adds two integers together in a bitwise manner without using a longer integer type and truncation for sums that would exceed the limit for a standard integer. This function is used particularly by the Saru random number generator.

Parameters

a	First integer to be added
b	Second integer to be added

bitmult()

```
integer function numeric_container::bitmult (integer, intent(in) a,
                                             integer, intent(in) b
                                            )
```

Multiplies two integers in a bitwise manner without using a longer integer type and truncation for products that would exceed the limit for a standard integer. This function is used particularly by the Saru random number generator.

Parameters

a	First integer to be multiplied
b	Second integer to be multiplied

create_local_id_mol_map()

```
subroutine numeric_container::create_local_id_mol_map (integer, intent(in) nobeads,
                                                       integer, dimension(:),␣
↪intent(in) global_ids,
                                                       integer, dimension(:),␣
↪intent(in) molecules,
                                                       integer, dimension(:),␣
↪intent(out) map
                                                      )
```

Sorts the global particle indices for available particles and uses the sorted list and the original indices to determine the molecule numbers each particle belongs to. This information is only used during simulation startup for information provided in trajectory outputs and is not used during simulations.

Parameters

nobeads	Number of available beads
global_ids	Global particle indices of available beads
molecules	Array of global particle index extents for molecules
map	Resulting array of molecule numbers for available beads

duni()

```
real(kind=dp) function numeric_container::duni (integer seed)
```

The random number generator is an implementation of the Universal Random Number Generator of Marsaglia, Zaman and Tsang [88]. The first call to this routine sets up the initial state using the provided seed and sets an internal variable to indicate whether or not it has been initialised - this is made threadsafe for the OpenMP version. This random number generator is used in DL_MESO_DPD to generate a consistent sequence of random numbers for e.g. simulation startup across all processor nodes without requiring all-processor communications.

Parameters

seed	Initial seed for random number generator

erfcdp()

```
real(kind=dp) function numeric_container::erfcdp (real(kind=dp) x)
```

Calculates the complementary error function (erfc) of an input value using an approximation based on Chebyshev polynomial fitting [52].

Parameters

x	Input value to find complementary error function

erfdp()

```
real(kind=dp) function numeric_container::erfdp (real(kind=dp) x)
```

Calculates the error function (erf) of an input value using value using an approximation based on Chebyshev polynomial fitting [52].

Parameters

x	Input value to find error function

fft3d()

```
subroutine numeric_container::fft3d (complex(kind=dp), dimension (:,:,:),␣
↪intent(inout) array,
                                     integer, intent(in) nx,
                                     integer, intent(in) ny,
                                     integer, intent(in) nz,
                                     logical, intent(in) inv
                                    )
```

Carries out a Fast Fourier Transform (FFT) on a three-dimensional array of complex numbers: this is a helper subroutine to call Singleton's mixed-radix FFT solver and does not need to be called if using other FFT solvers (e.g. ESSL, FFTW).

Parameters

array	Three-dimensional array of complex values to be transformed
nx	Extent of array in x dimension
ny	Extent of array in y dimension
nz	Extent of array in z dimension
inv	Flag to indicate whether or not the inverse transform is to be applied

gaussmp()

```
real(kind=dp) function numeric_container::gaussmp (integer idnode)
```

This is an implementation of the Marsaglia polar method [87] to convert pairs of uniform random numbers generated by the Mersenne Twister method to Gaussian random variables with zero mean and unity variance. Conversion of uniform random numbers only occurs every other call, as the second Gaussian random number is stored for the following call.

Parameters

idnode	Processor number - used as dummy input seed for Mersenne Twister random number generator

idcube()

```
integer function numeric_container::idcube (integer i,
                                            integer j,
                                            integer k,
                                            integer npx,
                                            integer npy,
                                            integer npz
                                           )
```

Using the node coordinates in x, y and z as well as the total numbers of nodes in each dimension, this function calculates the unique number for a given processor node. No check is made to the node coordinates, each of which should be between 0 and the number of nodes in that dimension less 1.

Parameters

i	Node coordinate in x dimension
j	Node coordinate in y dimension
k	Node coordinate in z dimension
npx	Number of nodes in x dimension
npy	Number of nodes in y dimension
npz	Number of nodes in z dimension

images()

```
subroutine numeric_container::images (real(kind=dp) dx,
                                      real(kind=dp) dy,
                                      real(kind=dp) dz,
                                      real(kind=dp) lx,
                                      real(kind=dp) ly,
                                      real(kind=dp) lz,
                                      integer shearx,
                                      integer sheary,
                                      integer shearz,
                                      real(kind=dp) sldx,
                                      real(kind=dp) sldy,
                                      real(kind=dp) sldz
                                     )
```

Calculates the minimum distance between two particles in a periodic orthogonal box, which is adjusted when Lees-Edwards shearing is in use based on the shifting distance of each periodic image.

Parameters

dx	Distance between two particles in x dimension
dy	Distance between two particles in y dimension
dz	Distance between two particles in z dimension
lx	Size of the periodic box in x dimension
ly	Size of the periodic box in y dimension
lz	Size of the periodic box in z dimension
shearx	Flag to indicate if box is undergoing shear orthogonal to x axis
sheary	Flag to indicate if box is undergoing shear orthogonal to y axis
shearz	Flag to indicate if box is undergoing shear orthogonal to z axis
sldx	Distance periodic box is shifted in x dimension
sldy	Distance periodic box is shifted in y dimension
sldz	Distance periodic box is shifted in z dimension

mtrnd()

```
real(kind=dp) function numeric_container::mtrnd (integer seed)
```

The random number generator is an implementation of the Mersenne Twister Random Number Generator of Matsumoto and Nishimura [92] based on a Fortran code by Hiroshi Takano. The first call to this routine sets up the initial state using the provided seed and sets an internal variable to indicate whether or not it has been initialised - this is made threadsafe for the OpenMP version. This random number generator is used in DL_MESO_DPD for DPD random forces or similarly stochastic sections of most pairwise thermostats, as well as generating initial particle velocities during simulation setup.

Parameters

seed	Initial seed for random number generator

qsort_integer()

```
recursive subroutine numeric_container::qsort_integer (integer, dimension (:),␣
↪intent(inout) list,
                                                        integer, dimension (:),␣
↪intent(inout) index,
                                                        integer, intent(in) stride,
                                                        integer, intent(in) low,
                                                        integer, intent(in) high
                                                       )
```

Applies quicksort (partition-based sorting) to an input array while recording the original positions of its items. This subroutine is recursive, i.e. it can call itself to implement the sort over smaller sections of the list, and resorts to a bubble sort for lists of 5 items or fewer.

Parameters

list	List of values to sort
index	Array of original positions for items in sorted list
stride	Stride between values in list to sort
low	Smallest index in list to consider
high	Largest index in list to consider

quicksort_integer_indexed()

```
subroutine numeric_container::quicksort_integer_indexed (integer, dimension (:),␣
↪intent(inout) list,
                                                          integer, intent(in)␣
↪stride,
                                                          integer, intent(in) n,
                                                          integer, dimension (:),␣
↪intent(out) indices
                                                         )
```

Takes a list of values and applies a quicksort algorithm to sort the list in numerical order, while also recording the original positions of the values in that list in another array. The quicksort implementation uses a recursive subroutine to carry out partition-based sorts and may be substituted with similar library routines (e.g. ISORTX in IBM's Engineering and Scientific Subroutine Library, ESSL) that are optimised for particular machines.

Parameters

list	List of values to sort
stride	Stride between values in list to sort
n	Number of items in list
indices	Array of original positions for items in sorted list

sarurnd()

```
real(kind=dp) function numeric_container::sarurnd (integer, intent(in) seed1,
                                                    integer, intent(in) seed2,
                                                    integer, intent(in) seed3
                                                   )
```

The random number generator is an implementation of the Saru Random Number Generator of Steve Worley with a three-seed premixing algorithm [1]. This random number generator is intended for pairwise random force calculations, where the first two seeds are the global particle indices in numerical order and the third is the timestep number. No state needs to be stored and the generated numbers are independent of processor number or thread: this random number generator can thus be used to ensure the same random forces are calculated regardless of the number of processors used in calculations. This random number generator is currently used in DL_MESO_DPD when the same random number is required more than once for a given particle pair, i.e. for DPD using Shardlow splitting.

Parameters

seed1	Seed 1 for random number generator - smaller particle index
seed2	Seed 2 for random number generator - larger particle index
seed3	Seed 3 for random number generator - timestep number

sclsum()

```
real(kind=dp) function numeric_container::sclsum (integer n,
                                                   real(kind=dp), dimension(:) a,
                                                   integer i
                                                  )
```

Calculates the scalar sum of a double precision array that can represent multiple vectors or a simple series of values.

Parameters

n	Number of elements to sum together
a	Array with values to sum together
i	Stride or distance in array between consecutive values for summation

sfft()

```
subroutine numeric_container::sfft (complex(kind=dp), dimension (:), intent(inout)␣
↪array,
                                    integer, intent(in) ntotal,
                                    integer, intent(in) npass,
                                    integer, intent(in) nspan,
                                    logical, intent(in) inv,
                                    integer, intent(out) ierr
                                   )
```

Applies a Fast Fourier Transform to the supplied complex array using Singleton's mixed-radix algorithm [122] restricted here to dimensions with factors of 2, 3 and/or 5. This subroutine requires the array to be one-dimensional, but strides can be specified to ensure transforms are applied in the right directions: the results are supplied in the same input array. Either forward or inverse transforms can be applied, and a non-zero error value can be output if the number of grid points cannot be factorised or if there are too many prime factors to consider.

Parameters

array	Array of complex values to be transformed
ntotal	Total number of values in the supplied array
npass	Number of values in required dimension
nspan	Stride in values for given dimension
inv	Flag to indicate whether or not the inverse transform is to be applied
ierr	Error reporting value (0 if no error, -1 if array size cannot be factorised, -2 if too many prime factors)

10.4.5 Variable Documentation

lmask

```
integer parameter numeric_container::lmask = 2147483647
```

Highest positive signed integer value available in Fortran, required for Mersenne Twister Random Number Generator.

mmt

```
integer parameter numeric_container::mmt = 397
```

Number used in Mersenne Twister Random Number Generator when resetting state array.

mt

```
integer, dimension(0:nmt) numeric_container::mt
```

Array used as the state for the Mersenne Twister Random Number Generator (one produced per processor and written to REVIVE file for simulation restart).

nmt

```
integer parameter numeric_container::nmt = 624
```

Size of arrays required for Mersenne Twister Random Number Generator states.

nuni

```
integer parameter numeric_container::nuni = 102
```

Size of array required for Universal Random Number Generator state.

tmaskb

```
integer parameter numeric_container::tmaskb = -1658038656
```

Integer required during second operation in Mersenne Twister Random Number Generator to advance state and generate subsequent random number.

tmaskc

```
integer parameter numeric_container::tmaskc = -272236544
```

Integer required during third operation in Mersenne Twister Random Number Generator to advance state and generate subsequent random number.

umask

```
integer parameter numeric_container::umask = -LMASK-1
```

Highest negative signed integer value available in Fortran, required for Mersenne Twister Random Number Generator.

uni

```
real(kind=dp), dimension(nuni) numeric_container::uni
```

Array used as the state for the Universal Random Number Generator (single state used for all processors to generate consistent sequence for simulation setup).

10.5 parse_utils.F90

10.5.1 Summary

Module with functions and subroutines required to parse text (including numbers) read from input files.

10.5.2 Functions/Subroutines

- `character(len=mxword) function, public` *getword()*

  Gets a specified word from a line of delimited text.

- `integer(kind=li) function, public` *parseint()*

  Obtains an integer from a given word (string).

- `real(kind=dp) function, public` *parsedble()*

  Obtains a double precision real number from a given word (string).

- `integer(kind=li) function, public` *getint()*

  Gets a specified word from a line of delimited text and obtains a long integer from that word.

- `real(kind=dp) function, public` *getdble()*

  Gets a specified word from a line of delimited text and obtains a double precision real number from that word.

- `subroutine, public` *lowercase()*

  Converts all uppercase letters in a string to lowercase.

10.5.3 Function/Subroutine Documentation

getdble()

```
real(kind=dp) function, public parse_utils::getdble (character(len=*), intent(in)␣
↪txt,
                                                        integer n
                                                      )
```

Obtains the n-th word from a line of text separated by spaces, commas or tabs and parse the word to produce a double precision real number. If the word includes any characters besides numerals, plus/minus signs or exponent characters (E or e), the function will return zero.

Parameters

txt	Line of text to be read in
n	Required word number from which to obtain double precision real number

getint()

```
integer(kind=li) function, public parse_utils::getint (character(len=*),␣
↪intent(in) txt,
                                                          integer n
                                                        )
```

Obtains the n-th word from a line of text separated by spaces, commas or tabs and parse the word to produce a long integer. If the word includes a dot (decimal point), the number is truncated at that point, while any other characters besides numerals and minus signs will produce a value of zero.

Parameters

txt	Line of text to be read in
n	Required word number from which to obtain integer

getword()

```
character(len=mxword) function, public parse_utils::getword (character(len=*),␣
↪intent(in) txt,
                                                                integer n
                                                              )
```

Obtains the n-th word from a line of text separated by spaces, commas or tabs and outputs the word as a string. If the input text contains fewer words than the number specified, the output string will be a single space.

Parameters

txt	Line of text to be read in
n	Required word number

lowercase()

```
subroutine, public parse_utils::lowercase (character(len=*), intent(inout) word)
```

Changes any uppercase letters in a provided word (string) to lowercase letters. This subroutine is used to enable input files to have any form of capitalisation of keywords.

Parameters

word	String to be converted

parsedble()

```
real(kind=dp) function, public parse_utils::parsedble (character(len=*),␣
↪intent(in) word)
```

Reads a string to find the double precision real number contained inside it and outputs its value. If the string includes any characters besides numerals, positive/minus signs or an exponent character (E or e), zero will be returned.

Parameters

word	String to be parsed

parseint()

```
integer(kind=li) function, public parse_utils::parseint (character(len=*),␣
↪intent(in) word)
```

Reads a string to find the long integer contained inside it and outputs its value. If the string includes a dot, the function will truncate the number at that point as a decimal point. If the string contains any other characters besides numerals or a minus sign, zero will be returned.

Parameters

word	String to be parsed

10.6 bond_module.F90

10.6.1 Summary

Module to maintain book-keeping and calculate forces for bonds, angles and dihedrals. (OpenMP multithreaded version available with bond_module_omp.F90.)

10.6.2 Functions/Subroutines

- `subroutine` *shellsort_list()*

  Reorders list of global/local particle numbers in terms of global particle numbers using a Shell sort.

- `integer function` *search_list()*

  Searches a sorted list of global/local particle numbers to find a given value among the global particle numbers.

- `subroutine` *contract_bndtbl()*

  Strips out all bond pairs from bond table that have been reassigned to neighbouring processors.

- `subroutine` *contract_angtbl()*

  Strips out all bond angle triples from angle table that have been reassigned to neighbouring processors.

- `subroutine` *contract_dhdtbl()*

  Strips out all bond dihedral quadruples from dihedral table that have been reassigned to neighbouring processors.

- `subroutine` *bond_force()*

  Calculates the stretching force and potential energy between a pair of bonded particles.

- `subroutine` *angle_force()*

  Calculates the bond angle force, potential energy and virial across a triple of bonded particles.

- `subroutine` *dihedral_force()*

  Calculates the bond dihedral force and potential energy across a quadruple of bonded particles.

- `subroutine` *bondforceslocal()*

  Calculates all bond (stretching, angle, dihedral) forces between particles in system using locally-defined bond, angle and dihedral lists.

- `subroutine` *bondpotentialslocal()*

  Calculates all bond (stretching, angle, dihedral) potentials between particles in system using locally-defined bond, angle and dihedral lists.

- `subroutine` *bondforcesglobal()*

  Calculates all bond (stretching, angle, dihedral) forces between particles in system using globally-defined bond, angle and dihedral lists.

- `subroutine` *bondpotentialsglobal()*

  Calculates all bond (stretching, angle, dihedral) potentials between particles in system using globally-defined bond, angle and dihedral lists.

10.6.3 Function/Subroutine Documentation

angle_force()

```
subroutine bond_module::angle_force (integer angtype,
                                     real(kind=dp) theta,
                                     real(kind=dp) rab,
                                     real(kind=dp) rcb,
                                     real(kind=dp) a,
                                     real(kind=dp) b,
                                     real(kind=dp) c,
                                     real(kind=dp) d,
```

(continues on next page)

(continued from previous page)

```
                                          real(kind=dp) force,
                                          real(kind=dp) potential,
                                          real(kind=dp) virial,
                                          real(kind=dp) dfab,
                                          real(kind=dp) dfcb
                                         )
```

This routine calculates bond angle forces, potentials and virials among three particles connected together by a pair of bonds. The three current options for these interactions are harmonic:

$$U_{ijk} = \frac{\kappa}{2}\left(\theta_{ijk} - \theta_0\right)^2$$

harmonic cosine:

$$U_{ijk} = \frac{\kappa}{2}\left(\cos\theta_{ijk} - \cos\theta_0\right)^2$$

and cosine angle potentials:

$$U_{ijk} = A\left[1 + \cos\left(m\theta_{ijk} - \delta\right)\right]$$

These can be expanded upon by the user. Additional forces and virial contributions for the two end particles from screening or truncation functions for the given bond angle are available as an option: the currently available angles do not make use of these and give a virial of zero [126], although they still contribute to the stress tensor. Note that the force is divided by the sine of the angle in this subroutine to optimise calculation of the forces acting on all three particles.

Parameters

angtype	Type (functional form) of bond angle among three particles
theta	Angle among three particles, θ_{ijk}, centred on particle j
rab	Vector between particles i and j
rcb	Vector between particles k and j
a	Parameter a for bond angle interaction
b	Parameter b for bond angle interaction
c	Parameter c for bond angle interaction
d	Parameter d for bond angle interaction
force	Resulting bond angle force among particles divided by the sine of the angle, $\frac{F_{ijk}}{\sin\theta_{ijk}}$
potential	Resulting bond angle potential, U_{ijk}
virial	Additional virial contribution resulting from bond angle
dfab	Additional force acting between particles i and j resulting from bond angle
dfcb	Additional force acting between particles k and j resulting from bond angle

bond_force()

```
subroutine bond_module::bond_force (integer bondtype,
                                    real(kind=dp) r,
                                    real(kind=dp) a,
                                    real(kind=dp) b,
                                    real(kind=dp) c,
                                    real(kind=dp) d,
                                    real(kind=dp) force,
                                    real(kind=dp) potential,
                                    real(kind=dp) mxlen
                                   )
```

This routine calculates bond stretching forces and potentials between connected pairs of particles. The four current options for these interactions are harmonic (Hookean/Fraenkel):

$$U_{ij} = \frac{\kappa}{2}\left(r_{ij} - r_0\right)^2$$

finitely-extensible non-linear elastic (FENE):

$$U_{ij} = \begin{cases} -\frac{1}{2}\kappa r_{max}^2 \ln \left[1 - \frac{(r_{ij}-r_0)^2}{r_{max}^2}\right] & r_{ij} < r_0 + r_{max} \\ \infty & r_{ij} \geq r_0 + r_{max} \end{cases}$$

Marko-Siggia Wormlike Chains (WLC) [86]:

$$U_{ij} = \begin{cases} \frac{k_B T}{2A_p} \left[\frac{1}{2\left(1-\frac{r_{ij}}{r_{max}}\right)} - \frac{1}{2}\left(1 + \frac{r_{ij}}{r_{max}}\right) + \frac{r_{ij}^2}{r_{max}^2}\right] & r_{ij} < r_{max} \\ \infty & r_{ij} \geq r_{max} \end{cases}$$

and Morse anharmonic [94] bonds:

$$U_{ij} = D_e \left[1 - \exp\left(-\beta\left(r_{ij} - r_0\right)\right)\right]^2$$

These can be expanded upon by the user. Note that the force is divided by the distance in this subroutine so that multiplying this value by the vector between the particles gives the required scalar force multiplied by the unit vector.

Parameters

bond-type	Type (functional form) of bond between particle pair
r	Distance between pair of particles, r_{ij}
a	Parameter a for bond interaction
b	Parameter b for bond interaction
c	Parameter c for bond interaction
d	Parameter d for bond interaction
force	Resulting bond force between particles divided by distance, $\frac{\vec{F_{ij}}}{r_{ij}}$
potential	Resulting bond potential, U_{ij}
mxlen	Maximum possible length for bond - used in error reporting if distance between particles is too large

bondforcesglobal()

```
subroutine bond_module::bondforcesglobal (integer nlimit)
```

Calculates all bonded interaction forces and potentials (stretching, angles and dihedrals) between particles using book-keeping tables that hold all available bond information for the entire system (i.e. globally-defined lists). This is less efficient for parallel running but ensures longer distances between particle pairs (e.g. in bonds) can be accommodated, particularly during equilibration. The OpenMP version of this subroutine divides the bonds, angles and dihedrals among the available threads, either using additional memory per thread or a critical region to assign forces to particles in a threadsafe manner.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

bondforceslocal()

```
subroutine bond_module::bondforceslocal (integer nlimit)
```

Calculates all bonded interaction forces and potentials (stretching, angles and dihedrals) between particles using book-keeping tables that are continously updated to only include particles held by each processor (i.e. locally-defined lists). This is the most efficient method for parallel running but runs the risk of losing track of bonded pairs etc. if distances between particle pairs become longer than the subdomain halo size. The OpenMP version of this subroutine divides the bonds, angles and dihedrals among the available threads, either using additional memory per thread or a critical region to assign forces to particles in a threadsafe manner.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

bondpotentialsglobal()

```
subroutine bond_module::bondpotentialsglobal (integer nlimit)
```

Calculates all bonded interaction potentials (stretching, angles and dihedrals) between particles using book-keeping tables that hold all available bond information for the entire system (i.e. globally-defined lists). This subroutine is intended to calculate potentials, virials and stress tensor contributions at the start of a DPD simulation when particle forces are already known. The OpenMP version of this subroutine divides the bonds, angles and dihedrals among the available threads.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

bondpotentialslocal()

```
subroutine bond_module::bondpotentialslocal (integer nlimit)
```

Calculates all bonded interaction potentials (stretching, angles and dihedrals) between particles using book-keeping tables that are continously updated to only include particles held by each processor (i.e. locally-defined lists). This subroutine is intended to calculate potentials, virials and stress tensor contributions at the start of a DPD simulation when particle forces are already known. The OpenMP version of this subroutine divides the bonds, angles and dihedrals among the available threads.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

contract_angtbl()

```
subroutine bond_module::contract_angtbl
```

Goes through each processor's table of angles and takes out any angles that have been marked for removal after being moved to a neighbouring processor. Only called by parallel version of DL_MESO_DPD when angle tables include only local angles in each processor: tables include all angles when running in serial or using 'global bonds' option in CONTROL file.

contract_bndtbl()

```
subroutine bond_module::contract_bndtbl
```

Goes through each processor's table of bonds and takes out any bonds that have been marked for removal after being moved to a neighbouring processor. Only called by parallel version of DL_MESO_DPD when bond tables include only local bonds in each processor: tables include all bonds when running in serial or using 'global bonds' option in CONTROL file.

contract_dhdtbl()

```
subroutine bond_module::contract_dhdtbl
```

Goes through each processor's table of dihedrals and takes out any dihedrals that have been marked for removal after being moved to a neighbouring processor. Only called by parallel version of DL_MESO_DPD when dihedral tables include only local dihedrals in each processor: tables include all dihedrals when running in serial or using 'global bonds' option in CONTROL file.

dihedral_force()

```
subroutine bond_module::dihedral_force (integer dhdtype,
                                        real(kind=dp) phi,
                                        real(kind=dp) rpb,
                                        real(kind=dp) rpc,
                                        real(kind=dp) a,
                                        real(kind=dp) b,
                                        real(kind=dp) c,
                                        real(kind=dp) d,
                                        real(kind=dp) force,
                                        real(kind=dp) potential)
```

This routine calculates bond dihedral forces and potentials among four particles connected together by a pair of planes each formed from two bonds or vectors. The three current options for these interactions are torsion:

$$U_{ijkl} = A\left[1 + \cos\left(m\phi_{ijkl} - \delta\right)\right]$$

harmonic improper:

$$U_{ijkl} = \frac{\kappa}{2}\left(\phi_{ijkl} - \phi_0\right)^2$$

and harmonic cosine dihedral potentials:

$$U_{ijkl} = \frac{\kappa}{2}\left(\cos\phi_{ijkl} - \cos\phi_0\right)^2$$

These can be expanded upon by the user. Dihedrals do not contribute to the virial [126], although they still contribute to the stress tensor. Note that the force is multiplied by the product of the dihedral vectors divided by the sine of the dihedral angle in this subroutine to optimise calculation of the forces acting on all four particles.

Parameters

dhd-type	Type (functional form) of bond dihedral among four particles
phi	Dihedral angle among four particles, ϕ_{ijkl}
rpb	Dihedral vector between vectors ij and jk, $\vec{r}_{ij} \times \vec{r}_{jk}$
rpc	Dihedral vector between vectors jk and kl, $\vec{r}_{jk} \times \vec{r}_{kl}$
a	Parameter a for bond dihedral interaction
b	Parameter b for bond dihedral interaction
c	Parameter c for bond dihedral interaction
d	Parameter d for bond dihedral interaction
force	Resulting bond dihedral force among particles multiplied by the product of dihedral vectors and divided by the sine of the dihedral angle, $\frac{F_{ijkl}(\vec{r}_{ij}\times\vec{r}_{jk})\cdot(\vec{r}_{jk}\times\vec{r}_{kl})}{\sin\phi_{ijkl}}$
po-ten-tial	Resulting bond dihedral potential, U_{ijkl}

search_list()

```
integer function bond_module::search_list (integer aim)
```

Carries out a binary search of the global particle numbers in a sorted list of global/local particle numbers to output the entry in the list that has the provided input value (of a global particle number). If no entry with this input value can be found, a negative number is output to indicate this. (Note that the list *must* be sorted in numerical order for global particle numbers beforehand to ensure searches can be carried out.)

Parameters

aim	Global particle number value to search for in global/local particle number list

shellsort_list()

```
subroutine bond_module::shellsort_list
```

Takes a list of pairs of values - global and local particle numbers - and reorders the pairs in numerical order for the global particle numbers by using a Shell sort.

10.7 comms_module.F90

10.7.1 Summary

Module to handle node-to-node communications using MPI in parallel or substitute/dummy subroutines in serial (as comms_module_ser.F90).

10.7.2 Functions/Subroutines

- `subroutine` *initcomms()*

  Starts off MPI, sets up communicators and works out kinds for various types of number used in DL_MESO_DPD.

- `subroutine` *exitcomms()*

  Close down MPI after completion of DL_MESO_DPD calculation in a controlled manner.

- `subroutine` *abortcomms()*

  Terminates MPI before the projected end of a calculation.

- `subroutine` *gsync()*

  Synchronises all processors before continuing.

- `subroutine` *group_gsync()*

  Synchronises all processors in a given group before continuing.

- `subroutine` *global_and_all()*

  Applies a global AND operation to a logical array and broadcasts result to all processors.

- `subroutine` *global_and()*

  Applies a global AND operation to a logical array and broadcasts result to a selected processor.

- `subroutine` *global_or_all()*

  Applies a global OR operation to a logical array and broadcasts result to all processors.

- `subroutine` *global_or()*

  Applies a global OR operation to a logical array and broadcasts result to a selected processor.
- `subroutine` *global_sca_and_all()*

  Applies a global AND operation to a logical scalar and broadcasts result to all processors.
- `subroutine` *global_sca_and()*

  Applies a global AND operation to a logical scalar and broadcasts result to a selected processor.
- `subroutine` *global_sca_or_all()*

  Applies a global OR operation to a logical scalar and broadcasts result to all processors.
- `subroutine` *global_sca_or()*

  Applies a global OR operation to a logical scalar and broadcasts result to a selected processor.
- `subroutine` *global_sca_max_int()*

  Find the global maximum for an integer and broadcast result to all processors.
- `subroutine` *global_sca_max_dble()*

  Find the global maximum for a double precision real number and broadcast result to all processors.
- `subroutine` *global_sca_min_int()*

  Find the global minimum for an integer and broadcast result to all processors.
- `subroutine` *global_sca_min_dble()*

  Find the global minimum for a double precision real number and broadcast result to all processors.
- `subroutine` *global_sum_dble()*

  Applies a global summation to a double precision real array and broadcasts result to all processors.
- `subroutine` *global_sum_sca_dble()*

  Applies a global summation to a double precision real number and broadcasts result to all processors.
- `subroutine` *global_sum_int()*

  Applies a global summation to an integer array and broadcasts result to all processors.
- `subroutine` *global_sum_sca_int()*

  Applies a global summation to an integer and broadcasts result to all processors.
- `subroutine` *group_sum_sca_int()*

  Applies a group-wide summation to an integer and broadcasts result to the root processor for the group.
- `subroutine` *group_sum_sca_int_all()*

  Applies a group-wide summation to an integer and broadcasts result to all processors in the group.
- `subroutine` *group_sum_cmplx_all()*

  Applies a group-wide summation to a double precision complex array and broadcasts result to all processors in the group.
- `subroutine` *broadcast_dble()*

  Broadcasts a double precision real array of numbers from a root processor to all other processors.
- `subroutine` *broadcast_sca_dble()*

  Broadcasts a double precision real number from a root processor to all other processors.
- `subroutine` *broadcast_int()*

  Broadcasts an integer array of numbers from a root processor to all other processors.

- `subroutine` *broadcast_sca_int()*

  Broadcasts an integer from a root processor to all other processors.

- `subroutine` *broadcast_int1()*

  Broadcasts an one-byte integer array of numbers from a root processor to all other processors.

- `subroutine` *broadcast_sca_int1()*

  Broadcasts a one-byte integer from a root processor to all other processors.

- `subroutine` *broadcast_logical()*

  Broadcasts a logical array from a root processor to all other processors.

- `subroutine` *broadcast_sca_logical()*

  Broadcasts a logical value from a root processor to all other processors.

- `subroutine` *broadcast_char()*

  Broadcasts a series of characters from a root processor to all other processors.

- `subroutine` *distribute_int_data()*

  Distributes an array of integers to intended destination processors within defined group.

- `subroutine` *distribute_int_data_root()*

  Distributes an array of integers to the root processor within defined group.

- `subroutine` *distribute_int_data_all()*

  Distributes an array of integers to all processors within defined group.

- `subroutine` *distribute_sca_int_data()*

  Distributes an array of integers from each processor to all processors within defined group and gather together an array with values received from each processor.

- `subroutine` *distribute_sca_int_data_root()*

  Distributes scalar integers to the root processor for the defined group.

- `subroutine` *distribute_sca_int_data_all()*

  Distributes scalar integers to all processors for the defined group.

- `subroutine` *distribute_dble_data()*

  Distributes an array of double precision real numbers to intended destination processors within defined group.

- `subroutine` *distribute_dble_data_root()*

  Distributes an array of double precision real numbers to the root processor within defined group.

- `subroutine` *distribute_dble_data_all()*

  Distributes an array of double precision real numbers to all processors within defined group.

- `subroutine` *distribute_sca_dble_data()*

  Distributes an array of double precision real numbers from each processor to all processors within defined group and gather together an array with values received from each processor.

- `subroutine` *distribute_sca_dble_data_root()*

  Distributes scalar double precision real numbers to the root processor for the defined group.

- `subroutine` *distribute_sca_dble_data_all()*

  Distributes scalar double precision real numbers to all processors for the defined group.

- `subroutine` *create_group_comms()*

  Creates an MPI communicator among a group of processors.
- `subroutine` *duplicate_group_comms()*

  Create a copy of an existing MPI communicator.
- `subroutine` *io_open()*

  Open a binary output file among processors sharing an MPI communicator for writing.
- `subroutine` *io_ansi_open()*

  Open an ANSI text output file among processors sharing an MPI communicator for writing.
- `subroutine` *io_close()*

  Closes a previously-opened output file.
- `subroutine` *io_write_int()*

  Writes a single integer to a previously-opened binary output file at a given position.
- `subroutine` *io_write_longint()*

  Writes a single long integer to a previously-opened binary output file at a given position.
- `subroutine` *io_write_dble()*

  Writes a single double precision real number to a previously-opened binary output file at a given position.
- `subroutine` *io_write_batch_int()*

  Writes an array of integers to a previously-opened binary output file starting at a given position.
- `subroutine` *io_write_batch_dble()*

  Writes an array of double precision real numbers to a previously-opened binary output file starting at a given position.
- `subroutine` *io_write_batch_char()*

  Writes an array of characters to a previously-opened ANSI text output file starting at a given position.
- `subroutine` *msg_receive_blocked()*

  Receive a data array from any processor in a blocking MPI call.
- `subroutine` *msg_receive_blocked_pe()*

  Receive a data array from a specific processor in a blocking MPI call.
- `subroutine` *msg_receive_sca_blocked()*

  Receive a single value from any processor in a blocking MPI call.
- `subroutine` *msg_receive_sca_blocked_pe()*

  Receive a single value from a specific processor in a blocking MPI call.
- `integer function` *msg_receive_unblocked()*

  Receive a data array from any processor in a non-blocking MPI call.
- `integer function` *msg_receive_unblocked_pe()*

  Receive a data array from a specific processor in a non-blocking MPI call.
- `integer function` *msg_receive_unblocked_grid_pe()*

  Receive a three-dimensional data array from a specific processor in a non-blocking MPI call.
- `subroutine` *msg_send_blocked()*

  Send a data array to a specific processor in a blocking MPI call.

- `subroutine` *msg_send_blocked_grid()*

  Send a three-dimensional data array to a specific processor in a blocking MPI call.

- `subroutine` *msg_send_sca_blocked()*

  Send a single data value to a specific processor in a blocking MPI call.

- `integer function` *msg_send_unblocked()*

  Send a data array to a specific processor in a non-blocking MPI call.

- `subroutine` *msg_wait()*

  Causes processor to wait for an non-blocking MPI message.

- `integer function` *msg_wait_and_size_double()*

  Causes processor to wait for an non-blocking MPI message and returns size of message.

- `integer function` *mynode()*

  Returns number (rank) of current processor.

- `integer function` *numnodes()*

  Returns total number of available processors.

- `subroutine` *timchk()*

  Determines the time elapsed since the start of the calculation and (if requested) prints the time to OUTPUT file.

10.7.3 Variables

- `integer, save` *dlm_comm_world* `= 0`

  All node communicator for instance of DL_MESO_DPD.

- `integer, save` *self_comm* `= 0`

  Communicator for current node.

- `integer, save` *dp_mpi* `= 0`

  MPI kind value for double precision real numbers.

- `integer, save` *cx_mpi* `= 0`

  MPI kind value for double precision complex numbers.

- `integer, save` *dlen* `= 0`

  Size of double precision real number in bytes, given as standard integer.

- `integer, save` *ilen* `= 0`

  Size of standard integer in bytes, given as standard integer.

- `integer, save` *lilen* `= 0`

  Size of long integer in bytes, given as standard integer.

- `integer(kind=li), save` *dlen_li* `= 0`

  Size of double precision real number in bytes, given as long integer.

- `integer(kind=li), save` *ilen_li* `= 0`

  Size of standard integer in bytes, given as long integer.

- `integer(kind=li), save` *lilen_li* `= 0`

  Size of long integer in bytes, given as standard integer.

10.7.4 Function/Subroutine Documentation

abortcomms()

```
subroutine comms_module::abortcomms
```

In the event of a fatal error occurring during a calculation, this subroutine will terminate MPI and bring DL_MESO to a sudden halt, ideally after printing an error message indicating what went wrong. (Dummy subroutine in serial.)

broadcast_char()

```
subroutine comms_module::broadcast_char (character(len=nnn) aaa,
                                         integer, intent(in) nnn,
                                         integer, intent(in) root
                                        )
```

Carries out an MPI_bcast operation on a string of characters located in a root processor to broadcast the string to all other processors. (Dummy subroutine in serial.)

Parameters

aaa	Array of characters to broadcast/receive
nnn	Size of character array (string)
root	Number of processor broadcasting array to other processors

broadcast_dble()

```
subroutine comms_module::broadcast_dble (real(kind=dp), dimension(:) aaa,
                                         integer, intent(in) nnn,
                                         integer, intent(in) root
                                        )
```

Carries out an MPI_bcast operation on an array of double precision real numbers located in a root processor to broadcast the values to all other processors. (Dummy subroutine in serial.)

Parameters

aaa	Array of double precision real numbers to broadcast/receive
nnn	Size of double precision real array
root	Number of processor broadcasting array to other processors

broadcast_int()

```
subroutine comms_module::broadcast_int (integer, dimension(:) aaa,
                                        integer, intent(in) nnn,
                                        integer, intent(in) root
                                       )
```

Carries out an MPI_bcast operation on an array of integers located in a root processor to broadcast the values to all other processors. (Dummy subroutine in serial.)

Parameters

aaa	Array of integers to broadcast/receive
nnn	Size of integer array
root	Number of processor broadcasting array to other processors

broadcast_int1()

```
subroutine comms_module::broadcast_int1 (integer(kind=1), dimension(:) aaa,
                                         integer, intent(in) nnn,
                                         integer, intent(in) root
                                        )
```

Carries out an MPI_bcast operation on an array of one-byte integers located in a root processor to broadcast the values to all other processors. (Dummy subroutine in serial.)

Parameters

aaa	Array of one-byte integers to broadcast/receive
nnn	Size of one-byte integer array
root	Number of processor broadcasting array to other processors

broadcast_logical()

```
subroutine comms_module::broadcast_logical (logical, dimension(:) aaa,
                                            integer, intent(in) nnn,
                                            integer, intent(in) root
                                           )
```

Carries out an MPI_bcast operation on an array of logical values located in a root processor to broadcast the values to all other processors. (Dummy subroutine in serial.)

Parameters

aaa	Array of logicals to broadcast/receive
nnn	Size of logical array
root	Number of processor broadcasting array to other processors

broadcast_sca_dble()

```
subroutine comms_module::broadcast_sca_dble (real(kind=dp) aaa,
                                             integer, intent(in) root
                                            )
```

Carries out an MPI_bcast operation on a double precision real number located in a root processor to broadcast the value to all other processors. (Dummy subroutine in serial.)

Parameters

aaa	Double precision real number to broadcast/receive
root	Number of processor broadcasting number to other processors

broadcast_sca_int()

```
subroutine comms_module::broadcast_sca_int (integer aaa,
                                            integer, intent(in) root
                                           )
```

Carries out an MPI_bcast operation on an integer located in a root processor to broadcast the value to all other processors. (Dummy subroutine in serial.)

Parameters

aaa	Integer to broadcast/receive
root	Number of processor broadcasting number to other processors

broadcast_sca_int1()

```
subroutine comms_module::broadcast_sca_int1 (integer(kind=1) aaa,
                                             integer, intent(in) root
                                            )
```

Carries out an MPI_bcast operation on a one-byte integer located in a root processor to broadcast the value to all other processors. (Dummy subroutine in serial.)

Parameters

aaa	One-byte integer to broadcast/receive
root	Number of processor broadcasting number to other processors

broadcast_sca_logical()

```
subroutine comms_module::broadcast_sca_logical (logical aaa,
                                                integer, intent(in) root
                                               )
```

Carries out an MPI_bcast operation on a logical value located in a root processor to broadcast the value to all other processors. (Dummy subroutine in serial.)

Parameters

aaa	Logical to broadcast/receive
root	Number of processor broadcasting value to other processors

create_group_comms()

```
subroutine comms_module::create_group_comms (integer, dimension (:), intent(in)␣
↪members,
                                             integer, intent(in) number,
                                             integer, intent(inout) group,
                                             integer, intent(inout) comm,
                                             integer, intent(inout) rank
                                            )
```

Create a group of processors and the necessary communicator for that group based on a list of available processors for the DL_MESO_DPD calculation, identifying the rank of the current processor in that group if it is included. (Dummy subroutine in serial.)

Parameters

members	Array of processors to be included in group
number	Number of processors in group
group	MPI group identifier (not normally used directly)
comm	MPI communicator for group
rank	Rank for current processor in group (only defined if processor is included)

distribute_dble_data()

```
subroutine comms_module::distribute_dble_data (real(kind=dp), dimension (:),␣
↪intent(in) senddata,
                                               integer, dimension (:), intent(in)␣
↪sendcount,
                                               real(kind=dp), dimension (:),␣
↪intent(inout) recvdata,
                                               integer, dimension (:), intent(in)␣
↪recvcount,
                                               integer, dimension (:), intent(in)␣
↪displ,
                                               integer, dimension (:), intent(in)␣
↪nodemap,
                                               integer, intent(in) nodes,
                                               integer, intent(in) group
                                              )
```

Applies a series of MPI_Gatherv calls to send data in the form of an array of double precision real numbers to their intended destinations - given as arrays indicating which value goes to which processor and where the data should be placed in the destination array - within a given group. The data array to be sent should be sorted by destination processor to give continguous groups of values going to each processor. Note that this subroutine can be used to distribute data to all available processors by using the all-node communicator. (The serial version of this subroutine directly copies the array for sending data into the array for receiving data.)

Parameters

send-data	Array of double precision real numbers being sent to all processors
send-count	Array containing total numbers of double precision real numbers being sent by each processor
recv-data	Array of double precision real numbers with values arriving from all processors
recv-count	Array containing total numbers of double precision real numbers arriving from each processor
displ	Array indicating where data from each processor should be placed in data-receiving array (as a displacement from the start)
nodemap	Array indicating which data value is going to which processor - should be continguous for each processor and in numerical order
nodes	Total number of processors in group involved in data distribution
group	Communicator for group of processors

distribute_dble_data_all()

```
subroutine comms_module::distribute_dble_data_all (real(kind=dp), dimension (:),␣
↪intent(in) senddata,
                                                   integer, intent(in) sendcount,
                                                   real(kind=dp), dimension (:),␣
↪intent(inout) recvdata,
                                                   integer, dimension (:),␣
↪intent(in) recvcount,
                                                   integer, dimension (:),␣
↪intent(in) displ,
                                                   integer, intent(in) group
                                                  )
```

Applies an MPI_Allgatherv call to send data in the form of an array of double precision real numbers to all processors in the group as a combined array. An array indicating where each processor's data should be placed in the destination array is required. Note that this subroutine can be used to distribute data to all available processors by using the all-node communicator. (The serial version of this subroutine directly copies the array for sending data into the array for receiving data.)

Parameters

senddata	Array of double precision real numbers being sent to all processors
sendcount	Total number of double precision real numbers being sent by current processor
recvdata	Array of double precision real numbers with values arriving from all processors
recvcount	Array containing total numbers of double precision real numbers arriving from each processor
displ	Array indicating where data from each processor should be placed in data-receiving array (as a displacement from the start)
group	Communicator for group of processors

distribute_dble_data_root()

```
subroutine comms_module::distribute_dble_data_root (real(kind=dp), dimension (:),␣
↪intent(in) senddata,
                                                    integer, intent(in) sendcount,
                                                    real(kind=dp), dimension (:),␣
↪intent(inout) recvdata,
                                                    integer, dimension (:),␣
↪intent(in) recvcount,
                                                    integer, dimension (:),␣
↪intent(in) displ,
                                                    integer, intent(in) group
                                                   )
```

Applies a single MPI_Gatherv call to send data in the form of an array of double precision real numbers to the root processor for the given group as a single combined array. An array indicating where each processor's data should be placed in the destination array is required. Note that this subroutine can be used to distribute data from all available processors by using the all-node communicator. (The serial version of this subroutine directly copies the array for sending data into the array for receiving data.)

Parameters

send-data	Array of double precision real numbers being sent to all processors
send-count	Total number of double precision real numbers being sent by current processor
recv-data	Array of double precision real numbers with values arriving from all processors (only needs to be allocated for root processor)
recv-count	Array containing total numbers of double precision real numbers arriving at root processor from each processor
displ	Array indicating where data from each processor should be placed in data-receiving array (as a displacement from the start)
group	Communicator for group of processors

distribute_int_data()

```
subroutine comms_module::distribute_int_data (integer, dimension (:), intent(in)␣
↪senddata,
                                              integer, dimension (:), intent(in)␣
↪sendcount,
                                              integer, dimension (:),␣
↪intent(inout) recvdata,
                                              integer, dimension (:), intent(in)␣
↪recvcount,
                                              integer, dimension (:), intent(in)␣
↪displ,
                                              integer, dimension (:), intent(in)␣
↪nodemap,
                                              integer, intent(in) nodes,
                                              integer, intent(in) group
                                             )
```

Applies a series of MPI_Gatherv calls to send data in the form of an array of integers to their intended destinations - given as arrays indicating which value goes to which processor and where the data should be placed in the destination array - within a given group. The data array to be sent should be sorted by destination processor to give continguous groups of values going to each processor. Note that this subroutine can be used to distribute data to all available processors by using the all-node communicator. (The serial version of this subroutine directly copies the array for sending data into the array for receiving data.)

Parameters

send-data	Array of integers being sent to all processors
send-count	Array containing total numbers of integers being sent by each processor
recv-data	Array of integers with values arriving from all processors
recv-count	Array containing total numbers of integers arriving from each processor
displ	Array indicating where data from each processor should be placed in data-receiving array (as a displacement from the start)
nodemap	Array indicating which data value is going to which processor - should be continguous for each processor and in numerical order
nodes	Total number of processors in group involved in data distribution
group	Communicator for group of processors

distribute_int_data_all()

```
subroutine comms_module::distribute_int_data_all (integer, dimension (:),␣
↪intent(in) senddata,
                                                   integer, intent(in) sendcount,
                                                   integer, dimension (:),␣
↪intent(inout) recvdata,
                                                   integer, dimension (:),␣
↪intent(in) recvcount,
                                                   integer, dimension (:),␣
↪intent(in) displ,
                                                   integer, intent(in) group
                                                  )
```

Applies an MPI_Allgatherv call to send data in the form of an array of integers to all processors in the group as a combined array. An array indicating where each processor's data should be placed in the destination array is required. Note that this subroutine can be used to distribute data to all available processors by using the all-node communicator. (The serial version of this subroutine directly copies the array for sending data into the array for receiving data.)

Parameters

send-data	Array of integers being sent to all processors
send-count	Total number of integers being sent by current processor
recv-data	Array of integers with values arriving from all processors
recv-count	Array containing total numbers of integers arriving from each processor
displ	Array indicating where data from each processor should be placed in data-receiving array (as a displacement from the start)
group	Communicator for group of processors

distribute_int_data_root()

```
subroutine comms_module::distribute_int_data_root (integer, dimension (:),␣
↪intent(in) senddata,
                                                    integer, intent(in) sendcount,
                                                    integer, dimension (:),␣
↪intent(inout) recvdata,
                                                    integer, dimension (:),␣
↪intent(in) recvcount,
                                                    integer, dimension (:),␣
↪intent(in) displ,
                                                    integer, intent(in) group
                                                   )
```

Applies a single MPI_Gatherv call to send data in the form of an array of integers to the root processor for the given group as a single combined array. An array indicating where each processor's data should be placed in the destination array is required. Note that this subroutine can be used to distribute data from all available processors by using the all-node communicator. (The serial version of this subroutine directly copies the array for sending data into the array for receiving data.)

Parameters

send-data	Array of integers being sent to all processors
send-count	Total number of integers being sent by current processor
recv-data	Array of integers with values arriving from all processors (only needs to be allocated for root processor)
recv-count	Array containing total numbers of integers arriving at root processor from each processor
displ	Array indicating where data from each processor should be placed in data-receiving array (as a displacement from the start)
group	Communicator for group of processors

distribute_sca_dble_data()

```
subroutine comms_module::distribute_sca_dble_data (real(kind=dp), dimension (:),␣
↪intent(in) senddata,
                                                   real(kind=dp), dimension (:),␣
↪intent(inout) recvdata,
                                                   integer, intent(in) nodes,
                                                   integer, intent(in) group
                                                  )
```

Applies a series of MPI_Gather calls to scatter an array of double precision real numbers from each processor to all processors in a group, such that each processor will receive a single value from each sending processor, and gather together the received values into an array. The data in the array being scattered by each processor should be in processor order for the group, also noting that this subroutine can be used for all processors by using the all-node communicator. (The serial version of this subroutine directly copies the array for sending data into the array for receiving data.)

Parameters

send-data	Array of double precision real numbers being scattered to all processors
recv-data	Array of double precision real numbers with values arriving from all processors
nodes	Total number of processors in group involved in data distribution (equal to number of integers being sent and received by all processors)
group	Communicator for group of processors

distribute_sca_dble_data_all()

```
subroutine comms_module::distribute_sca_dble_data_all (real(kind=dp), intent(in)␣
↪senddata,
                                                       real(kind=dp), dimension␣
↪(:), intent(inout) recvdata,
                                                       integer, intent(in) group
                                                      )
```

Applies an MPI_Allgather call to gather together individual double precision real numbers from each processor into an array of values that is distributed to all processors. Note that this subroutine can be used for all processors by using the all-node communicator. (The serial version of this subroutine directly copies the array for sending data into the array for receiving data.)

Parameters

senddata	Double precision real number being sent to all processors for gathering together
recvdata	Array of double precision real numbers with gathered values
group	Communicator for group of processors

distribute_sca_dble_data_root()

```
subroutine comms_module::distribute_sca_dble_data_root (real(kind=dp), intent(in)␣
↪senddata,
                                                         real(kind=dp), dimension␣
↪(:), intent(inout) recvdata,
                                                         integer, intent(in) group
                                                        )
```

Applies an MPI_Gather call to gather together individual double precision real numbers from each processor to the group's root processor, which will gather together the received values into an array. Note that this subroutine can be used for all processors by using the all-node communicator. (The serial version of this subroutine directly copies the array for sending data into the array for receiving data.)

Parameters

send-data	Double precision real number being sent to root processor
recv-data	Array of double precision real numbers with values arriving from all processors (only needs to be allocated for root processor)
group	Communicator for group of processors

distribute_sca_int_data()

```
subroutine comms_module::distribute_sca_int_data (integer, dimension (:),␣
↪intent(in) senddata,
                                                   integer, dimension (:),␣
↪intent(inout) recvdata,
                                                   integer, intent(in) nodes,
                                                   integer, intent(in) group
                                                  )
```

Applies a series of MPI_Gather calls to scatter an array of integers from each processor to all processors in a group, such that each processor will receive a single integer value from each sending processor, and gather together the received values into an array. The data in the array being scattered by each processor should be in processor order for the group, also noting that this subroutine can be used for all processors by using the all-node communicator. (The serial version of this subroutine directly copies the array for sending data into the array for receiving data.)

Parameters

send-data	Array of integers being scattered to all processors
recv-data	Array of integers with values arriving from all processors
nodes	Total number of processors in group involved in data distribution (equal to number of integers being sent and received by all processors)
group	Communicator for group of processors

distribute_sca_int_data_all()

```
subroutine comms_module::distribute_sca_int_data_all (integer, intent(in) senddata,
                                                       integer, dimension (:),␣
↪intent(inout) recvdata,
                                                       integer, intent(in) group
                                                      )
```

Applies an MPI_Allgather call to gather together single integer values from each processor into an array of values that is distributed to all processors. Note that this subroutine can be used for all processors by using the all-node communicator. (The serial version of this subroutine directly copies the array for sending data into the array for receiving data.)

Parameters

senddata	Integer being sent to all processors for gathering together
recvdata	Array of integers with gathered values
group	Communicator for group of processors

distribute_sca_int_data_root()

```
subroutine comms_module::distribute_sca_int_data_root (integer, intent(in)␣
↪senddata,
                                                        integer, dimension (:),␣
↪intent(inout) recvdata,
                                                        integer, intent(in) group
                                                       )
```

Applies an MPI_Gather call to gather together single integer values from each processor to the group's root processor, which will gather together the received values into an array. Note that this subroutine can be used for all processors by using the all-node communicator. (The serial version of this subroutine directly copies the array for sending data into the array for receiving data.)

Parameters

send-data	Integer being sent to root processor
recv-data	Array of integers with values arriving from all processors (only needs to be allocated for root processor)
group	Communicator for group of processors

duplicate_group_comms()

```
subroutine comms_module::duplicate_group_comms (integer, intent(inout) comm1,
                                                 integer, intent(inout) comm2
                                                )
```

Duplicate the provided MPI communicator for a group to produce a new copy. (Dummy subroutine in serial.)

Parameters

comm1	Communicator to be copied
comm2	Newly copied communicator

exitcomms()

```
subroutine comms_module::exitcomms
```

Synchronises all processors and calls MPI routine to close all communications to end the DL_MESO_DPD run after a successful calculation. (Dummy subroutine in serial.)

global_and()

```
subroutine comms_module::global_and (logical, dimension (:) iii,
                                     integer nnn,
                                     integer nod,
                                     integer idnode
                                    )
```

Carries out an MPI_reduce operation on a logical array to find AND for all elements across all processors and send the result to a single processor. (Dummy subroutine in serial.)

Parameters

iii	Logical array with which to find global AND
nnn	Size of logical array
nod	Destination processor for result
idnode	Current processor number

global_and_all()

```
subroutine comms_module::global_and_all (logical, dimension (:) iii,
                                         integer nnn
                                        )
```

Carries out an MPI_allreduce operation on a logical array to find AND for all elements across all processors and share the result. (Dummy subroutine in serial.)

Parameters

iii	Logical array with which to find global AND
nnn	Size of logical array

global_or()

```
subroutine comms_module::global_or (logical, dimension (:) iii,
                                    integer nnn,
                                    integer nod,
                                    integer idnode
                                   )
```

Carries out an MPI_reduce operation on a logical array to find OR for all elements across all processors and send the result to a single processor. (Dummy subroutine in serial.)

Parameters

iii	Logical array with which to find global OR
nnn	Size of logical array
nod	Destination processor for result
idnode	Current processor number

global_or_all()

```
subroutine comms_module::global_or_all (logical, dimension (:) iii,
                                        integer nnn
                                       )
```

Carries out an MPI_allreduce operation on a logical array to find OR for all elements across all processors and share the result. (Dummy subroutine in serial.)

Parameters

iii	Logical array with which to find global OR
nnn	Size of logical array

global_sca_and()

```
subroutine comms_module::global_sca_and (logical iii,
                                         integer nod,
                                         integer idnode
                                        )
```

Carries out an MPI_reduce operation on a logical scalar to find AND across all processors and send the result to a single processor. (Dummy subroutine in serial.)

Parameters

iii	Logical scalar with which to find global AND
nod	Destination processor for result
idnode	Current processor number

global_sca_and_all()

```
subroutine comms_module::global_sca_and_all (logical iii)
```

Carries out an MPI_allreduce operation on a logical scalar to find AND across all processors and share the result. (Dummy subroutine in serial.)

Parameters

iii	Logical scalar with which to find global AND

global_sca_max_dble()

```
subroutine comms_module::global_sca_max_dble (real(kind=dp) aaa)
```

Carries out an MPI_allreduce operation on a double precision real number to find maximum across all processors and share the result. (Dummy subroutine in serial.)

Parameters

aaa	Double precision real number with which to find global maximum

global_sca_max_int()

```
subroutine comms_module::global_sca_max_int (integer iii)
```

Carries out an MPI_allreduce operation on an integer to find maximum across all processors and share the result. (Dummy subroutine in serial.)

Parameters

iii	Integer with which to find global maximum

global_sca_min_dble()

```
subroutine comms_module::global_sca_min_dble (real(kind=dp) aaa)
```

Carries out an MPI_allreduce operation on a double precision real number to find minimum across all processors and share the result. (Dummy subroutine in serial.)

Parameters

aaa	Double precision real number with which to find global minimum

global_sca_min_int()

```
subroutine comms_module::global_sca_min_int (integer iii)
```

Carries out an MPI_allreduce operation on an integer to find minimum across all processors and share the result. (Dummy subroutine in serial.)

Parameters

iii	Integer with which to find global minimum

global_sca_or()

```
subroutine comms_module::global_sca_or (logical iii,
                                        integer nod,
                                        integer idnode
                                       )
```

Carries out an MPI_reduce operation on a logical scalar to find OR across all processors and send the result to a single processor. (Dummy subroutine in serial.)

Parameters

iii	Logical scalar with which to find global OR
nod	Destination processor for result
idnode	Current processor number

global_sca_or_all()

```
subroutine comms_module::global_sca_or_all (logical iii)
```

Carries out an MPI_allreduce operation on a logical scalar to find OR across all processors and share the result. (Dummy subroutine in serial.)

Parameters

iii	Logical scalar with which to find global OR

global_sum_dble()

```
subroutine comms_module::global_sum_dble (real(kind=dp), dimension (:) aaa,
                                          integer nnn
                                         )
```

Carries out an MPI_allreduce operation on an array of double precision real numbers to find the sums for all elements across all processors and share the result. (Dummy subroutine in serial.)

Parameters

aaa	Double precision real array on which to apply global summation
nnn	Size of double precision real array

global_sum_int()

```
subroutine comms_module::global_sum_int (integer, dimension (:) iii,
                                         integer nnn
                                        )
```

Carries out an MPI_allreduce operation on an array of integers to find the sums for all elements across all processors and share the result. (Dummy subroutine in serial.)

Parameters

iii	Integer array on which to apply global summation
nnn	Size of integer array

global_sum_sca_dble()

```
subroutine comms_module::global_sum_sca_dble (real(kind=dp) aaa)
```

Carries out an MPI_allreduce operation on a double precision real number to find the sum across all processors and share the result. (Dummy subroutine in serial.)

Parameters

aaa	Double precision real number on which to apply global summation

global_sum_sca_int()

```
subroutine comms_module::global_sum_sca_int (integer iii)
```

Carries out an MPI_allreduce operation on an integer to find the sum across all processors and share the result. (Dummy subroutine in serial.)

Parameters

iii	Integer on which to apply global summation

group_gsync()

```
subroutine comms_module::group_gsync (integer, intent(in) group)
```

Pauses running until all processors in the group are synchronised and have reached a given point in the code: needed when the entire group needs to be involved with what happens subsequently. (Dummy subroutine in serial.)

Parameters

group	Communicator for group

group_sum_cmplx_all()

```
subroutine comms_module::group_sum_cmplx_all (complex(kind=dp), dimension (:) aaa,
                                              integer nnn,
                                              integer group
                                             )
```

Carries out an MPI_allreduce operation on an array of double precision complex numbers to find the sum across all processors in a specified group and share the result among the entire group. (Dummy subroutine in serial.)

Parameters

aaa	Double precision complex array on which to apply group-wide summation
nnn	Size of double precision complex array
group	Communicator for specified group of processors

group_sum_sca_int()

```
subroutine comms_module::group_sum_sca_int (integer, intent(inout) iii,
                                            integer, intent(in) group,
                                            integer, intent(in) rank
                                           )
```

Carries out an MPI_reduce operation on an integer to find the sum across all processors in a specified group and send the result to the group's root processor. (Dummy subroutine in serial.)

Parameters

iii	Integer on which to apply group-wide summation
group	Communicator for specified group of processors
rank	Rank for processor within the specified group

group_sum_sca_int_all()

```
subroutine comms_module::group_sum_sca_int_all (integer, intent(inout) iii,
                                                integer, intent(in) group
                                               )
```

Carries out an MPI_allreduce operation on an integer to find the sum across all processors in a specified group and share the result among the entire group. (Dummy subroutine in serial.)

Parameters

iii	Integer on which to apply group-wide summation
group	Communicator for specified group of processors

gsync()

```
subroutine comms_module::gsync
```

Pauses running until all processors are synchronised and have reached a given point in the code: needed when all processors need to be involved with what happens subsequently. (Dummy subroutine in serial.)

initcomms()

```
subroutine comms_module::initcomms
```

Starts the Message Passing Interface (MPI), each processor establishes a communicator for itself and for all processors involved in the instance of DL_MESO_DPD, works out the sizes of real and integer kinds and assigns MPI kinds for real and complex numbers. (In serial, only the sizes of real and integer kinds are determined.)

io_ansi_open()

```
subroutine comms_module::io_ansi_open (integer, intent(in) comm,
                                       character(len=*), intent(in) name,
                                       integer, intent(inout) handle,
                                       integer, intent(in) ioport
                                      )
```

Open an ANSI text file that can be accessed by a group of writing processors identified by an MPI communicator, creating a handle to identify the file when using MPI-IO (part of MPI-2 and later). If running in serial, this subroutine opens a stream formatted (text) file using a standard Fortran I/O channel (not used in parallel) and sets the handle to the value of this channel.

Parameters

comm	MPI communicator for writing group of processors
name	Name of file being opened
handle	Handle or information object used for MPI-IO calls
ioport	Fortran I/O channel used for serial running

io_close()

```
subroutine comms_module::io_close (integer, intent(inout) handle)
```

The file writing handle is used to close the specified output file (either binary or ANSI text).

Parameters

handle	Handle used for MPI-IO calls (parallel) or to identify Fortran I/O channel (serial)

io_open()

```
subroutine comms_module::io_open (integer, intent(in) comm,
                                  character(len=*), intent(in) name,
                                  integer, intent(inout) handle,
                                  integer, intent(in) ioport
                                 )
```

Open a stream binary file that can be accessed by a group of writing processors identified by an MPI communicator, creating a handle to identify the file when using MPI-IO (part of MPI-2 and later). If running in serial, this subroutine opens a stream unformatted (binary) file using a standard Fortran I/O channel (not used in parallel) and sets the handle to the value of this channel.

Parameters

comm	MPI communicator for writing group of processors
name	Name of file being opened
handle	Handle or information object used for MPI-IO calls
ioport	Fortran I/O channel used for serial running

io_write_batch_char()

```
subroutine comms_module::io_write_batch_char (integer, intent(in) handle,
                                              integer(kind=li), intent(in) offset,
                                              character(len=*), intent(in)␣
↪charwrite,
                                              integer, intent(in) writecount
                                             )
```

Writes a contiguous series (stream) of characters (string) to an opened ANSI text file identified by the file-writing handle (equivalent to the I/O channel in serial) starting at an offset from the beginning of the file given in bytes: the first character in the file is 1.

Parameters

handle	Handle used for MPI-IO calls (parallel) or to identify Fortran I/O channel (serial)
offset	Starting location for string to be written as byte number in file
charwrite	Characters (string) to be written to file
writecount	Number of characters to write to file

io_write_batch_dble()

```
subroutine comms_module::io_write_batch_dble (integer, intent(in) handle,
                                              integer(kind=li), intent(in) offset,
                                              real(kind=dp), dimension (:),␣
↪intent(in) dblewrite,
                                              integer, intent(in) writecount
                                             )
```

Writes a contiguous series (stream) of double precision real numbers to an opened binary file identified by the file-writing handle (equivalent to the I/O channel in serial) starting at an offset from the beginning of the file given in bytes: the first character in the file is 1.

Parameters

handle	Handle used for MPI-IO calls (parallel) or to identify Fortran I/O channel (serial)
offset	Starting location for double precision real numbers to be written as byte number in file
dblewrite	Double precision real numbers to be written to file
writecount	Number of double precision real numbers to write to file

io_write_batch_int()

```
subroutine comms_module::io_write_batch_int (integer, intent(in) handle,
                                             integer(kind=li), intent(in) offset,
                                             integer, dimension (:), intent(in)␣
↪intwrite,
                                             integer, intent(in) writecount
                                            )
```

Writes a contiguous series (stream) of integers to an opened binary file identified by the file-writing handle (equivalent to the I/O channel in serial) starting at an offset from the beginning of the file given in bytes: the first character in the file is 1.

Parameters

handle	Handle used for MPI-IO calls (parallel) or to identify Fortran I/O channel (serial)
offset	Starting location for integers to be written as byte number in file
intwrite	Integers to be written to file
writecount	Number of integers to write to file

io_write_dble()

```
subroutine comms_module::io_write_dble (integer, intent(in) handle,
                                        integer(kind=li), intent(in) offset,
                                        real(kind=dp), intent(in) dblewrite
                                       )
```

Writes a single double precision real number to an opened binary file identified by the file-writing handle (equivalent to the I/O channel in serial) at an offset from the beginning of the file given in bytes: the first character in the file is 1.

Parameters

handle	Handle used for MPI-IO calls (parallel) or to identify Fortran I/O channel (serial)
offset	Starting location for double precision real number to be written as byte number in file
dblewrite	Double precision real number to be written to file

io_write_int()

```
subroutine comms_module::io_write_int (integer, intent(in) handle,
                                       integer(kind=li), intent(in) offset,
                                       integer, intent(in) intwrite
                                      )
```

Writes a single integer to an opened binary file identified by the file-writing handle (equivalent to the I/O channel in serial) at an offset from the beginning of the file given in bytes: the first character in the file is 1.

Parameters

handle	Handle used for MPI-IO calls (parallel) or to identify Fortran I/O channel (serial)
offset	Starting location for integer to be written as byte number in file
intwrite	Integer to be written to file

io_write_longint()

```
subroutine comms_module::io_write_longint (integer, intent(in) handle,
                                           integer(kind=li), intent(in) offset,
                                           integer(kind=li), intent(in)␣
↪longintwrite
                                          )
```

Writes a single long integer to an opened binary file identified by the file-writing handle (equivalent to the I/O channel in serial) at an offset from the beginning of the file given in bytes: the first character in the file is 1.

Parameters

handle	Handle used for MPI-IO calls (parallel) or to identify Fortran I/O channel (serial)
offset	Starting location for long integer to be written as byte number in file
l ongintwrite	Long integer to be written to file

msg_receive_blocked()

```
subroutine comms_module::msg_receive_blocked (integer msgtag,
                                              real(kind=dp), dimension (:) buf,
                                              integer length
                                             )
```

Receives an array of double precision real numbers from any processor as part of an MPI blocking send/receive call, i.e. the code has to wait until the array is received. (Dummy subroutine in serial.)

Parameters

msgtag	MPI message tag associated with send/receive
buf	Received array of double precision real numbers
length	Size of incoming array in bytes

msg_receive_blocked_pe()

```
subroutine comms_module::msg_receive_blocked_pe (integer msgtag,
                                                  real(kind=dp), dimension (:) buf,
                                                  integer length,
                                                  integer pe
                                                 )
```

Receives an array of double precision real numbers from a specific processor as part of an MPI blocking send/receive call, i.e. the code has to wait until the array is received.

Parameters

msgtag	MPI message tag associated with send/receive
buf	Received array of double precision real numbers
length	Size of incoming array in bytes
pe	Source processor of incoming array

msg_receive_sca_blocked()

```
subroutine comms_module::msg_receive_sca_blocked (integer msgtag,
                                                   real(kind=dp) buf,
                                                   integer length
                                                  )
```

Receives one double precision real number from any processor as part of an MPI blocking send/receive call, i.e. the code has to wait until the array is received. (Dummy subroutine in serial.)

Parameters

msgtag	MPI message tag associated with send/receive
buf	Received double precision real number
length	Size of incoming number in bytes

msg_receive_sca_blocked_pe()

```
subroutine comms_module::msg_receive_sca_blocked_pe (integer msgtag,
                                                      real(kind=dp) buf,
                                                      integer length,
                                                      integer pe
                                                     )
```

Receives one double precision real number from a specific processor as part of an MPI blocking send/receive call, i.e. the code has to wait until the array is received.

Parameters

msgtag	MPI message tag associated with send/receive
buf	Received double precision real number
length	Size of incoming number in bytes
pe	Source processor of incoming number

msg_receive_unblocked()

```
integer function comms_module::msg_receive_unblocked (integer msgtag,
                                                       real(kind=dp), dimension (:) ␣
↪buf,
                                                       integer length
                                                      )
```

Receives an array of double precision real numbers from any processor as part of an MPI non-blocking send/receive call, i.e. the code does not have to wait until the array is received but the MPI message has to be explicitly requested using the number returned by this function. (Dummy subroutine in serial.)

Parameters

msgtag	MPI message tag associated with send/receive
buf	Received array of double precision real numbers
length	Size of incoming array in bytes

msg_receive_unblocked_grid_pe()

```
integer function comms_module::msg_receive_unblocked_grid_pe (integer msgtag,␣
↪real(kind=dp), dimension (x0:x1,y0:y1,z0:z1) buf,
                                                              integer x0,
                                                              integer x1,
                                                              integer y0,
                                                              integer y1,
                                                              integer z0,
                                                              integer z1,
                                                              integer length,
                                                              integer pe
                                                             )
```

Receives a three-dimensional array of double precision real numbers from a specific processor as part of an MPI non-blocking send/receive call, i.e. the code does not have to wait until the array is received but the MPI message has to be explicitly requested using the number returned by this function.

Parameters

msgtag	MPI message tag associated with send/receive
buf	Received three-dimensional array of double precision real numbers
x0	Minimum extent of x in array
x1	Maximum extent of x in array
y0	Minimum extent of y in array
y1	Maximum extent of y in array
z0	Minimum extent of z in array
z1	Maximum extent of z in array
length	Size of incoming array in bytes
pe	Source processor of incoming array

msg_receive_unblocked_pe()

```
integer function comms_module::msg_receive_unblocked_pe (integer msgtag,
                                                          real(kind=dp), dimension␣
↪(:) buf,
                                                          integer length,
                                                          integer pe
                                                         )
```

Receives an array of double precision real numbers from a specific processor as part of an MPI non-blocking send/receive call, i.e. the code does not have to wait until the array is received but the MPI message has to be explicitly requested using the number returned by this function. (Dummy subroutine in serial.)

Parameters

msgtag	MPI message tag associated with send/receive
buf	Received array of double precision real numbers
length	Size of incoming array in bytes
pe	Source processor of incoming array

msg_send_blocked()

```
subroutine comms_module::msg_send_blocked (integer msgtag,
                                           real(kind=dp), dimension (:) buf,
                                           integer length,
                                           integer pe
                                          )
```

Sends an array of double precision real numbers to a specific processor as part of an MPI blocking send/receive call, i.e. the code has to wait until the array is received. (Dummy subroutine in serial.)

Parameters

msgtag	MPI message tag associated with send/receive
buf	Sent array of double precision real numbers
length	Size of outgoing array in bytes
pe	Destination processor of outgoing array

msg_send_blocked_grid()

```
subroutine comms_module::msg_send_blocked_grid (integer msgtag,
                                                real(kind=dp), dimension (x0:x1,
↪y0:y1,z0:z1) buf,
                                                integer x0,
                                                integer x1,
                                                integer y0,
                                                integer y1,
                                                integer z0,
                                                integer z1,
                                                integer length,
                                                integer pe
                                               )
```

Sends a three-dimensional array of double precision real numbers from a specific processor as part of an MPI blocking send/receive call, i.e. the code has to wait until the array is received. (Dummy subroutine in serial.)

Parameters

msgtag	MPI message tag associated with send/receive
buf	Sent three-dimensional array of double precision real numbers
x0	Minimum extent of x in array
x1	Maximum extent of x in array
y0	Minimum extent of y in array
y1	Maximum extent of y in array
z0	Minimum extent of z in array
z1	Maximum extent of z in array
length	Size of outgoing array in bytes
pe	Destination processor of outgoing array

msg_send_sca_blocked()

```
subroutine comms_module::msg_send_sca_blocked (integer msgtag,
                                               real(kind=dp) buf,
                                               integer length,
                                               integer pe
                                              )
```

Sends an individual double precision real number to a specific processor as part of an MPI blocking send/receive call, i.e. the code has to wait until the number is received. (Dummy subroutine in serial.)

Parameters

msgtag	MPI message tag associated with send/receive
buf	Sent double precision real number
length	Size of outgoing number in bytes
pe	Destination processor of outgoing number

msg_send_unblocked()

```
integer function comms_module::msg_send_unblocked (integer msgtag,
                                                   real(kind=dp), dimension (:)␣
↪buf,
                                                   integer length,
                                                   integer pe
                                                  )
```

Sends an array of double precision real numbers to a specific processor as part of an MPI non-blocking send/receive call, i.e. the code does not have to wait until the array is received but the MPI message has to be explicitly requested using the number returned by this function.

Parameters

msgtag	MPI message tag associated with send/receive
buf	Received array of double precision real numbers
length	Size of incoming array in bytes
pe	Destination processor of outgoing array

msg_wait()

```
subroutine comms_module::msg_wait (integer request)
```

As part of an MPI non-blocking send/receive call, this routine waits for a specified MPI request obtained from a non-blocking send or receive to complete before continuing. (Dummy subroutine in serial.)

Parameters

request	MPI call request number

msg_wait_and_size_double()

```
integer function comms_module::msg_wait_and_size_double (integer request)
```

As part of an MPI non-blocking send/receive call, this function waits for a specified MPI request obtained from a non-blocking send or receive to complete before continuing and returns the message size, i.e. the number of values in the (presumed) double precision real array.

Parameters

request	MPI call request number

mynode()

```
integer function comms_module::mynode
```

Finds rank (number) of current processor, which can range from 0 to the number of processors less 1. (The serial version of this function always returns 0.)

numnodes()

```
integer function comms_module::numnodes
```

Finds the total number of processors available for DL_MESO_DPD to run. (The serial version of this function always returns 1.)

timchk()

```
subroutine comms_module::timchk (integer ktim,
                                 real(kind=dp) time
                                )
```

This subroutine is used to determine the time elapsed since the start of the calculation: the first call initiates the timer and subsequent calls return the number of second elapsed since then. If the input parameter is greater than 0, a statement giving the elapsed time will be printed to either the OUTPUT file or the standard output (if the ‘l_scr’ option in CONTROL is invoked). The parallel version of this subroutine uses MPI wall time, while the serial version uses a generic Fortran system clock call.

Parameters

ktim	Flag for printing time to OUTPUT or standard output (greater than 0 to activate)
time	Number of seconds elapsed since first call to subroutine to start timer

10.7.5 Variable Documentation

cx_mpi

```
integer save comms_module::cx_mpi = 0
```

Integer with the MPI kind value for double precision complex numbers, with the initial default value of 0 replaced only during *initcomms()* when running in parallel.

dlen

```
integer save comms_module::dlen = 0
```

Integer with the Fortran kind value for double precision real numbers, with the initial default value of 0 replaced during *initcomms()*.

dlen_li

```
integer (kind=li), save comms_module::dlen_li = 0
```

Long integer with the Fortran kind value for double precision real numbers, with the initial default value of 0 replaced during *initcomms()*.

dlm_comm_world

```
integer save comms_module::dlm_comm_world = 0
```

Variable with number of all-node MPI communicator for the current instance of DL_MESO_DPD: the initial default value of 0 (retained in the serial version) is replaced in the parallel version during *initcomms()* with the value of `MPI_COMM_WORLD`.

dp_mpi

```
integer save comms_module::dp_mpi = 0
```

Integer with the MPI kind value for double precision real numbers, with the initial default value of 0 replaced only during *initcomms()* when running in parallel.

ilen

```
integer save comms_module::ilen = 0
```

Integer with the Fortran kind value for standard integers, with the initial default value of 0 replaced during *init-comms()*.

ilen_li

```
integer (kind=li), save comms_module::ilen_li = 0
```

Long integer with the Fortran kind value for standard integers, with the initial default value of 0 replaced during *initcomms()*.

lilen

```
integer save comms_module::lilen = 0
```

Integer with the Fortran kind value for long integers, with the initial default value of 0 replaced during *initcomms()*.

lilen_li

```
integer (kind=li), save comms_module::lilen_li = 0
```

Long integer with the Fortran kind value for long integers, with the initial default value of 0 replaced during *initcomms()*.

self_comm

```
integer save comms_module::self_comm = 0
```

Variable with number of single node MPI communicator for the current instance of DL_MESO_DPD: the initial default value of 0 (retained in the serial version) is replaced in the parallel version during *initcomms()* with the value of `MPI_COMM_SELF`.

10.8 config_module.F90

10.8.1 Summary

Module to set up simulation by reading in system data and determining simulation properties.

10.8.2 Functions/Subroutines

- `subroutine` *sysdef()*

  Reads in system data to specify simulation, writes information to OUTPUT or standard output and sets up main arrays.

- `subroutine` *elecgen()*

  Sets up electrostatic parameters for Ewald self-interaction and charged system corrections.

- `logical function` *fft_length_ok()*

  Checks magnitude of maximum reciprocal vector for FFT solvers.

- `subroutine` *free_memory()*

  Deallocates all arrays at the end of the simulation.

- `subroutine` *zero()*

  Initialises counters, system parameters etc. before starting calculation.

10.8.3 Function/Subroutine Documentation

elecgen()

```
subroutine config_module::elecgen
```

Based on total numbers of charged particles, calculates system-wide corrections to potential energies for Ewald self-interactions of charges and for systems that are not charge neutral. This subroutine also checks for valid maximum reciprocal vectors and adjusts these upwards for SPME calculations if any component does not factorise into powers of 2, 3 or 5.

fft_length_ok()

```
logical function config_module::fft_length_ok (integer kmax)
```

Checks the magnitude of a supplied maximum reciprocal space extent to see if it factorises into powers of 2, 3 and 5, as required for available Fast Fourier Transform (FFT) solvers used for Smooth Particle Mesh Ewald (SPME). Returns true if the value can be factorised by at least one of these values, false if it cannot.

Parameters

kmax	Maximum reciprocal vector being tested

free_memory()

```
subroutine config_module::free_memory
```

Frees up memory used by DL_MESO_DPD during the calculation for arrays, both those used directly during calculations and those specifying simulation properties.

sysdef()

```
subroutine config_module::sysdef (logical l_config,
                                  logical l_rest
                                 )
```

Reads in system data, determines simulation properties (including how it will be run on available processor nodes), allocates main arrays for calculations and prints information to OUTPUT file or standard output. This subroutine calculates maximum array sizes for numbers of particles, pairwise interactions and communication buffers based on supplied total numbers of particles, link cells and available processor nodes.

Parameters

l_config	Flag to determine if CONFIG file is to be used to provide initial configuration
l_rest	Flag to determine if restarting a previous simulation using an export file (and REVIVE file)

zero()

```
subroutine config_module::zero
```

Sets step counters, initial time parameters, system parameters, accumulators for statistical properties, long-range potential corrections and particle properties (forces, velocities, positions) to zero, and initialises random number generators using supplied seed.

10.9 domain_module.F90

10.9.1 Summary

Module to specify subdomains for each processor node, establish linked cell lists and subroutines to carry out node-to-node communications. (A serial version of this module is available as domain_module_ser.F90, which includes some modified versions of the subroutines and excludes others that are not necessary for running on a single processor.)

10.9.2 Functions/Subroutines

- `subroutine` *domain_decompose()*

  Determines how system is divided up among processor nodes.

- `subroutine` *domain_dimensions()*

  Determines the dimensions of the subdomain and link cells.

- `subroutine` *parlnk()*

  Constructs parallel link cells.

- `subroutine` *resize_buffer()*

  Resizes arrays for communication transfer buffers between processor nodes.

- `subroutine` *deport()*

  Deports particles outside subdomain to neighbouring nodes.

- `subroutine` *deport_shear()*

  Deports particles outside subdomain to appropriate nodes for Lees-Edwards shearing boundaries.

- `subroutine` *import()*

  Imports forces for particles in boundary halos back to original processor nodes.

- `subroutine` *import_shear()*

  Imports forces for particles in boundary halos back to original processor nodes for Lees-Edwards shearing boundaries.

- `subroutine` *importvariable()*

  Imports two sets of forces for particles in boundary halos back to original processor nodes.

- `subroutine` *importvariable_shear()*

  Imports two sets of forces for particles in boundary halos back to original processor nodes for Lees-Edwards shearing boundaries.

- `subroutine` *export*

  Exports particle data to neighbouring nodes or across box for boundary halo creation.

- `subroutine` *export_shear()*

  Exports particle data to neighbouring nodes or across box for boundary halo creation when using Lees-Edwards boundaries.

- `subroutine` *exportvelocity()*

  Exports particle data to neighbouring nodes or across box for updates to positions or velocities in boundary halo.

- `subroutine` *exportvelocity_shear()*

  Exports particle data to neighbouring nodes or across box for updates to positions or velocities in boundary halo when using Lees-Edwards boundaries.

- `subroutine` *exportdensity()*

  Exports particle data to neighbouring nodes or across box for boundary halo used in many-body DPD density calculations.

- `subroutine` *exportdensity_shear()*

  Exports particle data to neighbouring nodes or across box for boundary halo used in many-body DPD density calculations when using Lees-Edwards boundaries.

- `subroutine` *deportdata()*

  Applies particle deport to neighbouring domains and/or periodic boundary conditions.

- `subroutine` *importdata()*

  Applies particle import from neighbouring domains and/or across periodic boundaries.

- `subroutine` *importdata_dpdvv()*

  Applies particle import from neighbouring domains and/or across periodic boundaries for DPD Velocity Verlet.

- `subroutine` *importdata_stoyanov()*

  Applies particle import from neighbouring domains and/or across periodic boundaries for Stoyanov-Groot.

- `subroutine` *exportdata()*

  Applies particle export from neighbouring domains and/or across periodic boundaries.

- `subroutine` *exportvelocitydata()*

  Applies particle export from neighbouring domains and/or across periodic boundaries to update particle positions or velocities.

- `subroutine` *exportdensitydata()*

  Applies particle export from neighbouring domains and/or across periodic boundaries for many-body DPD localised density calculations.

10.9.3 Variables

- `integer, parameter bufnum`

  Maximum number of communication buffers.

10.9.4 Function/Subroutine Documentation

deport()

```
subroutine domain_module::deport (integer nlimit,
                                  integer mdir,
                                  integer mp_send,
                                  integer mp_recv,
                                  real(kind=dp) begin,
                                  real(kind=dp) final,
                                  real(kind=dp) shove,
                                  logical skip
                                 )
```

Moves particles that currently exist outside volume for subdomain (after first stage of Velocity Verlet integration) to neighbouring processor nodes. All data including any relevant bond, angle and dihedral information are transferred for each particle. A switch is available to prevent particles being transferred when using non-periodic boundary conditions (e.g. hard surfaces, Lees-Edwards shearing). This subroutine is only needed for the parallel version of DL_MESO_DPD, as no particles need to be transferred when only using a single processor node.

Parameters

nlimit	Number of particles in subdomain (subjected to change during routine)
mdir	Dimension to search for particles outside subdomain (1 = x, 2 = y, 3 = z)
mp_send	Destination processor node for sending particle data
mp_recv	Source processor node for receiving particle data
begin	Smallest coordinate in given direction for particle to be sent to neighbouring node
final	Largest coordinate in given direction for particle to be sent to neighbouring node
shove	Required shift for coordinate in given direction when particle is sent to neighbouring node
skip	Switch to specify whether or not particles should be omitted from being moved in current direction

deport_shear()

```
subroutine domain_module::deport_shear (integer nlimit,
                                        integer mdir,
                                        integer shft,
                                        real(kind=dp) begin,
                                        real(kind=dp) final,
                                        real(kind=dp) shove,
                                        real(kind=dp) shove1,
                                        real(kind=dp) shove2,
                                        real(kind=dp) vshove1,
                                        real(kind=dp) vshove2,
                                        real(kind=dp) side1,
                                        real(kind=dp) side2
                                       )
```

Moves particles that currently exist outside volume for subdomain (after first stage of Velocity Verlet integration) to processor nodes on opposite side of simulation box that enable position shifting required for Lees-Edwards shearing boundaries. All data including any relevant bond, angle and dihedral information are transferred for each particle. This subroutine requires the displacement of particles due to shear and the velocity corrections in both directions orthogonal to the boundary. A switch is included to use different MPI tags if the subroutine is called twice by the same processor node. This subroutine is only needed for the parallel version of DL_MESO_DPD, as no particles need to be transferred when only using a single processor node.

Parameters

nlimit	Number of particles in subdomain (subjected to change during routine)
mdir	Direction to search for particles outside subdomain (1 = -x, 2 = +x, 3 = -y, 4 = +y, 5 = -z, 6 = +z)
shft	Switch to change communication tag if calling more than once
begin	Smallest coordinate in given direction for particle to be sent to destination nodes
final	Largest coordinate in given direction for particle to be sent to destination nodes
shove	Required shift for coordinate in given direction when particle is sent to destination nodes
shove1	Required shift in first orthogonal coordinate when particle is sent to destination nodes
shove2	Required shift in second orthogonal coordinate when particle is sent to destination nodes
vshove1	Required change in velocity for first orthogonal coordinate when particle is sent to destination nodes
vshove2	Required change in velocity in second orthogonal coordinate when particle is sent to destination nodes
side1	Size of subdomain in first orthogonal coordinate
side2	Size of subdomain in second orthogonal coordinate

deportdata()

```
subroutine domain_module::deportdata (integer nlimit)
```

Determines particles that need to be moved to/from current processor, simulataneously applying periodic boundary conditions with and without Lees-Edwards shearing. The serial version of this routine applies the boundary conditions by using mathematical functions on particle positions, as no change in the number of particles is required. This routine is a common interface in DL_MESO_DPD for both parallel and serial running.

Parameters

nlimit	Number of particles currently in subdomain (subjected to change during routine if running in parallel)

domain_decompose()

```
subroutine domain_module::domain_decompose
```

Determines the three-dimensional domain decomposition for the system, including the number of nodes in each direction, the location for each node in the system and the nearest neighbouring nodes required for node-to-node communications.

domain_dimensions()

```
subroutine domain_module::domain_dimensions
```

Determines the size of the subdomain (the same for all processor nodes) and the number and sizes of link cells in each direction for pairwise interactions. This subroutine is called during simulation setup and whenever the system volume changes due to use of a barostat. Different link cells exist for ‘standard’ pairwise forces, electrostatic forces in real space and many-body DPD localised density calculations: the subroutine determines the sizes of linked cell list arrays and which link cells are neighbours to each cell in the subdomain.

export()

Parallel version

```
subroutine domain_module::export (integer nlimit,
                                  integer mdir,
                                  integer mp_send,
                                  integer mp_recv,
                                  real(kind=dp) begin,
                                  real(kind=dp) final,
                                  real(kind=dp) shove,
                                  logical skip,
                                  logical listadd,
                                  integer bmove
                                 )
```

Serial version

```
subroutine domain_module::export (integer nlimit,
                                  integer mdir,
                                  real(kind=dp) begin,
                                  real(kind=dp) final,
                                  real(kind=dp) shove
                                 )
```

Sends particle positions, velocities and (if using many-body DPD) localised densities to neighbouring nodes or to opposite side of simulation volume to create boundary halos for force calculations, particularly those involving pairwise interactions. Two switches are available: one to prevent particle data being transferred across non-periodic boundaries (i.e. hard surfaces, Lees-Edwards shearing boundaries), the other to add any incoming particles to the global/local particle index list for bonded interactions. In parallel, this subroutine also returns the number of particles received to check and adjust buffer array sizes for future time steps.

Parameters

nlimit	Number of particles in subdomain and boundary halo (subjected to change during routine)
mdir	Dimension to send particles from inside subdomain (1 = x, 2 = y, 3 = z)
mp_send	Destination processor node for sending particle data
mp_recv	Source processor node for receiving particle data
begin	Smallest coordinate in given direction for particle data to be sent to neighbouring node/across box
final	Largest coordinate in given direction for particle data to be sent to neighbouring node/across box
shove	Required shift for coordinate in given direction when particle data are sent to neighbouring node/across box
skip	Switch to specify whether or not particles should be omitted from being moved in current direction
listadd	Switch to specify whether or not incoming particles should be added to global/local particle index list for bonded interactions
bmove	Number of particles received during communication to check and adjust buffer array sizes

export_shear()

Parallel version

```
subroutine domain_module::export_shear (integer nlimit,
                                        integer mdir,
                                        integer shft,
                                        real(kind=dp) begin,
                                        real(kind=dp) final,
                                        real(kind=dp) shove,
                                        real(kind=dp) shove1,
                                        real(kind=dp) shove2,
```

(continues on next page)

(continued from previous page)

```
                                                   real(kind=dp) vshove1,
                                                   real(kind=dp) vshove2,
                                                   real(kind=dp) side1,
                                                   real(kind=dp) side2,
                                                   logical listadd,
                                                   integer bmove
                                                  )
```

Serial version

```
subroutine domain_module::export_shear (integer nlimit,
                                        integer mdir,
                                        real(kind=dp) begin,
                                        real(kind=dp) final,
                                        real(kind=dp) shove,
                                        real(kind=dp) shove1,
                                        real(kind=dp) shove2,
                                        real(kind=dp) vshove1,
                                        real(kind=dp) vshove2,
                                        real(kind=dp) side1,
                                        real(kind=dp) side2
                                       )
```

Sends particle positions, velocities and (if using many-body DPD) localised densities - either (in parallel) to processor nodes on opposite side of simulation box that enable position shifting required for Lees-Edwards shearing boundaries, or (in serial) to opposite side of simulation box while applying position shifting required for Lees-Edwards shearing boundaries - to create boundary halos for force calculations, particularly those involving pairwise interactions. This subroutine requires the displacement of particles due to shear and the velocity corrections in both directions orthogonal to the boundary: the velocities of particles entering the boundary halo are adjusted to remove the discontinuity in relative velocities that would otherwise exist [77]. Two switches are available for parallel running: one to use different MPI tags if the subroutine is called twice by the same processor node, the other to add any incoming particles to the global/local particle index list for bonded interactions. The parallel version of this subroutine also returns the number of particles received to check and adjust buffer array sizes for future time steps.

Parameters

nlimit	Number of particles in subdomain (subjected to change during routine)
mdir	Direction to send particles from inside subdomain (1 = -x, 2 = +x, 3 = -y, 4 = +y, 5 = -z, 6 = +z)
shft	Switch to change communication tag if calling more than once
begin	Smallest coordinate in given direction for particle to be sent to destination nodes/across box
final	Largest coordinate in given direction for particle to be sent to destination nodes/across box
shove	Required shift for coordinate in given direction when particle is sent to destination nodes/across box
shove1	Required shift in first orthogonal coordinate when particle is sent to destination nodes/across box
shove2	Required shift in second orthogonal coordinate when particle is sent to destination nodes/across box
vshove1	Required change in velocity for first orthogonal coordinate when particle is sent to destination nodes/ across box
vshove2	Required change in velocity in second orthogonal coordinate when particle is sent to destination nodes/ across box
side1	Size of subdomain in first orthogonal coordinate
side2	Size of subdomain in second orthogonal coordinate
listadd	Switch to specify whether or not incoming particles should be added to global/local particle index list for bonded interactions
bmove	Number of particles received during communication to check and adjust buffer array sizes

exportdata()

```
subroutine domain_module::exportdata (integer nlimit)
```

Sends and receives particle data from neighbouring processors - including those used for Lees-Edwards shearing boundaries, but excluding other non-periodic boundaries - to create boundary halos for force calculations. The particles sent in each direction are recorded in memory for any subsequent export steps to update particle velocities and positions. The serial version of this routine applies this procedure by creating copies of particles that would be in the boundary halo, so as to allow use of the same force calculation methods for both serial and parallel running. As the most memory intensive form of export step, the parallel version also checks the transfer buffer arrays used to send and receive data are large enough while the system is still equilibrating or if many-body DPD is in use and readjusts the buffers for subsequent timesteps if more than 85% of the allocated memory was in use. This subroutine is a common interface in DL_MESO_DPD for both parallel and serial running.

Parameters

nlimit	Number of particles currently in subdomain and boundary halo (subjected to change during routine)

exportdensity()

Parallel version

```
subroutine domain_module::exportdensity (integer nlimit,
                                          integer mdir,
                                          integer mp_send,
                                          integer mp_recv,
                                          real(kind=dp) begin,
                                          real(kind=dp) final,
                                          real(kind=dp) shove,
                                          logical skip
                                         )
```

Serial version

```
subroutine domain_module::exportdensity (integer nlimit,
                                          integer mdir,
                                          real(kind=dp) begin,
                                          real(kind=dp) final,
                                          real(kind=dp) shove
                                         )
```

Sends particle positions to neighbouring nodes or across simulation box to create boundary halos for calculations of localised densities used for many-body DPD interactions. A switch is available in the parallel version to prevent particle data being transferred across non-periodic boundaries (i.e. hard surfaces, Lees-Edwards shearing boundaries).

Parameters

nlimit	Number of particles in subdomain and boundary halo (subjected to change during routine)
mdir	Dimension to send particles from inside subdomain (1 = x, 2 = y, 3 = z)
mp_send	Destination processor node for sending particle data
mp_recv	Source processor node for receiving particle data
begin	Smallest coordinate in given direction for particle data to be sent to neighbouring node/across box
final	Largest coordinate in given direction for particle data to be sent to neighbouring node/across box
shove	Required shift for coordinate in given direction when particle and its force are sent to neighbouring node/ across box
skip	Switch to specify whether or not particles should be omitted from being moved in current direction

exportdensity_shear()

Parallel version

```
subroutine domain_module::exportdensity_shear (integer nlimit,
                                               integer mdir,
                                               integer shft,
                                               real(kind=dp) begin,
                                               real(kind=dp) final,
                                               real(kind=dp) shove,
                                               real(kind=dp) shove1,
                                               real(kind=dp) shove2,
                                               real(kind=dp) side1,
                                               real(kind=dp) side2
                                              )
```

Serial version

```
subroutine domain_module::exportdensity_shear (integer nlimit,
                                               integer mdir,
                                               real(kind=dp) begin,
                                               real(kind=dp) final,
                                               real(kind=dp) shove,
                                               real(kind=dp) shove1,
                                               real(kind=dp) shove2,
                                               real(kind=dp) side1,
                                               real(kind=dp) side2
                                              )
```

Sends particle positions - either (in parallel) to processor nodes on opposite side of simulation box, or (in serial) across simulation box - with position shifting required for Lees-Edwards shearing boundaries, as needed to create boundary halos for calculations of localised densities used for many-body DPD interactions. This subroutine requires the displacement of of particles due to shear in both directions orthogonal to the boundary. A switch is available to use different MPI tags if the subroutine is called twice by the same processor node.

Parameters

nlimit	Number of particles in subdomain and boundary halo (subjected to change during routine)
mdir	Direction to send particles from inside subdomain (1 = -x, 2 = +x, 3 = -y, 4 = +y, 5 = -z, 6 = +z)
shft	Switch to change communication tag if calling more than once
begin	Smallest coordinate in given direction for particle data to be sent to destination nodes/across box
final	Largest coordinate in given direction for particle data to be sent to destination nodes/across box
shove	Required shift for coordinate in given direction when particle is sent to destination nodes/across box
shove1	Required shift in first orthogonal coordinate when particle is sent to destination nodes/across box
shove2	Required shift in second orthogonal coordinate when particle is sent to destination nodes/across box
side1	Size of subdomain in first orthogonal coordinate
side2	Size of subdomain in second orthogonal coordinate

exportdensitydata()

```
subroutine domain_module::exportdensitydata (integer nlimit)
```

Sends and receives particle data from neighbouring processors - including those used for Lees-Edwards shearing boundaries, but excluding other non-periodic boundaries - to create boundary halos for localised density calculations required for many-body DPD interactions. The serial version of this routine applies this procedure by creating copies of particles that would be in the boundary halo, so as to allow use of the same density calculation methods for both serial and parallel running. This subroutine is a common interface in DL_MESO_DPD for both parallel and serial running.

Parameters

nlimit	Number of particles currently in subdomain and boundary halo (subjected to change during routine)

exportvelocity()

Parallel version

```
subroutine domain_module::exportvelocity (integer nlimit,
                                           integer mdir,
                                           integer mp_send,
                                           integer mp_recv,
                                           real(kind=dp) begin,
                                           real(kind=dp) final,
                                           real(kind=dp) shove,
                                           logical skip
                                          )
```

Serial version

```
subroutine domain_module::exportvelocity (integer nlimit,
                                           integer mdir,
                                           real(kind=dp) begin,
                                           real(kind=dp) final
                                          )
```

Sends updated particle positions and velocities to neighbouring nodes or across simulation box as required for re-calculation of dissipative forces for DPD Velocity Verlet and integration of DPD forces with Shardlow splitting. A switch is available in the parallel version to prevent particle data being transferred across non-periodic boundaries (i.e. hard surfaces, Lees-Edwards shearing boundaries).

Parameters

nlimit	Number of particles in subdomain and boundary halo (subjected to change during routine)
mdir	Dimension to send particles from inside subdomain (1 = x, 2 = y, 3 = z)
mp_send	Destination processor node for sending particle data
mp_recv	Source processor node for receiving particle data
begin	Smallest coordinate in given direction for particle to be sent to destination nodes/across box
final	Largest coordinate in given direction for particle to be sent to destination nodes/across box
shove	Required shift for coordinate in given direction when particle is sent to neighbouring node/across box
skip	Switch to specify whether or not particles should be omitted from being moved in current direction

exportvelocity_shear()

Parallel version

```
subroutine domain_module::exportvelocity_shear (integer nlimit,
                                                 integer mdir,
                                                 integer shft,
                                                 real(kind=dp) begin,
                                                 real(kind=dp) final,
                                                 real(kind=dp) shove,
                                                 real(kind=dp) shove1,
                                                 real(kind=dp) shove2,
                                                 real(kind=dp) vshove1,
                                                 real(kind=dp) vshove2,
                                                 real(kind=dp) side1,
                                                 real(kind=dp) side2
                                                )
```

Serial version

```
subroutine domain_module::exportvelocity_shear (integer nlimit,
                                                integer mdir,
                                                real(kind=dp) begin,
                                                real(kind=dp) final,
                                                real(kind=dp) vshove1,
                                                real(kind=dp) vshove2
                                               )
```

Sends updated particle positions and velocities - either (in parallel) to processor nodes on opposite side of simulation box, or (in serial) across simulation box - with position shifting required for Lees-Edwards shearing boundaries, as needed for recalculation of dissipative forces for DPD Velocity Verlet, and integration of DPD forces with Shardlow splitting. This subroutine requires the displacement of of particles due to shear in both directions orthogonal to the boundary: the velocities of particles in the boundary halo are adjusted to remove the discontinuity in relative velocities that would otherwise exist [77]. A switch is available in the parallel version to use different MPI tags if the subroutine is called twice by the same processor node.

Parameters

nlimit	Number of particles in subdomain and boundary halo (subjected to change during routine)
mdir	Direction to send particles from inside subdomain (1 = -x, 2 = +x, 3 = -y, 4 = +y, 5 = -z, 6 = +z)
shft	Switch to change communication tag if calling more than once
begin	Smallest coordinate in given direction for particle to be sent to destination nodes/across box
final	Largest coordinate in given direction for particle to be sent to destination nodes/across box
shove	Required shift for coordinate in given direction when particle is sent to destination nodes/across box
shove1	Required shift in first orthogonal coordinate when particle is sent to destination nodes/across box
shove2	Required shift in second orthogonal coordinate when particle is sent to destination nodes/across box
vshove1	Required change in velocity for first orthogonal coordinate when particle is sent to destination nodes/ across box
vshove2	Required change in velocity in second orthogonal coordinate when particle is sent to destination nodes/ across box
side1	Size of subdomain in first orthogonal coordinate
side2	Size of subdomain in second orthogonal coordinate

exportvelocitydata()

```
subroutine domain_module::exportvelocitydata (integer nlimit)
```

Sends and receives particle data from neighbouring processors - including those used for Lees-Edwards shearing boundaries, but excluding other non-periodic boundaries - to update particle positions or velocities for recalculation of dissipative forces for DPD Velocity Verlet and integration of DPD forces using Shardlow splitting. The serial version of this routine applies this procedure by replacing the positions and velocities of particle copies in the boundary halo. This subroutine is a common interface in DL_MESO_DPD for both parallel and serial running.

Parameters

nlimit	Number of particles currently in subdomain and boundary halo (subjected to change during routine)

import()

```
subroutine domain_module::import (integer nlimit,
                                  integer mdir,
                                  integer mp_send,
                                  integer mp_recv,
                                  real(kind=dp) begin,
                                  real(kind=dp) final,
                                  real(kind=dp) shove,
                                  logical skip
                                 )
```

Sends contributions to forces for particles in boundary halos back to the processor nodes where the particles are located to complete these particles' forces. This version of the subroutine is suitable for thermostats and integration schemes that only require a single set of forces to be calculated (i.e. DPD with MD Velocity Verlet or Shardlow splitting, Lowe-Andersen and Peters). A switch is available to prevent particle forces being transferred when using non-periodic boundary conditions (e.g. hard surfaces, Lees-Edwards shearing). This subroutine is only required for the parallel version of DL_MESO_DPD, as the force contributions will already be available on a single processor node.

Parameters

nlimit	Number of particles in subdomain and boundary halo (subjected to change during routine)
mdir	Dimension to search for particles in subdomain (1 = x, 2 = y, 3 = z)
mp_send	Destination processor node for sending particle data
mp_recv	Source processor node for receiving particle data
begin	Smallest coordinate in given direction for particle forces to be sent to neighbouring node
final	Largest coordinate in given direction for particle forces to be sent to neighbouring node
shove	Required shift for coordinate in given direction when particle and its force are sent to neighbouring node
skip	Switch to specify whether or not particles should be omitted from being moved in current direction

import_shear()

```
subroutine domain_module::import_shear (integer nlimit,
                                        integer mdir,
                                        integer shft,
                                        real(kind=dp) begin,
                                        real(kind=dp) final,
                                        real(kind=dp) shove,
                                        real(kind=dp) shove1,
                                        real(kind=dp) shove2,
                                        real(kind=dp) side1,
                                        real(kind=dp) side2
                                       )
```

Sends contributions to forces for particles in boundary halos back to the processor nodes where the particles are located to complete these particles' forces when Lees-Edwards shearing boundaries are in use. This version of the subroutine is suitable for thermostats and integration schemes that only require a single set of forces to be calculated (i.e. DPD with MD Velocity Verlet or Shardlow splitting, Lowe-Andersen and Peters). This subroutine requires the displacement of particles due to shear in both directions orthogonal to the boundary. A switch is included to use different MPI tags if the subroutine is called twice by the same processor node. This subroutine is only needed for the parallel version of DL_MESO_DPD, as the force contributions will already be available on a single processor node.

Parameters

nlimit	Number of particles in subdomain and boundary halo (subjected to change during routine)
mdir	Direction to search for particles inside subdomain (1 = -x, 2 = +x, 3 = -y, 4 = +y, 5 = -z, 6 = +z)
shft	Switch to change communication tag if calling more than once
begin	Smallest coordinate in given direction for particle to be sent to destination nodes
final	Largest coordinate in given direction for particle to be sent to destination nodes
shove	Required shift for coordinate in given direction when particle is sent to destination nodes
shove1	Required shift in first orthogonal coordinate when particle is sent to destination nodes
shove2	Required shift in second orthogonal coordinate when particle is sent to destination nodes
side1	Size of subdomain in first orthogonal coordinate
side2	Size of subdomain in second orthogonal coordinate

importdata()

```
subroutine domain_module::importdata (integer nlimit)
```

Collects contributions for particle forces in current processor from boundary halos in neighbouring processors, including those used for Lees-Edwards shearing boundaries. The serial version of this routine applies this procedure by simply adding force contributions from particles in the boundary halo to the forces for actual particles: this information is already available. This subroutine is applicable for integrators and thermostats that require a single set of particle forces (i.e. DPD with MD Velocity Verlet, Shardlow splitting, Lowe-Andersen, Peters) and is a common interface in DL_MESO_DPD for both parallel and serial running.

Parameters

nlimit	Number of particles currently in subdomain and boundary halo (subjected to change during routine if running in parallel)

importdata_dpdvv()

```
subroutine domain_module::importdata_dpdvv (integer nlimit)
```

Collects contributions for particle forces in current processor from boundary halos in neighbouring processors, including those used for Lees-Edwards shearing boundaries. The serial version of this routine applies this procedure by simply adding force contributions from particles in the boundary halo to the forces for actual particles: this information is already available. This subroutine is applicable for integrators and thermostats that require two sets of particle forces (i.e. DPD with DPD Velocity Verlet) and is a common interface in DL_MESO_DPD for both parallel and serial running.

Parameters

nlimit	Number of particles currently in subdomain and boundary halo (subjected to change during routine if running in parallel)

importdata_stoyanov()

```
subroutine domain_module::importdata_stoyanov (integer nlimit)
```

Collects contributions for particle forces in current processor from boundary halos in neighbouring processors, including those used for Lees-Edwards shearing boundaries. The serial version of this routine applies this procedure by simply adding force contributions from particles in the boundary halo to the forces for actual particles: this information is already available. This subroutine is applicable for integrators and thermostats that require two sets of particle forces (i.e. Stoyanov-Groot) and is a common interface in DL_MESO_DPD for both parallel and serial running.

Parameters

nlimit	Number of particles currently in subdomain and boundary halo (subjected to change during routine if running in parallel)

importvariable()

```
subroutine domain_module::importvariable (integer nlimit,
                                          integer mdir,
                                          integer mp_send,
                                          integer mp_recv,
                                          real(kind=dp) begin,
                                          real(kind=dp) final,
                                          real(kind=dp) shove,
                                          logical skip
                                         )
```

Sends contributions to forces for particles in boundary halos back to the processor nodes where the particles are located to complete these particles' forces. This version of the subroutine is suitable for thermostats and integration schemes that require two sets of forces to be calculated (i.e. DPD with DPD Velocity Verlet, Stoyanov-Groot). A switch is available to prevent particle forces being transferred when using non-periodic boundary conditions (e.g. hard surfaces, Lees-Edwards shearing). This subroutine is only required for the parallel version of DL_MESO_DPD, as the force contributions will already be available on a single processor node.

Parameters

nlimit	Number of particles in subdomain and boundary halo (subjected to change during routine)
mdir	Dimension to search for particles inside subdomain (1 = x, 2 = y, 3 = z)
mp_send	Destination processor node for sending particle data
mp_recv	Source processor node for receiving particle data
begin	Smallest coordinate in given direction for particle forces to be sent to neighbouring node
final	Largest coordinate in given direction for particle forces to be sent to neighbouring node
shove	Required shift for coordinate in given direction when particle and its force are sent to neighbouring node
skip	Switch to specify whether or not particles should be omitted from being moved in current direction

importvariable_shear()

```
subroutine domain_module::importvariable_shear (integer nlimit,
                                                integer mdir,
                                                integer shft,
                                                real(kind=dp) begin,
                                                real(kind=dp) final,
                                                real(kind=dp) shove,
                                                real(kind=dp) shove1,
                                                real(kind=dp) shove2,
                                                real(kind=dp) side1,
                                                real(kind=dp) side2
                                               )
```

Sends contributions to forces for particles in boundary halos back to the processor nodes where the particles are located to complete these particles' forces when Lees-Edwards shearing boundaries are in use. This version of the subroutine is suitable for thermostats and integration schemes that require two sets of forces to be calculated (i.e. DPD with DPD Velocity Verlet, Stoyanov-Groot). This subroutine requires the displacement of particles due to shear in both directions orthogonal to the boundary. A switch is included to use different MPI tags if the subroutine is called twice by the same processor node. This subroutine is only needed for the parallel version of DL_MESO_DPD, as the force contributions will already be available on a single processor node.

Parameters

nlimit	Number of particles in subdomain and boundary halo (subjected to change during routine)
mdir	Direction to search for particles inside subdomain (1 = -x, 2 = +x, 3 = -y, 4 = +y, 5 = -z, 6 = +z)
shft	Switch to change communication tag if calling more than once
begin	Smallest coordinate in given direction for particle to be sent to destination nodes
final	Largest coordinate in given direction for particle to be sent to destination nodes
shove	Required shift for coordinate in given direction when particle is sent to destination nodes
shove1	Required shift in first orthogonal coordinate when particle is sent to destination nodes
shove2	Required shift in second orthogonal coordinate when particle is sent to destination nodes
side1	Size of subdomain in first orthogonal coordinate
side2	Size of subdomain in second orthogonal coordinate

parlnk()

```
subroutine domain_module::parlnk (integer num1,
                                  integer num2,
                                  integer nx,
                                  integer ny,
                                  integer nz,
                                  real(kind=dp) widthx,
                                  real(kind=dp) widthy,
                                  real(kind=dp) widthz,
                                  integer, dimension (:) cell,
                                  integer, dimension (:) lnk,
                                  integer mxcell,
                                  integer typ
                                 )
```

Constructs the parallel link cells for calculations of pairwise forces (or localised densities) between particles and returns the maximum number of particles per cell. A switch is available for electrostatic interactions that typically act over longer distances solely between charged particles. Each linked-cell list is checked for infinite loops, which are broken if any are found.

Parameters

num1	Starting particle for list construction
num2	Finishing particle for list construction
nx	Number of link cells in x direction
ny	Number of link cells in y direction
nz	Number of link cells in z direction
widthx	Width of link cells in x direction
widthy	Width of link cells in y direction
widthz	Width of link cells in z direction
cell	Array with starting particle in list for each link cell
lnk	Array indicating next particles in linked-cell lists
mxcell	Maximum number of particles per link cell
typ	Switch to indicate if link cells are for Ewald real-space calculations (1) or not (0)

resize_buffer()

```
subroutine domain_module::resize_buffer
```

Deallocates and reallocates the arrays required for node-to-node communications to send and receive particle data. This is only required for the parallel version of DL_MESO: a dummy version of the subroutine is supplied for single-node running.

10.9.5 Variable Documentation

bufnum

```
integer parameter domain_module::bufnum
```

Maximum number of buffers (arrays) used for inter-processor communications. The parallel version of DL_MESO_DPD sets this value to 4 as required for Lees-Edwards shearing boundaries, while the serial version sets it to 0 as no buffers are required when running on a single processor.

10.10 error_module.F90

10.10.1 Summary

Module to print error/warning messages and, if necessary, close down DL_MESO_DPD in a controlled manner.

10.10.2 Functions/Subroutines

- `subroutine` *error()*

  Prints user-friendly error messages in OUTPUT file (or standard output) and closes down DL_MESO_DPD.

10.10.3 Function/Subroutine Documentation

error()

```
subroutine error_module::error (integer idnode,
                                integer iode,
                                integer value
                               )
```

Based on the error code supplied, this subroutine will print either an error message or a warning message to either the OUTPUT file or to standard output if the ‘l_scr’ option is selected in the CONTROL file. If the code indicates an error that cannot be resolved during runtime, DL_MESO_DPD will then close down in a rapid but controlled manner (using MPI_Abort in parallel). If the error code fed into this routine is negative, this indicates a non-fatal warning that does not affect the ability of DL_MESO_DPD to continue running the simulation, although it may indicate problems with simulation setup or options chosen by the user.

Parameters

idnode	Processor node number calling subroutine (0 is the default for printing)
iode	Error/warning code
value	Optional value used by some error messages to provide more context or pin down the location where DL_MESO_DPD fails

10.11 ewald_module.F90

10.11.1 Summary

Module to calculate real-space (short-range) and reciprocal-space (long-range) electrostatic forces for Ewald sums. (OpenMP multithreaded version available with ewald_module_omp.F90.)

10.11.2 Functions/Subroutines

- `subroutine` *ewald_initialize()*

  Sets up variables required for Ewald sum calculations.

- `subroutine` *loadpart_ewald()*

  Fills an array with particle indices in a link cell for real-space Ewald force calculations.

- `subroutine` *diff_ewald()*

  Finds all interacting pairs of particles inside current link cell and its neighbouring cells for Ewald sum real-space force calculations.

- `subroutine` *ewald_real_point()*

  Calculates pairwise real-space Ewald forces between pairs of charged particles, using point charges (no charge smearing scheme).

- `subroutine` *ewald_real_linear()*

  Calculates pairwise real-space Ewald forces between pairs of charged particles, using linear charge smearing.

- `subroutine` *ewald_real_slater_exact()*

  Calculates pairwise real-space Ewald forces between pairs of charged particles, using exact Slater charge smearing.

- `subroutine` *ewald_real_slater_approx()*

  Calculates pairwise real-space Ewald forces between pairs of charged particles, using approximate Slater charge smearing.

- `subroutine` *ewald_real_gauss()*

  Calculates pairwise real-space Ewald forces between pairs of charged particles, using Gaussian charge smearing.

- `subroutine` *ewald_real_sinusoidal()*

  Calculates pairwise real-space Ewald forces between pairs of charged particles, using sinusoidal charge smearing.

- `subroutine` *ewald_real_potentials()*

  Calculates pairwise real-space Ewald potentials between pairs of charged particles, using specified charge smearing scheme, based on initial configuration.

- `subroutine` *ewald_reciprocal_map()*

  Determines vectors within range for Ewald reciprocal-space calculations.

- `subroutine` *ewald_reciprocal()*

  Calculates reciprocal-space forces on charged particles for Ewald summation.

- `subroutine` *ewald_reciprocal_potentials()*

  Calculates reciprocal-space potentials on charged particles for Ewald summation.

- `subroutine` *ewald_frozen()*

  Calculates corrective forces, potential energies, virials and stress tensors to remove electrostatic interactions between frozen charged particles in Ewald sums.

10.11.3 Variables

- `real(kind=dp), save` *kmax1r*

  Double-precision real value for maximum k-vector (x-component)

- `real(kind=dp), save` *kmax2r*

  Double-precision real value for maximum k-vector (y-component)

- `real(kind=dp), save` *kmax3r*

  Double-precision real value for maximum k-vector (z-component)

10.11.4 Function/Subroutine Documentation

diff_ewald()

```
subroutine ewald_module::diff_ewald (integer, dimension (:,:), intent(in) pone,
                                     integer, dimension (:), intent(in) tone,
                                     integer, dimension (:,:), intent(inout) pair,
                                     real(kind=dp), dimension (:,:), intent(inout)␣
↪pdata,
                                     integer, intent(inout) tpairs
                                    )
```

Takes the constructed lists of particle indices for a given Ewald sum link cell and adds all pairs within the maximum real-space cutoff distance to an array listing the pairs, storing the vector and distance between each particle pair. The number of pairs is also recorded in preparation for looping through them to calculate the real-space Ewald forces. More stringent checks in qualifying distances between particle pairs are applied if the simulation box is small (i.e. if at least one simulation box dimension only has a single Ewald link cell across all processors). Pairs of particles in the excluded interaction list are omitted from the list.

Parameters

pone	List of particle indices for all available link cells
tone	Total numbers of particles in all available link cells
pair	List of interacting particle pairs
pdata	Vectors and distances between particle pairs
tpairs	Total number of particle pairs for link cell

ewald_frozen()

```
subroutine ewald_module::ewald_frozen
```

Finds all pairs of frozen charged particles using a replicated data strategy and calculates corrections to reciprocal-space forces, potential energies, virials and stress tensors to remove their contributions. No corrections need to be made to real-space contributions as all pairs of frozen charged particles are automatically omitted. This subroutine only needs to be called once if the simulation does not use a barostat, but otherwise has to be called whenever the simulation volume changes (i.e. every timestep).

ewald_initialize()

```
subroutine ewald_module::ewald_initialize
```

Assigns double-precision real values for maximum reciprocal vectors (supplied by user as integers), in preparation for calculating electrostatic interactions using Ewald sums. This subroutine only has to be called once before calculations commence.

ewald_real_gauss()

```
subroutine ewald_module::ewald_real_gauss (integer nlimit)
```

Calculates all real-space forces for Ewald sums between pairs of charged particles within a cutoff radius (the real-space cutoff). The main loop in this subroutine goes through all of the Ewald link cells in the subdomain and searches for particle pairs in each cell and its neighbours: this loop is divided up among available threads in the OpenMP version, which either uses additional memory per thread or uses a critical region to assign forces to particles in a threadsafe manner. This subroutine uses a Gaussian charge smearing scheme [24][146] applied in real-space, as this should result in the standard Coulombic potential at the longer distances used for reciprocal-space calculations. (This routine is called if the charge smearing length does not match that for the Ewald sum itself: in the case of equal values, all real-space terms reduce to zero.)

Parameters

nlimit	Total number of particles in subdomain and boundary halo

ewald_real_linear()

```
subroutine ewald_module::ewald_real_linear (integer nlimit)
```

Calculates all real-space forces for Ewald sums between pairs of charged particles within a cutoff radius (the real-space cutoff). The main loop in this subroutine goes through all of the Ewald link cells in the subdomain and searches for particle pairs in each cell and its neighbours: this loop is divided up among available threads in the OpenMP version, which either uses additional memory per thread or uses a critical region to assign forces to particles in a threadsafe manner. This subroutine uses a linear charge smearing scheme [45] applied in real-space, as this should result in the standard Coulombic potential at the longer distances used for reciprocal-space calculations.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

ewald_real_point()

```
subroutine ewald_module::ewald_real_point (integer nlimit)
```

Calculates all real-space forces for Ewald sums between pairs of charged particles within a cutoff radius (the real-space cutoff). The main loop in this subroutine goes through all of the Ewald link cells in the subdomain and searches for particle pairs in each cell and its neighbours: this loop is divided up among available threads in the OpenMP version, which either uses additional memory per thread or uses a critical region to assign forces to particles in a threadsafe manner. This subroutine uses point charges, i.e. no charge smearing scheme is applied in real-space.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

ewald_real_potentials()

```
subroutine ewald_module::ewald_real_potentials (integer nlimit)
```

Calculates all real-space potentials for Ewald sums between pairs of charged particles within a cutoff radius (the real-space cutoff) without calculating the associated forces. The main loop in this subroutine goes through all of the Ewald link cells in the subdomain and searches for particle pairs in each cell and its neighbours: this loop is divided up among available threads in the OpenMP version, which uses a reduction operator to sum up contributions from all threads. Virial and stress tensors are also calculated. Any charge smearing scheme specified by the user is applied in real-space, as this should result in the standard Coulombic potential at the longer distances used for reciprocal-space calculations. This subroutine is use prior to starting a simulation if forces have already been provided in a CONFIG file.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

ewald_real_sinusoidal()

```
subroutine ewald_module::ewald_real_sinusoidal (integer nlimit)
```

Calculates all real-space forces for Ewald sums between pairs of charged particles within a cutoff radius (the real-space cutoff). The main loop in this subroutine goes through all of the Ewald link cells in the subdomain and searches for particle pairs in each cell and its neighbours: this loop is divided up among available threads in the OpenMP version, which either uses additional memory per thread or uses a critical region to assign forces to particles in a threadsafe manner. This subroutine uses a sinusoidal charge smearing scheme [37] applied in real-space, as this should result in the standard Coulombic potential at the longer distances used for reciprocal-space calculations.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

ewald_real_slater_approx()

```
subroutine ewald_module::ewald_real_slater_approx (integer nlimit)
```

Calculates all real-space forces for Ewald sums between pairs of charged particles within a cutoff radius (the real-space cutoff). The main loop in this subroutine goes through all of the Ewald link cells in the subdomain and searches for particle pairs in each cell and its neighbours: this loop is divided up among available threads in the OpenMP version, which either uses additional memory per thread or uses a critical region to assign forces to particles in a threadsafe manner. This subroutine uses an approximated Slater charge smearing scheme [44] applied in real-space, as this should result in the standard Coulombic potential at the longer distances used for reciprocal-space calculations.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

ewald_real_slater_exact()

```
subroutine ewald_module::ewald_real_slater_exact (integer nlimit)
```

Calculates all real-space forces for Ewald sums between pairs of charged particles within a cutoff radius (the real-space cutoff). The main loop in this subroutine goes through all of the Ewald link cells in the subdomain and searches for particle pairs in each cell and its neighbours: this loop is divided up among available threads in the OpenMP version, which either uses additional memory per thread or uses a critical region to assign forces to particles in a threadsafe manner. This subroutine uses an exact Slater charge smearing scheme [145] applied in real-space, as this should result in the standard Coulombic potential at the longer distances used for reciprocal-space calculations.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

ewald_reciprocal()

```
subroutine ewald_module::ewald_reciprocal
```

Calculates long-range Coulombic forces and potential energies using standard Ewald summation (i.e. analytical calculation of Fourier transforms of charges), including self-energy corrections for charged systems and any dipole moment-based corrections for slab-like geometries [153] (i.e non-periodic boundaries in one dimension). The OpenMP version divides the reciprocal vectors among available threads for potential calculations and divides the particles among threads for force calculations.

ewald_reciprocal_map()

```
subroutine ewald_module::ewald_reciprocal_map
```

Puts together list of reciprocal vectors within range for the long-range (reciprocal-space) part of the Ewald sum, taking user-specified vacuum gaps and Lees-Edwards shearing boundaries [148] into account. This subroutine only needs to be called once for constant volume (NVT) systems without shearing, but needs to be called every timestep for systems with barostats or Lees-Edwards shearing boundary conditions.

ewald_reciprocal_potentials()

```
subroutine ewald_module::ewald_reciprocal_potentials
```

Calculates long-range Coulombic potential energies using standard Ewald summation (i.e. analytical calculation of Fourier transforms of charges), including self-energy corrections for charged systems and any dipole moment-based corrections for slab-like geometries [153] (i.e non-periodic boundaries in one dimension). The OpenMP version divides the reciprocal vectors among available threads for potential calculations and divides the particles among threads for force calculations. This subroutine does not assign forces to particles and is intended to calculate potential, virial and stress tensor contributions at the start of a DPD simulation when forces are already known.

loadpart_ewald()

```
subroutine ewald_module::loadpart_ewald (integer, intent(in) cell,
                                          integer, intent(in) kc,
                                          integer, dimension (:,:), intent(inout)␣
↪loaded,
                                          integer, dimension (:), intent(inout)␣
↪total
                                         )
```

Starting from a given link cell, read the linked-cell list arrays for Ewald summation to create a list of particles (by local indices) in either the link cell itself or a neighbouring link cell, recording the total number of particles in that cell.

Parameters

cell	Link cell number to search for particles
kc	Link cell neighbour (1 for the link cell itself)
loaded	List of particles in link cell and its neighbours
total	Total number of particles in link cell and its neighbours

10.11.5 Variable Documentation

kmax1r

```
real(kind=dp), save ewald_module::kmax1r
```

Double precision real value for x-component of maximum reciprocal vector (number of periodic images) for Ewald sum calculations: value obtained by converting integer value either directly supplied in CONTROL file or calculated from supplied relative error in electrostatic potential.

kmax2r

```
real(kind=dp), save ewald_module::kmax2r
```

Double precision real value for y-component of maximum reciprocal vector (number of periodic images) for Ewald sum calculations: value obtained by converting integer value either directly supplied in CONTROL file or calculated from supplied relative error in electrostatic potential.

kmax3r

```
real(kind=dp), save ewald_module::kmax3r
```

Double precision real value for z-component of maximum reciprocal vector (number of periodic images) for Ewald sum calculations: value obtained by converting integer value either directly supplied in CONTROL file or calculated from supplied relative error in electrostatic potential.

10.12 field_module.F90

10.12.1 Summary

Module to calculate pairwise forces between particles other than bonded, surface or electrostatic interactions. (OpenMP multithreaded version available with field_module_omp.F90.)

10.12.2 Functions/Subroutines

- `subroutine` *conservativeforce()*

  Calculates conservative interaction forces and potentials between a specified pair of particles.

- `subroutine` *loadpart()*

  Fills an array with particle indices in a link cell.

- `subroutine` *diff()*

  Finds all interacting pairs of particles inside current link cell and its neighbouring cells for standard force calculations.

- `subroutine` *forces_mdvv()*

  Calculates pairwise forces for system applying DPD with MD Velocity Verlet.

- `subroutine` *forces_dpdvv()*

  Calculates pairwise forces for system applying DPD with DPD Velocity Verlet.

- `subroutine` *dragforces_dpdvv()*

  Recalculates pairwise dissipative forces for system applying DPD with DPD Velocity Verlet.

- `subroutine` *forces_shardlow()*

  Calculates pairwise forces for system applying DPD with Shardlow splitting.

- `subroutine` *shardlow_integrate()*

  Applies Shardlow operator to particle velocities to integrate DPD dissipative and random forces.

- `subroutine` *forces_lowe()*

  Calculates pairwise forces for system applying Lowe-Andersen thermostat.

- `subroutine` *forces_peters()*

  Calculates pairwise forces for system applying Peters thermostat.

- `subroutine` *forces_stoyanov()*

  Calculates pairwise forces for system applying Stoyanov-Groot thermostat.

- `subroutine` *potentials_initial()*

  Calculates pairwise potentials for system based on initial configuration.

- `subroutine` *plcfor_mdvv()*

  Sets up parallel link cells and calculates all forces on particles for system applying DPD with MD Velocity Verlet.

- `subroutine` *plcfor_dpdvv()*

  Sets up parallel link cells and calculates all forces on particles for system applying DPD with DPD Velocity Verlet.

- subroutine *plcfor_shardlow()*

  Sets up parallel link cells and calculates all forces on particles for system applying DPD with Shardlow splitting (either first or second order).

- subroutine *plcfor_lowe()*

  Sets up parallel link cells and calculates all forces on particles for system applying the Lowe-Andersen thermostat.

- subroutine *plcfor_peters()*

  Sets up parallel link cells and calculates all forces on particles for system applying the Peters thermostat.

- subroutine *plcfor_stoyanov()*

  Sets up parallel link cells and calculates all forces on particles for system applying the Stoyanov-Groot thermostat.

- subroutine *plcfor_initial()*

  Sets up parallel link cells and calculates all forces or potentials on particles when starting a new simulation.

- subroutine *freeze_beads()*

  Quenches forces and velocities for frozen particles.

10.12.3 Function/Subroutine Documentation

conservativeforce()

```
subroutine field_module::conservativeforce (integer i,
                                            integer j,
                                            integer k,
                                            real(kind=dp) rrr,
                                            real(kind=dp) rsq,
                                            real(kind=dp) gforce,
                                            real(kind=dp) pot
                                           )
```

This routine calculates non-bonded, non-electrostatic, non-surface interaction forces and potentials. The four current options for these forces and potentials include Lennard-Jones [66]:

$$U_{ij} = 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right]$$

Weeks-Chandler-Andersen [147]:

$$U_{ij} = 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] + \epsilon$$

'standard' Groot-Warren DPD [46]:

$$\vec{F}_{ij}^{C} = A_{ij}\left(1 - \frac{r_{ij}}{r_c}\right)\frac{\vec{r}_{ij}}{r_{ij}}$$

and a two-term many-body DPD interaction for vapour/liquid mixtures [142]:

$$\vec{F}_{ij}^{C} = \left[A_{ij}\left(1 - \frac{r_{ij}}{r_c}\right) + B_{ij}(\rho_i + \rho_j)\left(1 - \frac{r_{ij}}{r_d}\right)\right]\frac{\vec{r}_{ij}}{r_{ij}}$$

(only calculating the non-density dependent potential at this stage). This subroutine may be changed by users who wish to use different or additional interaction functional forms. Note that the force is divided by the distance in this subroutine so that multiplying this value by the vector between the particles gives the required scalar force multiplied by the unit vector.

Parameters

i	Local index for particle i
j	Local index for particle j
k	Potential number dependent on species of both particles in pair (used to identify interaction type and parameters)
rrr	Distance between particles i and j, r_{ij}
rsq	Squared distance between particles i and j
gforce	Resulting conservative force between pair of particles divided by distance, $\frac{\vec{F}_{ij}^C}{r_{ij}}$
pot	Resulting potential for pair of particles, U_{ij}

diff()

```
subroutine field_module::diff (integer, dimension (:,:), intent(in) pone,
                               integer, dimension (:), intent(in) tone,
                               integer, dimension(:,:), intent(inout) pair,
                               real(kind=dp), dimension(:,:), intent(inout) pdata,
                               integer, intent(inout) tpairs,
                               integer, intent(in) icell,
                               integer, intent(in) kcmax
                              )
```

Takes the constructed lists of particle indices for a given link cell and adds all pairs within the maximum interaction cutoff distance to an array listing the pairs, storing the vector and distance between each particle pair and various switches to indicate whether or not the pair is involved in conservative interactions and/or pairwise thermostatting. The number of pairs is also recorded in preparation for looping through them to calculate the forces. More stringent checks in qualifying distances between particle pairs are applied if the simulation box is small (i.e. if at least one simulation box dimension only has a single link cell over all available processors).

Parameters

pone	List of particle indices for all available link cells
tone	Total numbers of particles in all available link cells
pair	List of interacting particle pairs and flags to indicate if pairs are included in conservative interactions and thermostatting
pdata	Vectors and distances between particle pairs
tpairs	Total number of particle pairs for link cell
icell	Current link cell number
kc-max	Maximum number of neighbouring link cells in which to search for interacting pairs

dragforces_dpdvv()

```
subroutine field_module::dragforces_dpdvv (integer nlimit)
```

Repeats calculations of dissipative forces between pairs of particles within a cut-off radius, using updated particle velocities after the second stage of Velocity Verlet force integration as part of applying the DPD thermostat using DPD Velocity Verlet integration. The main loop in this subroutine goes through all of the link cells in the subdomain and searches for particle pairs in each cell and its neighbours: this loop is divided up among available threads in the OpenMP version, which either uses additional memory per thread or uses a critical region to assign forces to particles in a threadsafe manner. Dissipative contributions to the virial and stress tensor are calculated at this point. Unlike other interaction forces, this subroutine does not require a subsequent force import communication step as the forces are deterministic and thus link cells close to the edge of the subdomain can look in all neighbouring cells in the boundary halo to find all particle pairs.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

forces_dpdvv()

```
subroutine field_module::forces_dpdvv (integer nlimit)
```

Calculates all interaction forces (other than bonded, surface and electrostatic interactions) between pairs of particles within a cut-off radius, as well as random and dissipative forces to apply the DPD thermostat using DPD Velocity Verlet integration: this requires recalculation of dissipative forces after the second stage of Velocity Verlet integration of forces, so these forces are assigned to different arrays from the conservative and random forces. The main loop in this subroutine goes through all of the link cells in the subdomain and searches for particle pairs in each cell and its neighbours: this loop is divided up among available threads in the OpenMP version, which either uses additional memory per thread or uses a critical region to assign forces to particles in a threadsafe manner. DPD random forces make use of a uniform random number generator (default: Mersenne Twister) to calculate approximate Gaussian random numbers [46]:

$$\zeta_{ij} \approx \sqrt{12}\left(u_{ij} - 0.5\right)$$

that gives statistically similar results to real Gaussian random numbers [26] for lower computational cost. Potential energy, virial and stress tensors are also calculated, the latter separated out into conservative and random contributions. (Dissipative contributions to virial and stress tensors are assigned after force recalculation.)

Parameters

nlimit	Total number of particles in subdomain and boundary halo

forces_lowe()

```
subroutine field_module::forces_lowe (integer nlimit)
```

Calculates all interaction forces (other than bonded, surface and electrostatic interactions) between pairs of particles within a cut-off radius. The main loop in this subroutine goes through all of the link cells in the subdomain and searches for particle pairs in each cell and its neighbours: this loop is divided up among available threads in the OpenMP version, which either uses additional memory per thread or uses a critical region to assign forces to particles and create thermostatting lists in a threadsafe manner. Potential energy, virial and stress tensors are also calculated, the latter providing the conservative contributions for this property. To apply the Lowe-Andersen pairwise thermostat, a random number is generated for each pair that can be thermostatted: if this number is less than the product of collision frequency and timestep, the particle pair is added to a list for thermostatting during the second stage of Velocity Verlet force integration.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

forces_mdvv()

```
subroutine field_module::forces_mdvv (integer nlimit)
```

Calculates all interaction forces (other than bonded, surface and electrostatic interactions) between pairs of particles within a cut-off radius, as well as random and dissipative forces to apply the DPD thermostat using MD (simple) Velocity Verlet integration of all forces. The main loop in this subroutine goes through all of the link cells in the subdomain and searches for particle pairs in each cell and its neighbours: this loop is divided up among available threads in the OpenMP version, which either uses additional memory per thread or uses a critical region to assign forces to particles in a threadsafe manner. DPD random forces make use of a uniform random number generator (default: Mersenne Twister) to calculate approximate Gaussian random numbers [46]:

$$\zeta_{ij} \approx \sqrt{12}\,(u_{ij} - 0.5)$$

that gives statistically similar results to real Gaussian random numbers [26] for lower computational cost. Potential energy, virial and stress tensors are also calculated, the latter separated out into conservative, dissipative and random contributions.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

forces_peters()

```
subroutine field_module::forces_peters (integer nlimit)
```

Calculates all interaction forces (other than bonded, surface and electrostatic interactions) between pairs of particles within a cut-off radius. The main loop in this subroutine goes through all of the link cells in the subdomain and searches for particle pairs in each cell and its neighbours: this loop is divided up among available threads in the OpenMP version, which either uses additional memory per thread or uses a critical region to assign forces to particles and create thermostatting lists in a threadsafe manner. Potential energy, virial and stress tensors are also calculated, the latter providing the conservative contributions for this property. To apply the Peters pairwise thermostat, each pair of particles for thermostatting is added to a list for applying the thermostat during the second stage of Velocity Verlet force integration.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

forces_shardlow()

```
subroutine field_module::forces_shardlow (integer nlimit)
```

Calculates all interaction forces (other than bonded, surface and electrostatic interactions) between pairs of particles within a cut-off radius. The main loop in this subroutine goes through all of the link cells in the subdomain and searches for particle pairs in each cell and its neighbours: this loop is divided up among available threads in the OpenMP version, which either uses additional memory per thread or uses a critical region to assign forces to particles in a threadsafe manner. Potential energy, virial and stress tensors are also calculated, the latter providing the conservative contributions for this property. This subroutine is also used to calculate forces acting on particles during simulation startup if these are not provided in a CONFIG file.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

forces_stoyanov()

```
subroutine field_module::forces_stoyanov (integer nlimit)
```

Calculates all interaction forces (other than bonded, surface and electrostatic interactions) between pairs of particles within a cut-off radius. The main loop in this subroutine goes through all of the link cells in the subdomain and searches for particle pairs in each cell and its neighbours: this loop is divided up among available threads in the OpenMP version, which either uses additional memory per thread or uses a critical region to assign forces to particles and create thermostatting lists in a threadsafe manner. To apply the Stoyanov-Groot pairwise thermostat, a random number is generated for each pair that can be thermostatted. If this number is less than the product of collision frequency and timestep, the particle pair is added to a list for thermostatting during the second stage of Velocity Verlet force integration (similar to the Lowe-Andersen thermostat). An additional pairwise force is calculated for other thermostatting pairs that is scaled with an instantaneous temperature based on contributions from all pairs. Potential energy, virial and stress tensors are also calculated: the latter are separated out into conservative and dissipative contributions, with the dissipative contributions coming from the temperature-dependent pairwise thermostatting forces.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

freeze_beads()

```
subroutine field_module::freeze_beads
```

Resets the velocities and forces for frozen particles in the system to zero: these particles are readily found as the first particles in local arrays.

loadpart()

```
subroutine field_module::loadpart (integer, intent(in) cell,
                                   integer, intent(in) kc,
                                   integer, dimension (:,:), intent(inout) loaded,
                                   integer, dimension (:), intent(inout) total
                                  )
```

Starting from a given link cell, read the linked-cell list arrays to create a list of particles (by local indices) in either the link cell itself or a neighbouring link cell, recording the total number of particles in that cell.

Parameters

cell	Link cell number to search for particles
kc	Link cell neighbour (1 for the link cell itself)
loaded	List of particles in link cell and its neighbours
total	Total number of particles in link cell and its neighbours

plcfor_dpdvv()

```
subroutine field_module::plcfor_dpdvv
```

Creates boundary halo, sets up parallel link cells and carries out calculations of all forces acting on particles due to interactions and DPD thermostat to be integrated using DPD Velocity Verlet, i.e. recalculation of dissipative forces after force integration.

plcfor_initial()

```
subroutine field_module::plcfor_initial (logical l_config)
```

Creates boundary halo, sets up parallel link cells and carries out calculations of all forces or potentials acting on particles due to interactions (and thermostatting if using DPD with MD or DPD Velocity Verlet integration). Only the potentials are needed if a CONFIG file with particle forces is supplied.

Parameters

l_config	Flag to determine if CONFIG file is being used to provide initial forces

plcfor_lowe()

```
subroutine field_module::plcfor_lowe
```

Creates boundary halo, sets up parallel link cells and carries out calculations of all forces acting on particles due to interactions with collection of particle pairs to be thermostatted using the Lowe-Andersen thermostat after force integration.

plcfor_mdvv()

```
subroutine field_module::plcfor_mdvv
```

Creates boundary halo, sets up parallel link cells and carries out calculations of all forces acting on particles due to interactions and DPD thermostat to be integrated using a standard Velocity Verlet scheme as used in molecular dynamics (MD) simulations.

plcfor_peters()

```
subroutine field_module::plcfor_peters
```

Creates boundary halo, sets up parallel link cells and carries out calculations of all forces acting on particles due to interactions with collection of particle pairs to be thermostatted using the Peters thermostat after force integration.

plcfor_shardlow()

```
subroutine field_module::plcfor_shardlow
```

Creates boundary halo, sets up parallel link cells and carries out calculations of all forces acting on particles due to interactions, with the dissipative and random forces for the DPD thermostat applied separately using Shardlow splitting.

plcfor_stoyanov()

```
subroutine field_module::plcfor_stoyanov
```

Creates boundary halo, sets up parallel link cells and carries out calculations of all forces acting on particles due to interactions and Nosé-Hoover thermostatting, as well as collecting particle pairs to be thermostatted using the Lowe-Andersen thermostat after force integration.

potentials_initial()

```
subroutine field_module::potentials_initial (integer nlimit)
```

Calculates all interaction potentials (other than bonded, surface and electrostatic interactions) between pairs of particles within a cut-off radius without calculating the associated forces. The main loop in this subroutine goes through all of the link cells in the subdomain and searches for particle pairs in each cell and its neighbours: this loop is divided up among available threads in the OpenMP version, which uses a reduction operator to sum up contributions from all threads. Virial and stress tensors are also calculated: only the conservative contributions are given here. This subroutine is used prior to starting a simulation if forces have already been provided in a CONFIG file.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

shardlow_integrate()

```
subroutine field_module::shardlow_integrate (integer nlimit,
                                              real(kind=dp) tstepfrac,
                                              integer stage
                                             )
```

Calculates and applies the Shardlow operator [120] for particle pairs within a cut-off radius. The main loops in this subroutine go through all of the link cells in the subdomain and search for particle pairs in each cell and its neighbours: these loops are divided up among available threads in the OpenMP version, which either uses additional memory per thread or uses a critical region to assign velocity changes to particles in a threadsafe manner. DPD random forces make use of a uniform random number generator (Saru) to calculate approximate Gaussian random numbers [46]:

$$\zeta_{ij} \approx \sqrt{12}\left(u_{ij} - 0.5\right)$$

that gives statistically similar results to real Gaussian random numbers [26] for lower computational cost. The same random number for each particle pair is required for both stages in applying the SHardlow operator. Dissipative and random contributions to virial and stress tensors are calculated during this subroutine. This subroutine is applied once for first-order Shardlow splitting and twice for second-order Shardlow splitting, the latter integrating forces over half a timestep during each pass.

Parameters

nlimit	Total number of particles in subdomain and boundary halo
tstepfrac	Fraction of timestep to integrate dissipative and random forces in each stage of Shardlow operator
stage	Velocity Verlet stage during which the operator is applied - this affects random number generator seeding

10.13 manybody_module.F90

10.13.1 Summary

Module to calculate local densities and potentials for many-body DPD interactions. (OpenMP multi-threaded version available with manybody_module_omp.F90.)

10.13.2 Functions/Subroutines

- `subroutine` *loadpart_manybody()*

  Fills an array with particle indices in a link cell for many-body DPD local density calculations.

- `subroutine` *diff_manybody()*

  Finds all pairs of particles inside current link cell and its neighbouring cells within many-body DPD cutoff for local density calculations.

- `subroutine` *local_density()*

  Calculates local densities for many-body DPD interactions.

- `real(kind=dp) function` *weight_rho()*

  Calculates normalised weight function for many-body DPD localised densities.

- `subroutine` *manybody_potential()*

  Calculates self-energies resulting from density-dependent (many-body DPD) interactions for every particle involved.

10.13.3 Function/Subroutine Documentation

diff_manybody()

```
subroutine manybody_module::diff_manybody (integer, dimension (:,:), intent(in)␣
↪pone,
                                           integer, dimension (:), intent(in) tone,
                                           integer, dimension(:,:), intent(inout)␣
↪pair,
                                           real(kind=dp), dimension(:,:),␣
↪intent(inout) pdata,
                                           integer, intent(inout) tpairs,
                                           integer, intent(in) kcmax
                                          )
```

Takes the constructed lists of particle indices for a given many-body DPD link cell and adds all pairs within the maximum cutoff distance for local densities to an array listing the pairs, storing the vector and distance between each particle pair. The number of pairs is also recorded in preparation for looping through them to calculate local densities. More stringent checks in qualifying distances between particle pairs are applied if the simulation box is small (i.e. if at least one simulation box dimension only has a single many-body DPD link cell over all available processors).

Parameters

pone	List of particle indices for all available link cells
tone	Total numbers of particles in all available link cells
pair	List of interacting particle pairs
pdata	Vectors and distances between particle pairs
tpairs	Total number of particle pairs for link cell
kcmax	Maximum number of neighbouring link cells in which to search for interacting pairs

loadpart_manybody()

```
subroutine manybody_module::loadpart_manybody (integer, intent(in) cell,
                                               integer, intent(in) kc,
                                               integer, dimension (:,:),␣
↪intent(inout) loaded,
                                               integer, dimension (:),␣
↪intent(inout) total
                                              )
```

Starting from a given link cell, read the linked-cell list arrays for many-body DPD local densities to create a list of particles (by local indices) in either the link cell itself or a neighbouring link cell, recording the total number of particles in that cell.

Parameters

cell	Link cell number to search for particles
kc	Link cell neighbour (1 for the link cell itself)
loaded	List of particles in link cell and its neighbours
total	Total number of particles in link cell and its neighbours

local_density()

```
subroutine manybody_module::local_density (integer nlimit)
```

Calculates local densities summed up over pairs of particles within the many-body DPD cutoff radius:

$$\rho_i = \sum_{j \neq i} w^{\rho}\left(r_{ij}\right)$$

omitting self-contributions for each particle [140]. The main loop in this subroutine goes through all the link cells in the subdomain and searches for particle pairs in each cell and its neighbours: this loop is divided up among available threads in the OpenMP version, which either uses additional memory per thread or uses a critical region to assign densities to particles in a threadsafe manner. Contributions are calculated separately for individual particle species, which can be summed together for obtain densities for all particle species. A full search of boundary link cells is carried out in this subroutine to ensure all pairwise contributions are included for all particles in the current subdomain (negating the need to communicate contributions from neighbours afterwards).

Parameters

nlimit	Total number of particles in subdomain and boundary halo

manybody_potential()

```
subroutine manybody_module::manybody_potential
```

Sums up self-energy potential terms for many-body DPD interactions over all involved particles and species (pairs of frozen particles are excluded). The default form is based on the density term used in the two-term vapour-liquid model [142] where δ_{kl} is the Kronecker delta):

$$U^{mb} = \frac{\pi r_d^4}{30} B \sum_k \sum_l \rho_k \rho_l \left(2 - \delta_{kl}\right)$$

(The remainder of the potential is calculated with conservative interaction forces.) Users may wish to modify this routine to use their own many-body DPD interaction models.

weight_rho()

```
real(kind=dp) function manybody_module::weight_rho (real(kind=dp) rrr)
```

This function provides the pairwise weight function required for calculating localised densities needed for many-body DPD interactions. The default weight function [142] 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.

Parameters

rrr	Distance between pair of particles, r

10.14 read_module.F90

10.14.1 Summary

Module for reading DL_MESO_DPD input files with system, interaction, molecular and configuration data.

10.14.2 Data Types

- `type` *type read_module::particledata_p*

  Particle data for molecules.

- `type` *type read_module::particledata*

  Particle data.

10.14.3 Functions/Subroutines

- `subroutine` *scan_control()*

  Scans CONTROL file for essential simulation directives.

- `subroutine` *scan_field()*

  Scan FIELD file for information on array bounds for interaction data.

- `subroutine` *scan_config()*

  Scan CONFIG file for system unit cell dimensions.

- `subroutine` *scan_export()*

  Scan export file for system unit cell dimensions.

- `subroutine` *read_control()*

  Reads in system data from CONTROL file.

- `subroutine` *read_field()*

  Reads in interaction information from FIELD file.

- `subroutine` *read_config()*

  Reads initial system configuration from CONFIG file.

- `subroutine` *readpos()*

  Finds starting position of entry in CONFIG file closest to given byte number.

- `subroutine` *read_export()*

  Read configuration from export file for simulation restart.

- `subroutine` *read_revive()*

  Read statistical accumulators from REVIVE file for simulation restart.

10.14.4 Data Type Documentation

type read_module::particledata_p

Table 10.1: Class Members

real(kind=dp), dimension(:), allocatable	fx	Particle force (x-component)
real(kind=dp), dimension(:), allocatable	fy	Particle force (y-component)
real(kind=dp), dimension(:), allocatable	fz	Particle force (z-component)
integer, dimension(:), allocatable	species	Particle species
real(kind=dp), dimension(:), allocatable	vx	Particle velocity (x-component)
real(kind=dp), dimension(:), allocatable	vy	Particle velocity (y-component)
real(kind=dp), dimension(:), allocatable	vz	Particle velocity (z-component)
real(kind=dp), dimension(:), allocatable	x	Particle position (x-component)
real(kind=dp), dimension(:), allocatable	y	Particle position (y-component)
real(kind=dp), dimension(:), allocatable	z	Particle position (z-component)

type read_module::particledata

Table 10.2: Class Members

real(kind=dp), dimension(:), allocatable	fx	Particle force (x-component)
real(kind=dp), dimension(:), allocatable	fy	Particle force (y-component)
real(kind=dp), dimension(:), allocatable	fz	Particle force (z-component)
integer, dimension(:), allocatable	gb	Particle global index (after expansion)
integer, dimension(:), allocatable	global	Particle original global index.
integer, dimension(:), allocatable	species	Particle species.
real(kind=dp), dimension(:), allocatable	vx	Particle velocity (x-component)
real(kind=dp), dimension(:), allocatable	vy	Particle velocity (y-component)
real(kind=dp), dimension(:), allocatable	vz	Particle velocity (z-component)
real(kind=dp), dimension(:), allocatable	x	Particle position (x-component)
real(kind=dp), dimension(:), allocatable	y	Particle position (y-component)
real(kind=dp), dimension(:), allocatable	z	Particle position (z-component)

Function/Subroutine Documentation

read_config()

```
subroutine read_module::read_config
```

Reads initial system configuration (positions, velocities, forces) from a CONFIG file (in DL_POLY-style format) and assigns particles, bonds etc. to system, accounting for any system duplication specified using the ‘nfold’ directive in the CONTROL file. Each processor reads a near-equal size part of the CONFIG file and distributes particle data to the relevant processors based on positions. The number of particles in the CONFIG file must match the number given in the corresponding FIELD file - which are checked and reported if they do not match up - as should any molecule and bond information. The periodic boundary key (imcon) is effectively ignored as all DL_MESO_DPD systems are orthorhombic. This routine also adds any required frozen bead walls as cubic lattices. No direct checks are made to ensure molecules do not cross non-periodic boundaries or form cross-linked structures across periodic boundaries. The likely number of particles to be exported to boundary halos is evaluated and, where necessary, the sizes of transfer buffers for communications are adjusted.

read_control()

```
subroutine read_module::read_control (logical l_readvol,
                                      logical l_config,
                                      logical l_rest
                                     )
```

Reads in all system data from CONTROL file not previously obtained from initial scan.

Parameters

l_readvol	Flag to determine whether or not to read simulation volume provided in CONTROL file
l_config	Flag to determine whether or not a CONFIG file is to be read
l_rest	Flag to determine whether or not a previous simulation is to be restarted

read_export()

```
subroutine read_module::read_export (logical, intent(in) ltempscale)
```

Read in state of previous simulation from export file: particle positions, velocities and forces, Lees-Edwards shearing displacements and (if required) factor for rescaling particle velocities. Each processor reads a near-equal sized part of the export file: the particle data are redistributed to the relevant processors based on their positions. The likely number of particles to be exported to boundary halos is evaluated and, where necessary, the sizes of transfer buffers for communications are adjusted.

Parameters

ltempscale	Flag to determine if particles velocities are to be rescaled for system temperature

read_field()

```
subroutine read_module::read_field
```

Reads in all interaction information from FIELD file not previously obtained from initial scan, including interaction parameters, molecule configurations and connectivity for simulation setup, and external fields (constant body/gravity forces and electric fields). This subroutine will determine interaction parameters for species pairs not specified in the FIELD file unless any interactions are many-body DPD (requiring parameters for all species pairs). All energy-based interaction parameters (for standard pairwise interactions, surface interactions, bonded and electrostatic interactions) will be scaled with temperature if specified in the FIELD file.

read_revive()

```
subroutine read_module::read_revive
```

Read in statistical accumulators for system properties, barostat properties and random number generator states from REVIVE file. The REVIVE file is not absolutely necessary for a simulation restart: only a warning is printed if it cannot be found. The random number generator states replace those previously initialised with the exception of higher number processors if fewer processors were originally used to create the REVIVE file.

readpos()

```
subroutine read_module::readpos (integer(kind=li), intent(inout) testpos,
                                 integer(kind=li), intent(in) header,
                                 integer, intent(out) startpart,
                                 integer, intent(out) nextpart,
                                 integer, intent(in) readlines
                                )
```

Finds the starting position of a particle entry in the CONFIG file at or before the provided byte number, as well as the associated particle number.

Parameters

testpos	Byte number at start of particle entry (initially provided as estimate)
header	Number of bytes as header in CONFIG file before particle data starts
startpart	Particle index in current particle entry
nextpart	Particle index in next particle entry
readlines	Number of lines per particle entry in CONFIG file

scan_config()

```
subroutine read_module::scan_config
```

Scan the top of the CONFIG file to find the amount of data per particle (positions, velocities, forces), the simulation box shape and the dimensions of the system unit cell.

scan_control()

```
subroutine read_module::scan_control (logical l_exist,
                                      logical l_safe,
                                      logical l_scr,
                                      logical l_temp,
                                      logical l_time,
                                      logical l_conf,
                                      logical l_init,
                                      logical l_rest
                                     )
```

Checks for the existence of the CONTROL file and whether or not it can safely be read, before scanning for valid temperature and timestep values, as well as the 'l_scr' directive to divert simulation output to the screen or standard output, the 'l_conf' directive to inform DL_MESO_DPD whether or not to use CONFIG file, the 'l_init' directive to create a CONFIG file (called CFGINI) from the devised initial state, and whether or not the simulation is being restarted. This subroutine is called directly by the main DL_MESO_DPD code.

Parameters

l_exist	Flag to indicate if CONTROL file exists
l_safe	Flag to indicate if CONTROL file can be safely read
l_scr	Flag to indicate if simulation output is to be directed to screen or standard output
l_temp	Flag to indicate if simulation temperature is defined in CONTROL file
l_time	Flag to indicate if simulation timestep is defined in CONTROL file
l_conf	Flag to indicate whether or not CONFIG file is to be used
l_init	Flag to indicate if CFGINI file is to be written
l_rest	Flag to indicate if a previous simulation is to be restarted

scan_export()

```
subroutine read_module::scan_export
```

Scan the export file from a previous DL_MESO_DPD simulation to find the dimensions of the simulation box for a restart.

scan_field()

```
subroutine read_module::scan_field
```

Scan through FIELD file to find numbers of species, molecule types and interactions, the maximum numbers of parameters and particles per molecule, data for particle species (names, masses, charges and frozen flags), and parameters for bonds, angles and dihedrals.

10.15 run_module.F90

10.15.1 Summary

Module with main loops over timesteps for DPD simulations.

10.15.2 Functions/Subroutines

- `subroutine` *mdvv()*

  Calculation loop for simulation with DPD thermostat and standard (MD) Velocity Verlet integration.

- `subroutine` *dpdvv()*

  Calculation loop for simulation with DPD thermostat and DPD Velocity Verlet integration.

- `subroutine` *dpds1()*

  Calculation loop for simulation with DPD thermostat using first-order Shardlow splitting.

- `subroutine` *dpds2()*

  Calculation loop for simulation with DPD thermostat using second-order Shardlow splitting.

- `subroutine` *lowe()*

  Calculation loop for simulation with Lowe-Andersen thermostat.

- `subroutine` *peters()*

  Calculation loop for simulation with Peters thermostat.

- `subroutine` *stoyanov()*

  Calculation loop for simulation with Stoyanov-Groot thermostat.

10.15.3 Variables

- `logical` *finish* = `.false.`

  Flag to indicate whether or not DPD simulation has finished (scheduled or not)

- `integer` *i*

  Counter for loop to reassign particle properties at each timestep.

- `integer` *npage* = `25`

  Number of timesteps to report to OUTPUT file before writing column headers.

- `integer` *lines* = `0`

  Number of timesteps written to OUTPUT file.

- `integer` *klock* = `0`

  Number of timesteps passed since start of DPD simulation loop (accounting for restarts)

- `real(kind=dp)` *time*

  Simulation time in DPD units.

- `real(kind=dp)` *frctim*

  Walltime taken to calculate particle forces during timestep.

- `real(kind=dp)` *timelp*

  Current walltime for simulation (used to obtain time from function)

- `real(kind=dp)` *timsrt*

  Walltime at start of current timestep.

- `real(kind=dp)` *timfst*

  Walltime when force calculations started for current timestep.

- `real(kind=dp)` *stptim*

  Walltime after calculations for current timestep are complete (before file writing starts)

10.15.4 Function/Subroutine Documentation

dpds1()

```
subroutine run_module::dpds1 (logical l_scr)
```

Carries out a simulation using the DPD thermostat, with dissipative and random pairwise forces integrated at the start of each timestep using a first-order Shardlow splitting approach, and conservative (interaction) forces integrated using Velocity Verlet integration.

Parameters

l_scr	Flag to indicate if simulation outputs are to be diverted to standard output/screen

dpds2()

```
subroutine run_module::dpds2 (logical l_scr)
```

Carries out a simulation using the DPD thermostat, with dissipative and random pairwise forces integrated twice per timestep - once at the start, again at the middle - using a second-order Shardlow splitting approach, and conservative (interaction) forces integrated using Velocity Verlet integration.

Parameters

l_scr	Flag to indicate if simulation outputs are to be diverted to standard output/screen

dpdvv()

```
subroutine run_module::dpdvv (logical l_scr)
```

Carries out a simulation using the DPD thermostat, with dissipative and random pairwise forces integrated along with conservative (interaction) forces using Velocity Verlet integration before dissipative forces are recalculated afterwards (known as DPD Velocity Verlet).

Parameters

l_scr	Flag to indicate if simulation outputs are to be diverted to standard output/screen

lowe()

```
subroutine run_module::lowe (logical l_scr)
```

Carries out a simulation using the Lowe-Andersen thermostat, with replacements to relative velocities between particle pairs carried out at the end of each timestep, and conservative (interaction) forces integrated using Velocity Verlet integration.

Parameters

l_scr	Flag to indicate if simulation outputs are to be diverted to standard output/screen

mdvv()

```
subroutine run_module::mdvv (logical l_scr)
```

Carries out a simulation using the DPD thermostat, with dissipative and random pairwise forces integrated along with conservative (interaction) forces using the standard (molecular dynamics) form of Velocity Verlet integration.

Parameters

l_scr	Flag to indicate if simulation outputs are to be diverted to standard output/screen

peters()

```
subroutine run_module::peters (logical l_scr)
```

Carries out a simulation using the Peters thermostat, with replacements to relative velocities between particle pairs carried out at the end of each timestep, and conservative (interaction) forces integrated using Velocity Verlet integration.

Parameters

l_scr	Flag to indicate if simulation outputs are to be diverted to standard output/screen

stoyanov()

```
subroutine run_module::stoyanov (logical l_scr)
```

Carries out a simulation using the Stoyanov-Groot thermostat, with replacements to relative velocities between particle pairs carried out at the end of each timestep, and conservative (interaction) forces and instantaneous temperature-dependent pairwise forces integrated using Velocity Verlet integration.

Parameters

l_scr	Flag to indicate if simulation outputs are to be diverted to standard output/screen

10.15.5 Variable Documentation

finish

```
logical run_module::finish = .false.
```

Flag to indicate whether or not DPD simulation has finished, either due to running out of timesteps or due to running out of calculation time: checked across all processors and used to break out of the calculation loop.

frctim

```
real(kind=dp) run_module::frctim
```

Calculation walltime (in seconds) taken to calculate particle forces during current timestep.

i

```
integer run_module::i
```

Counter for loop to reassign properties (mass, species and molecule names) to each particle at each timestep after the first Velocity Verlet stage.

klock

```
integer run_module::klock = 0
```

Number of timesteps that have passed since the DPD simulation loop started for the current run (not including any timesteps passed before a simulation restart): used to calculate average wall clock times for simulation as performance measures.

lines

```
integer run_module::lines = 0
```

Number of lines of properties previously written to OUTPUT file (or to standard output/screen), used to determine when to print column headers.

npage

```
integer run_module::npage = 25
```

Frequency for writing column headers to OUTPUT file (or standard output/screen) in terms of number of lines of properties.

stptim

```
real(kind=dp) run_module::stptim
```

Walltime (in seconds) taken to carry out calculations during current timestep, excluding time taken to write to output files.

time

```
real(kind=dp) run_module::time
```

Current time of the simulation in DPD time units, equal to the product of the timestep size Δt and the current timestep number less number of equilibration timesteps.

timelp

```
real(kind=dp) run_module::timelp
```

Walltime (in seconds) since the start of the simulation to after force calculations for the current timestep.

timfst

```
real(kind=dp) run_module::timfst
```

Accumulated wall clock time taken to calculate forces during current simulation, used to calculate average time per timestep as a performance measure.

timsrt

```
real(kind=dp) run_module::timsrt
```

Walltime (in seconds) since the start of the simulation to the beginning of the current timestep.

10.16 spme_module.F90

10.16.1 Summary

Module to calculate reciprocal-space (long-range) electrostatic forces using Smooth Particle Mesh Ewald (SPME). (OpenMP multithreaded version available with spme_module_omp.F90.)

10.16.2 Functions/Subroutines

- `subroutine` *spme_initialize()*

  Sets up variables, B-splines and global charge arrays for Smooth Particle Mesh Ewald (SPME) calculations.

- `subroutine` *spme_reciprocal_map()*

  Determines vectors within range for Smooth Particle Mesh Ewald (SPME) calculations.

- `subroutine` *spme_ewald_reciprocal()*

  Calculates reciprocal-space forces on charged particles using Smooth Particle Mesh Ewald (SPME).

- `subroutine` *spme_ewald_reciprocal_potentials()*

  Calculates reciprocal-space potentials on charged particles using Smooth Particle Mesh Ewald (SPME).

- `subroutine` *spme_bspline_gen()*

  Calculates B-splines for Smooth Particle Mesh Ewald (SPME) calculations to apply charges to grid and calculate derivatives for forces.

- `subroutine` *spme_free_memory()*

  Deallocates B-spline and global charge arrays for SPME calculations.

10.16.3 Variables

- `integer, save` *mxspl2*

  Square of maximum B-spline order.

- `integer, save` *mxspl3*

  Cube of maximum B-spline order.

- `real(kind=dp), save` *kmax1r*

  Double-precision real value for maximum k-vector (x-component)

- `real(kind=dp), save` *kmax2r*

  Double-precision real value for maximum k-vector (y-component)

- `real(kind=dp), save` *kmax3r*

  Double-precision real value for maximum k-vector (z-component)

- `real(kind=dp), dimension(:,:,:), allocatable, save` *qqc*

  Double-precision real charge grid.

- `complex(kind=dp), dimension(:), allocatable, save` *bscx*

  B-spline construction coefficients (x-component)

- `complex(kind=dp), dimension(:), allocatable, save` *bscy*

  B-spline construction coefficients (y-component)

- `complex(kind=dp), dimension(:), allocatable, save` *bscz*

  B-spline construction coefficients (z-component)

- `complex(kind=dp), dimension(:,:,:), allocatable, save` *qqq*

- `complex(c_double_complex), dimension (:,:,:), allocatable, save qqq`

  Double-precision complex charge grid.

- `type(c_ptr)` *planback*

  FFTW plan for inverse 3D FFT

- `type(c_ptr)` *planfor*

  FFTW plan for forward 3D FFT

10.16.4 Function/Subroutine Documentation

spme_bspline_gen()

```
subroutine spme_module::spme_bspline_gen (integer, intent(in) natms,
                                            integer, intent(in) nospl,
                                            real(kind=dp), dimension (:), intent(in)␣
↪xxx,
                                            real(kind=dp), dimension (:), intent(in)␣
↪yyy,
                                            real(kind=dp), dimension (:), intent(in)␣
↪zzz,
                                            real(kind=dp), dimension (:,:),␣
↪intent(out) bspx,
                                            real(kind=dp), dimension (:,:),␣
↪intent(out) bspy,
                                            real(kind=dp), dimension (:,:),␣
↪intent(out) bspz,
                                            real(kind=dp), dimension (:,:),␣
↪intent(out) bsdx,
                                            real(kind=dp), dimension (:,:),␣
↪intent(out) bsdy,
                                            real(kind=dp), dimension (:,:),␣
↪intent(out) bsdz
                                          )
```

Constructs B-splines based on the positions of particles relative to the maximum reciprocal vector grid in preparation for applying a Fast Fourier Transform (FFT) as part of Smooth Particle Mesh Ewald (SPME). The derivatives of the spline are also calculated to enable forces on particles to be calculated. The spline order should be an even number and no less than 4.

Parameters

natms	Number of particles in subdomain to construct B-splines
nospl	Spline order
xxx	Particle positions (x-coordinate)
yyy	Particle positions (y-coordinate)
zzz	Particle positions (z-coordinate)
bspx	Constructed B-splines for particles (x-coordinate)
bspy	Constructed B-splines for particles (y-coordinate)
bspz	Constructed B-splines for particles (z-coordinate)
bsdx	Constructed B-spline derivatives for particles (x-coordinate)
bsdy	Constructed B-spline derivatives for particles (y-coordinate)
bsdz	Constructed B-spline derivatives for particles (z-coordinate)

spme_ewald_reciprocal()

```
subroutine spme_module::spme_ewald_reciprocal (integer nlimit)
```

Calculates long-range Coulombic forces and potential energies using the Smooth Particle Mesh Ewald (SPME) method [31], including self-energy corrections for charged systems and dipole moment-based corrections for slab-like geometries [153] (i.e. non-periodic boundaries in one dimension). The charges are assigned to a grid of size equal to the maximum reciprocal vector range and the Fourier transform of this grid is found using a three-dimensional Fast Fourier Transform (FFT) solver. DL_MESO supplies an FFT solver, but the user can choose ESSL or FFTW as alternatives when compiling the code by using compile-time flags. The OpenMP version divides the particles among available threads for force calculations.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

spme_ewald_reciprocal_potentials()

```
subroutine spme_module::spme_ewald_reciprocal_potentials (integer nlimit)
```

Calculates long-range Coulombic potential energies using the Smooth Particle Mesh Ewald (SPME) method [31], including self-energy corrections for charged systems and dipole moment-based corrections for slab-like geometries [153] (i.e. non-periodic boundaries in one dimension). The charges are assigned to a grid of size equal to the maximum reciprocal vector range and the Fourier transform of this grid is found using a three-dimensional Fast Fourier Transform (FFT) solver. DL_MESO supplies an FFT solver, but the user can choose ESSL or FFTW as alternatives when compiling the code by using compile-time flags. This subroutine does not assign forces to particles and is intended to calculate potential, virial and stress tensor contributions at the start of a DPD simulation when forces are already known.

Parameters

nlimit	Total number of particles in subdomain and boundary halo

spme_free_memory()

```
subroutine spme_module::spme_free_memory
```

Deallocates all timestep-independent arrays used for Smooth Particle Mesh Ewald (SPME) calculations - those for constructing B-splines and the global charge grids - at the end of the DPD simulation.

spme_initialize()

```
subroutine spme_module::spme_initialize
```

Assigns double-precision real values for maximum reciprocal vectors (supplied by user as integers), allocates arrays for B-spline coefficients, global charge arrays and helper arrays used to assign charges to a grid. This subroutine only has to be called once before SPME calculations commence.

spme_reciprocal_map()

```
subroutine spme_module::spme_reciprocal_map
```

Puts together list of reciprocal vectors within range for the long-range (reciprocal-space) part of the Ewald sum to be carried out using Smooth Particle Mesh Ewald (SPME), taking user-specified vacuum gaps and Lees-Edwards shearing boundaries [148] into account. This subroutine only needs to be called once for constant volume (NVT) systems without shearing, but needs to be called every timestep for systems with barostats or Lees-Edwards shearing boundary conditions.

10.16.5 Variable Documentation

bscx

```
complex(kind=dp), dimension (:), allocatable, save spme_module::bscx
```

x-component of B-spline construction coefficients used to assign charges to grid for SPME calculations.

bscy

```
complex(kind=dp), dimension (:), allocatable, save spme_module::bscy
```

y-component of B-spline construction coefficients used to assign charges to grid for SPME calculations.

bscz

```
complex(kind=dp), dimension (:), allocatable, save spme_module::bscz
```

z-component of B-spline construction coefficients used to assign charges to grid for SPME calculations.

kmax1r

```
real(kind=dp), save spme_module::kmax1r
```

Double precision real value for x-component of maximum reciprocal vector (number of periodic images) for SPME calculations: value obtained by converting integer value either directly supplied in CONTROL file or calculated from supplied relative error in electrostatic potential.

kmax2r

```
real(kind=dp), save spme_module::kmax2r
```

Double precision real value for y-component of maximum reciprocal vector (number of periodic images) for SPME calculations: value obtained by converting integer value either directly supplied in CONTROL file or calculated from supplied relative error in electrostatic potential.

kmax3r

```
real(kind=dp), save spme_module::kmax3r
```

Double precision real value for z-component of maximum reciprocal vector (number of periodic images) for SPME calculations: value obtained by converting integer value either directly supplied in CONTROL file or calculated from supplied relative error in electrostatic potential.

mxspl2

```
integer save spme_module::mxspl2
```

Square of the maximum B-spline order specified in CONTROL file, used to construct 3D charge array for SPME calculations.

mxspl3

```
integer save spme_module::mxspl3
```

Cube of the maximum B-spline order specified in CONTROL file, used to construct 3D charge array for SPME calculations.

planback

```
type(c_ptr) :: planback
```

Calculation plan for FFTW to carry out inverse Fast Fourier Transform of charge grid. (Not used if using ESSL or internal FFT solver.)

planfor

```
type(c_ptr) :: planfor
```

Calculation plan for FFTW to carry out forward Fast Fourier Transform of charge grid. (Not used if using ESSL or internal FFT solver.)

qqc

```
real(kind=dp), dimension (:,:,:), allocatable, save spme_module::qqc
```

Three-dimensional grid of charges represented as double precision real numbers.

qqq

```
complex(kind=dp), dimension (:,:,:), allocatable, save spme_module::qqq
complex(c_double_complex), dimension (:,:,:), allocatable, save spme_module::qqq
```

Three-dimensional grid of charges represented as double precision complex numbers, which undergoes Fast Fourier Transforms as part of SPME calculation. (If using FFTW, double precision complex kind is taken from Fortran `iso_c_bindings` instrinics used for interoperability with C.)

10.17 start_module.F90

10.17.1 Summary

Module to start and revive DPD simulations based on available inputted data.

10.17.2 Functions/Subroutines

- `subroutine` *start()*

  Sets up starting configuration for DPD calculation.

- `subroutine` *initialize()*

  Sets up starting configuration for DPD simulation without an initial or restart configuration.

- `subroutine` *initialvelocity()*

  Assigns initial particle velocities to give required system temperature.

- `subroutine` *sort_beads()*

  Sorts particles held by current processor.

- `subroutine` *assign_bonds()*

  Assigns bond data (bonds, angles and dihedrals) to book-keeping tables.

assign_bonds()

```
subroutine start_module::assign_bonds
```

Creates bond, angle and dihedral tables for the current processor to identify which particles are involved in these interactions. If assigning bond data globally (using 'global bonds' directive in the CONTROL file), every processor will hold all available data. If assigning bond data locally (by default), only data involving particles held by each processor - i.e. index particles for bonds, angles and dihedrals - will be assigned to that processor.

initialize()

```
subroutine start_module::initialize
```

Assigns positions and velocities for particles in the DPD simulation when only system volume and numbers of particles are provided in CONTROL and FIELD files. Particles not included in molecules are assigned to a cubic lattice with species assigned as evenly as possible among processors, while molecules are inserted into the simulation box at random positions and rotations (to fit entirely box if using hard walls or frozen bead walls). Frozen bead walls are also assigned as cubic lattices. No duplication of the system using the 'nfold' directive in the CONTROL file is assumed.

initialvelocity()

```
subroutine start_module::initialvelocity
```

Randomly assigns velocities to particles, starting with values obtained from a uniform random number generator rescaled to give correct system-wide kinetic energy for required system temperature and no overall momentum.

sort_beads()

```
subroutine start_module::sort_beads
```

Re-orders particles held by each processor to put frozen particles first by local bead index, which makes identifying these and skipping them during force integration easier.

start()

```
subroutine start_module::start (logical l_config)
```

Sets up initial particle configuration (positions and velocities) and bond information for the DPD simulation. This can either be derived from the system volume and numbers of particles given in CONTROL and FIELD files, read from a CONFIG file, or obtained from an export file when restarting a previous simulation. The initial particle velocities can either be obtained from these input files or selected randomly to provide the required system temperature. A sample of particle positions and velocities is printed to the OUTPUT file (or standard output), and the HISTORY file for simulation trajectories is prepared.

Parameters

l_config	Flag to indicate if CONFIG file is to be read

10.18 statistics_module.F90

10.18.1 Summary

Module to collect system properties for statistical analysis.

10.18.2 Functions/Subroutines

- `subroutine` *statis()*

  Calculates system properties and accumulates statistical information for simulation outputs.

10.18.3 Function/Subroutine Documentation

statis()

```
subroutine statistics_module::statis
```

Gathers together system-wide simulation properties - potential and kinetic energies, bond lengths and angles, and stress tensors - for the current timestep, and uses values to calculate rolling averages (based on continuously-updated statistical data stacks), simulation averages and standard deviations (fluctuations). These values are reported in the OUTPUT file (or standard output), the CORREL file or Stress_*.d files. Temperature rescaling of particle velocities is also applied at user-specified intervals during equilibration.

10.19 surface_module.F90

10.19.1 Summary

Module for applying boundary conditions at system planes (e.g. solid walls) and wall interaction potentials.

10.19.2 Functions/Subroutines

- `subroutine` *surfacenodes()*

  Identifies processors containing surfaces or other boundary conditions.

- `subroutine` *surfaceforce()*

  Calculates interaction force and potential between a particle and a hard boundary (wall).

- `subroutine` *wallforces()*

  Calculates and applies wall forces and potentials on particles.

- `subroutine` *wallpotentials()*

  Calculates and applies wall potentials on particles.

- `subroutine` *hardreflect_specular()*

  Applies a specular reflective condition at a hard wall.

- `subroutine` *hardreflect_bounceback()*

  Applies a bounce-back reflective condition at a hard wall.

- `subroutine` *frozenbead()*

  Calculates numbers of frozen particles needed for boundary walls.

- `subroutine` *shearslide()*

  Calculates displacement of shearing boundaries for Lees-Edwards boundary conditions.

10.19.3 Function/Subroutine Documentation

frozenbead()

```
subroutine surface_module::frozenbead (integer, intent(out) numwallbeads)
```

Determines the number of frozen particles required for a frozen bead wall, given the wall thickness and bead density, and adjusts the system dimensions and particle counts to acommodate them.

Parameters

n umwallbeads	Calculated number of frozen particles for boundary walls

hardreflect_bounceback()

```
subroutine surface_module::hardreflect_bounceback
```

For any particle about to pass through a hard wall, this subroutine relocates the particle based on it being reflected by the wall in all three dimensions and reverses its velocity and momentum, applying bounce-back reflection for a no-slip boundary.

hardreflect_specular()

```
subroutine surface_module::hardreflect_specular
```

For any particle about to pass through a hard wall, this subroutine relocates the particle based on it being reflected orthogonally to the wall while preserving tangental momentum and reverses the orthogonal velocity component, applying specular reflection for a free-slip boundary.

shearslide()

```
subroutine surface_module::shearslide
```

Based on the time elapsed after equilibration, this subroutine determines the displacement of a periodic shearing boundary as required for Lees-Edwards boundary conditions [76]. This displacement is used to adjust the positions of beads passing through shearing boundaries.

surfaceforce()

```
subroutine surface_module::surfaceforce (integer j,
                                         real(kind=dp) rrr,
                                         real(kind=dp) rsq,
                                         real(kind=dp) srfforce,
                                         real(kind=dp) srfpot
                                        )
```

This subroutine calculates all surface-based interactions acting on particles. Two types are currently available: a Groot-Warren 'DPD' soft repulsive type [105] given as:

$$U_{wall,i}(z) = \frac{1}{2} A_{wall,i} z_{c,i} \left(1 - \frac{z}{z_{c,i}}\right)^2$$

for $z < z_{c,i}$, and a Weeks-Chandler-Andersen (WCA) type given as:

$$U_{wall,i}(z) = 4\epsilon_{wall,i} \left[\left(\frac{z}{\sigma_{wall,i}}\right)^{12} - \left(\frac{z}{\sigma_{wall,i}}\right)^{6}\right] + \epsilon_{wall,i}$$

for $z < 2^{\frac{1}{6}} \sigma_{wall,i}$. Note that the force is divided by the distance in this subroutine so that multiplying this value by the vector between the particle and the wall gives the required scalar force multiplied by the unit vector.

Parameters

j	Species number for particle (i)
rrr	Distance between particle and wall, $z_{wall,i}$
rsq	Squared distance between particle and wall
srfforce	Resulting surface force between particle and wall divided by distance, $\frac{\vec{F}_{wall,i}}{z_{wall,i}}$
srfpot	Resulting wall potential, $U_{wall,i}$

surfacenodes()

```
subroutine surface_module::surfacenodes
```

Determines if each processor includes a surface (e.g. a hard wall or a shearing boundary), indicates which surfaces exist for the purposes of applying communications (deport, export and import steps) between processors and, for reflecting boundaries, determines their locations within the processor's subdomain. If Lees-Edwards shearing boundaries are in use, these are only applied after equilibration.

wallforces()

```
subroutine surface_module::wallforces
```

Based on their positions relative to a reflecting hard wall boundary, calculates wall interaction forces and potentials acting on particles within a surface cutoff distance.

wallpotentials()

```
subroutine surface_module::wallpotentials
```

Based on their positions relative to a reflecting hard wall boundary, calculates wall interaction potentials acting on particles within a surface cutoff distance. This subroutine does not assign forces to particles and is intended to calculate potentials, virials and stress tensor contributions at the start of a DPD simulation when particle forces are already known.

10.20 write_module.F90

10.20.1 Summary

Module for writing DL_MESO_DPD output files with system properties, trajectories.

10.20.2 Data Types

- `type` *type write_module::output_distribution*

  Data gathering and output filewriting parameters.

- `type` *type write_module::outputgroup*

  Data gathering and output filewriting group properties.

10.20.3 Functions/Subroutines

- `subroutine` *write_output_summary()*

  Writes simulation summary at current timestep to OUTPUT file.

- `subroutine` *write_output_equil()*

  Writes end-of-equilibration message to OUTPUT file.

- `subroutine` *write_correl()*

  Writes system properties to CORREL file.

- `subroutine` *write_stress()*

  Write separated stress tensor data to Stress_*.d files.

- `subroutine` *init_output_groups()*

  Initialises data gathering and file writing groups and communicators for HISTORY, export and CONFIG (CFGINI) files.

- `subroutine` *write_history_header()*

  Writes header information to new HISTORY file or prepares to write to existing HISTORY file (if restarting simulation).

- `subroutine` *gather_write_data()*

  Gather together particle data before writing to HISTORY and/or export file(s).

- `subroutine` *write_history()*

  Writes gathered particle data to HISTORY file.

- `subroutine` *write_export()*

  Writes gathered particle data to export file.

- `subroutine` *write_revive()*

  Writes statistical accumulators etc. to REVIVE file.
- `subroutine` *write_config()*

  Writes a CONFIG file of the current system configuration.
- `subroutine` *write_output_result()*

  Writes final summary of DPD simulation to OUTPUT file.

10.20.4 Variables

- `type(outputgroup), save` *group_info*

  Group information for data gathering and file writing.
- `integer(kind=li)` *filesize*

  Current size of HISTORY file in bytes.
- `integer(kind=li)` *markerpos*

  Location in HISTORY file to write filesize, number of trajectory frames and current timestep number.
- `integer(kind=li)` *headersize*

  Size of HISTORY file header before trajectory data in bytes.
- `integer` *numframe*

  Current number of trajectory frames in HISTORY file.

10.20.5 Data Type Documentation

type write_module::output_distribution

Table 10.3: Class Members

integer(kind=li)	chunksize	Maximum amount of data to be gathered per group in bytes
integer(kind=li)	framesize	Maximum amount of data to be gathered per particle in bytes
integer, dimension(:), allocatable	nodes_per_group	Number of processors in each data gathering group
integer	number_groups	Number of data gathering groups and file writing processors

type write_module::outputgroup

Table 10.4: Class Members

integer	comm	Communicator for data gathering group
type(*type write_module::output_distribution*)	distribution_info	Data gathering and file writing parameters
integer	group	Data gathering group identifier (not normally used)
integer, dimension(:), allocatable	members	Processor numbers in data gathering group
integer	my_group	Data gathering group number
integer	number_members	Number of processors in data gathering group
integer	rank	Rank of current processor in data gathering group
integer	root	Root processor for data gathering group
integer	writecomm1	Communicator for writing HISTORY file (if defined)
integer	writecomm2	Communicator for writing export file (if defined)
integer	writecomm3	Communicator for writing CONFIG (CFGINI) file (if defined)
integer	writegroup	File writing group identifier (not normally used)
integer, dimension(:), allocatable	writemembers	Processor numbers of file writing group
integer	writerank	Rank of current processor in file writing group (if defined)

Function/Subroutine Documentation

gather_write_data()

```
subroutine write_module::gather_write_data (logical, intent(in) lexport,
                                            logical, intent(in) lhistory,
                                            real(kind=dp) time
                                           )
```

Gather together particle positions, velocities and forces, global indices, species and molecule type numbers in preparation for writing this information to either the HISTORY file as a new trajectory frame or an export file as the current simulation configuration for restarts.

Parameters

lexport	Flag to indicate whether or not to write gathered data to export file
lhistory	Flag to indicate whether or not to write gathered data to HISTORY file
time	Current simulation time in DPD units

init_output_groups()

```
subroutine write_module::init_output_groups
```

Creates two sets of processor groups and communicators: one set to gather particle data among groups of processors to a root processor, and the other set to allow root processors to write data to files simultaneously. The number of processors per gathering group is determined based on the maximum amount of data to be gathered and the amount of data required per particle. The latter set of communicators are duplicated twice to allow HISTORY, export and CONFIG (CFGINI) files to be written without conflicts.

write_config()

```
subroutine write_module::write_config (character (len=*) configname,
                                        integer levdata
                                       )
```

Gathers together particle data (global indices, species, positions, velocities, forces) and writes to a text-formatted CONFIG file. The CONFIG file is predominately written using stream I/O, with each output group in parallel writing to the file simultaneously with MPI-IO after creating a string with particle data. Particles are sent to output group root processors according to their global indices, and their data is written in global index order. This subroutine is intended for use in writing the initial configuration of a new simulation with the file name CFGINI.

Parameters

config-name	Name of CONFIG file to be written
levdata	Level of data to be written to CONFIG file (equivalent to 'levcfg': 0 = positions, 1 = positions and velocities, 2 = positions, velocities and forces)

write_correl()

```
subroutine write_module::write_correl (real(kind=dp) time)
```

Writes instantaneous system properties to CORREL file at periodic intervals. A header with column titles is printed when the file is created.

Parameters

time	Current simulation time in DPD units

write_export()

```
subroutine write_module::write_export (integer, intent(in) total_beads,
                                        integer, dimension (:), intent(in) glab,
                                        integer, dimension (:), intent(in) gltp,
                                        integer, dimension (:), intent(in) gltm,
                                        real(kind=dp), dimension (:), intent(in)␣
↪gxx,
                                        real(kind=dp), dimension (:), intent(in)␣
↪gyy,
                                        real(kind=dp), dimension (:), intent(in)␣
↪gzz,
                                        real(kind=dp), dimension (:), intent(in)␣
↪gvx,
                                        real(kind=dp), dimension (:), intent(in)␣
↪gvy,
                                        real(kind=dp), dimension (:), intent(in)␣
↪gvz,
                                        real(kind=dp), dimension (:), intent(in)␣
↪gfx,
                                        real(kind=dp), dimension (:), intent(in)␣
↪gfy,
                                        real(kind=dp), dimension (:), intent(in) gfz
                                       )
```

Writes current simulation state - including simulation box size, Lees-Edwards shearing displacements, global particle indices and gathered particle data (positions, velocities, forces) - to export file as restart file. The export file is written using stream I/O, which does not require record-keeping characters and allows for easy searching. When writing in parallel, the root processors in each data gathering group (the processors in the file writing group) write to the export file simultaneously using MPI-IO, appending their group's data without sorting by particle number. Any previous export file is overwritten each time it is written to by this subroutine.

Parameters

total_beads	Total number of particles gathered by current processor
glab	Gathered global particle indices
gltp	Gathered particle species
gltm	Gathered molecule types for particles
gxx	Gathered particle positions (x-component)
gyy	Gathered particle positions (y-component)
gzz	Gathered particle positions (z-component)
gvx	Gathered particle velocities (x-component)
gvy	Gathered particle velocities (y-component)
gvz	Gathered particle velocities (z-component)
gfx	Gathered particle forces (x-component)
gfy	Gathered particle forces (y-component)
gfz	Gathered particle forces (z-component)

write_history()

```
subroutine write_module::write_history (integer, intent(in) total_beads,
                                         integer, dimension (:), intent(in) glab,
                                         real(kind=dp), dimension (:), intent(in)␣
↪gxx,
                                         real(kind=dp), dimension (:), intent(in)␣
↪gyy,
                                         real(kind=dp), dimension (:), intent(in)␣
↪gzz,
                                         real(kind=dp), dimension (:), intent(in)␣
↪gvx,
                                         real(kind=dp), dimension (:), intent(in)␣
↪gvy,
                                         real(kind=dp), dimension (:), intent(in)␣
↪gvz,
                                         real(kind=dp), dimension (:), intent(in)␣
↪gfx,
                                         real(kind=dp), dimension (:), intent(in)␣
↪gfy,
                                         real(kind=dp), dimension (:), intent(in)␣
↪gfz,
                                         real(kind=dp), intent(in) time
                                        )
```

Writes global particle indices and then gathered particle data (positions, velocities, forces - depending on required data level) to HISTORY file as simulation trajectory frame. The HISTORY file is written using stream I/O, which does not require record-keeping characters and allows for easy searching. When writing in parallel, the root processors in each data gathering group (the processors in the file writing group) write to the HISTORY file simultaneously using MPI-IO, appending their group's data without sorting by particle number. The previous file size and number of trajectory frames are overwritten with updated values to include the new frame written by this subroutine.

Parameters

total_beads	Total number of particles gathered by current processor
glab	Gathered global particle indices
gxx	Gathered particle positions (x-component)
gyy	Gathered particle positions (y-component)
gzz	Gathered particle positions (z-component)
gvx	Gathered particle velocities (x-component)
gvy	Gathered particle velocities (y-component)
gvz	Gathered particle velocities (z-component)
gfx	Gathered particle forces (x-component)
gfy	Gathered particle forces (y-component)
gfz	Gathered particle forces (z-component)
time	Current simulation time in DPD units

write_history_header()

```
subroutine write_module::write_history_header
```

For a new simulation, open a new HISTORY file and write header information, including specifications for particle species and molecules, particle species and molecule data, and bond connectivity information. For a restarted simulation, open the previously-created HISTORY file and find where the next trajectory frame should be written, i.e. after the last completed frame closest to the current timestep, overwriting any data previously written after this point in the file.

write_output_equil()

```
subroutine write_module::write_output_equil
```

Once equilibration has come to an end, writes a message to the OUTPUT file (or standard output) to indicate this.

write_output_result()

```
subroutine write_module::write_output_result
```

Writes the numbers of timesteps for simulation and statistical sampling, average system properties and standard deviations (fluctuations) - including stress tensors separated out into conservative, dissipative, random and kinetic contributions - simulation timings, final particle buffer sizes and a sample of particle positions and velocities to the OUTPUT file (or standard output).

write_output_summary()

```
subroutine write_module::write_output_summary (real(kind=dp) time,
                                               logical lbegin,
                                               logical l_scr
                                              )
```

Writes a summary of the simulation - timestep number, walltime and system properties (both instantaneous and rolling averages) - at periodic intervals to the OUTPUT file (or standard output).

Parameters

time	Current walltime for simulation
lbegin	Flag to indicate whether or not column titles should be printed before system data
l_scr	Flag to indicate if summary is to be printed to standard output instead of OUTPUT file

write_revive()

```
subroutine write_module::write_revive
```

Writes current simulation state - statistical accumulators of system properties, barostat properties and random number generator states - to REVIVE file as restart file. The REVIVE file is written using stream I/O, which does not require record-keeping characters and allows for easy searching. One core writes the accumulators and barostat properties (as these are replicated over all processors), while MPI-IO is used by all processors to write random number generator states.

write_stress()

```
subroutine write_module::write_stress (real(kind=dp) time)
```

Writes instantaneous values of stress tensors separated into interaction potential, dissipative, random and kinetic components to Stress_*.d files (Stress_pot.d, Stress_diss.d, Stress_rn.d and Stress_kin.d). Only files requested in CONTROL file will be created and written to.

Parameters

time	Current simulation time in DPD units

Variable Documentation

filesize

```
integer (kind=li) write_module::filesize
```

Current size of HISTORY file in bytes: updated value written into header of HISTORY file after each trajectory frame is written.

group_info

```
type (outputgroup), save write_module::group_info
```

Information about groups of processors used for gathering data and writing data to output files (especially HISTORY and export).

headersize

```
integer (kind=li) write_module::headersize
```

Total size of header (in bytes) at the start of the HISTORY file before trajectory data is added, including particle and bond identifications: used to identify locations for writing each trajectory frame in HISTORY file.

markerpos

```
integer (kind=li) write_module::markerpos
```

Location in HISTORY file (as byte number) where the filesize, number of trajectory frames and the current (last) timestep number are to be written.

numframe

```
integer write_module::numframe
```

Current number of frames in HISTORY: updated value written into header of HISTORY file after each trajectory frame is written.

10.21 integrate_dpd_mdvv.F90

10.21.1 Summary

Module for integrating forces with DPD thermostat using standard MD form of Velocity Verlet integration.

10.21.2 Functions/Subroutines

- `subroutine` *mdvv_nvt()*

  Applies a constant volume and temperature (NVT) ensemble by using standard (MD) Velocity Verlet force integration on both interaction and DPD thermostat forces.

- `subroutine` *mdvv_lang_npt()*

  Applies a constant pressure and temperature (NPT) ensemble by using standard (MD) Velocity Verlet force integration on both interaction and DPD thermostat forces, and a Langevin barostat.

- `subroutine` *mdvv_lang_npat()*

  Applies a constant pressure, surface area and temperature (NPAT) ensemble by using standard (MD) Velocity Verlet force integration on both interaction and DPD thermostat forces, and a Langevin barostat.

- `subroutine` *mdvv_lang_nst()*

  Applies a constant pressure, surface tension and temperature (NST) ensemble by using standard (MD) Velocity Verlet force integration on both interaction and DPD thermostat forces, and a Langevin barostat.

- `subroutine` *mdvv_berend_npt()*

  Applies a constant pressure and temperature (NPT) ensemble by using standard (MD) Velocity Verlet force integration on both interaction and DPD thermostat forces, and a Berendsen barostat.

- `subroutine` *mdvv_berend_npat()*

  Applies a constant pressure, surface area and temperature (NPAT) ensemble by using standard (MD) Velocity Verlet force integration on both interaction and DPD thermostat forces, and a Berendsen barostat.

- `subroutine` *mdvv_berend_nst()*

  Applies a constant pressure, surface tension and temperature (NST) ensemble by using standard (MD) Velocity Verlet force integration on both interaction and DPD thermostat forces, and a Berendsen barostat.

10.21.3 Function/Subroutine Documentation

mdvv_berend_npat()

```
subroutine integrate_dpd_mdvv::mdvv_berend_npat (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate both interaction forces and DPD thermostat (dissipative and random) forces for a constant pressure, surface area and temperature (NPAT) ensemble using a Berendsen barostat. This implementation holds the x- and y-components of box dimensions and positions constant and only varies the z-component based on the zz-component of the pressure tensor. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

mdvv_berend_npt()

```
subroutine integrate_dpd_mdvv::mdvv_berend_npt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate both interaction forces and DPD thermostat (dissipative and random) forces for a constant pressure and temperature (NPT) ensemble using a Berendsen barostat. This implementation keeps the system isotropic by using equal (averaged) values for pressure tensors. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

mdvv_berend_nst()

```
subroutine integrate_dpd_mdvv::mdvv_berend_nst (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate both interaction forces and DPD thermostat (dissipative and random) forces for a constant pressure, surface tension and temperature (NST) ensemble using a Berendsen barostat. This implementation can either apply the barostat independently in all three directions or semi-isotropically by using averaged pressure tensors for x- and y-components: in either case, the surface tension terms are applied in x- and y-directions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

mdvv_lang_npat()

```
subroutine integrate_dpd_mdvv::mdvv_lang_npat (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate both interaction forces and DPD thermostat (dissipative and random) forces for a constant pressure, surface area and temperature (NPAT) ensemble using a Langevin barostat. This implementation holds the x- and y-components of box dimensions and positions constant and only varies the z-component based on the zz-component of the pressure tensor. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

mdvv_lang_npt()

```
subroutine integrate_dpd_mdvv::mdvv_lang_npt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate both interaction forces and DPD thermostat (dissipative and random) forces for a constant pressure and temperature (NPT) ensemble using a Langevin barostat. This implementation keeps the system isotropic by using equal (averaged) values for pressure tensors and random numbers in all three dimensions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

mdvv_lang_nst()

```
subroutine integrate_dpd_mdvv::mdvv_lang_nst (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate both interaction forces and DPD thermostat (dissipative and random) forces for a constant pressure, surface tension and temperature (NST) ensemble using a Langevin barostat. This implementation can either apply the barostat independently in all three directions or semi-isotropically by using averaged pressure tensors and the same random piston force for x- and y-components: in either case, the surface tension terms are applied in x- and y-directions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

mdvv_nvt()

```
subroutine integrate_dpd_mdvv::mdvv_nvt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate both interaction forces and DPD thermostat (dissipative and random) forces for a constant volume and temperature (NVT) ensemble. Particle positions are adjusted after the first integration stage for any applicable boundary conditions - reciprocal vector maps for Ewald sums or SPME are recalculated if the system undergoes Lees-Edwards shearing.

Parameters

stage	Velocity Verlet stage (1 or 2)

10.22 integrate_dpd_dpdvv.F90

10.22.1 Summary

Module for integrating forces with DPD thermostat using DPD Velocity Verlet integration (recalculation of dissipative forces after force integration).

10.22.2 Functions/Subroutines

- `subroutine` *dpdvv_nvt()*

  Applies a constant volume and temperature (NVT) ensemble by using DPD Velocity Verlet force integration on both interaction and DPD thermostat forces, recalculating dissipative forces at end of timestep.

- `subroutine` *dpdvv_lang_npt()*

  Applies a constant pressure and temperature (NPT) ensemble by using DPD Velocity Verlet force integration on both interaction and DPD thermostat forces, and a Langevin barostat, recalculating dissipative forces at end of timestep.

- `subroutine` *dpdvv_lang_npat()*

  Applies a constant pressure, surface area and temperature (NPAT) ensemble by using DPD Velocity Verlet force integration on both interaction and DPD thermostat forces, and a Langevin barostat.

- `subroutine` *dpdvv_lang_nst()*

  Applies a constant pressure, surface tension and temperature (NST) ensemble by using DPD Velocity Verlet force integration on both interaction and DPD thermostat forces, and a Langevin barostat.

- `subroutine` *dpdvv_berend_npt()*

  Applies a constant pressure and temperature (NPT) ensemble by using DPD Velocity Verlet force integration on both interaction and DPD thermostat forces, and a Berendsen barostat, recalculating dissipative forces at end of timestep.

- `subroutine` *dpdvv_berend_npat()*

  Applies a constant pressure, surface area and temperature (NPAT) ensemble by using DPD Velocity Verlet force integration on both interaction and DPD thermostat forces, and a Berendsen barostat.

- `subroutine` *dpdvv_berend_nst()*

  Applies a constant pressure, surface tension and temperature (NST) ensemble by using DPD Velocity Verlet force integration on both interaction and DPD thermostat forces, and a Berendsen barostat.

10.22.3 Function/Subroutine Documentation

dpdvv_berend_npat()

```
subroutine integrate_dpd_dpdvv::dpdvv_berend_npat (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate both interaction forces and DPD thermostat (dissipative and random) forces for a constant pressure, surface area and temperature (NPAT) ensemble using a Berendsen barostat, and recalculating dissipative forces at the end of the timestep - this approach is known as DPD Velocity Verlet [9]. This implementation holds the x- and y-components of box dimensions and positions constant and only varies the z-component based on the zz-component of the pressure tensor. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpdvv_berend_npt()

```
subroutine integrate_dpd_dpdvv::dpdvv_berend_npt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate both interaction forces and DPD thermostat (dissipative and random) forces for a constant pressure and temperature (NPT) ensemble using a Berendsen barostat, and recalculating dissipative forces at the end of the timestep - this approach is known as DPD Velocity Verlet [9]. This implementation keeps the system isotropic by using equal (averaged) values for pressure tensors in all three dimensions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpdvv_berend_nst()

```
subroutine integrate_dpd_dpdvv::dpdvv_berend_nst (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate both interaction forces and DPD thermostat (dissipative and random) forces for a constant pressure, surface tension and temperature (NST) ensemble using a Berendsen barostat, and recalculating dissipative forces at the end of the timestep - this approach is known as DPD Velocity Verlet [9]. This implementation can either apply the barostat independently in all three directions or semi-isotropically by using averaged pressure tensors for x- and y-components: in either case, the surface tension terms are applied in x- and y-directions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpdvv_lang_npat()

```
subroutine integrate_dpd_dpdvv::dpdvv_lang_npat (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate both interaction forces and DPD thermostat (dissipative and random) forces for a constant pressure, surface area and temperature (NPAT) ensemble using a Langevin barostat, and recalculating dissipative forces at the end of the timestep - this approach is known as DPD Velocity Verlet [9]. This implementation holds the x- and y-components of box dimensions and positions constant and only varies the z-component based on the zz-component of the pressure tensor. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpdvv_lang_npt()

```
subroutine integrate_dpd_dpdvv::dpdvv_lang_npt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate both interaction forces and DPD thermostat (dissipative and random) forces for a constant pressure and temperature (NPT) ensemble using a Langevin barostat, and recalculating dissipative forces at the end of the timestep - this approach is known as DPD Velocity Verlet [9]. This implementation keeps the system isotropic by using equal (averaged) values for pressure tensors and random numbers in all three dimensions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpdvv_lang_nst()

```
subroutine integrate_dpd_dpdvv::dpdvv_lang_nst (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate both interaction forces and DPD thermostat (dissipative and random) forces for a constant pressure, surface tension and temperature (NST) ensemble using a Langevin barostat, and recalculating dissipative forces at the end of the timestep - this approach is known as DPD Velocity Verlet [9]. This implementation can either apply the barostat independently in all three directions or semi-isotropically by using averaged pressure tensors and the same random piston force for x- and y-components: in either case, the surface tension terms are applied in x- and y-directions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpdvv_nvt()

```
subroutine integrate_dpd_dpdvv::dpdvv_nvt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate both interaction forces and DPD thermostat (dissipative and random) forces for a constant volume and temperature (NVT) ensemble, and recalculating dissipative forces at the end of the timestep - this approach is known as DPD Velocity Verlet [9]. Particle positions are adjusted after the first integration stage for any applicable boundary conditions - reciprocal vector maps for Ewald sums or SPME are recalculated if the system undergoes Lees-Edwards shearing.

Parameters

stage	Velocity Verlet stage (1 or 2)

10.23 integrate_dpd_shardlow.F90

10.23.1 Summary

Module for integrating forces with DPD thermostat using Shardlow splitting.

10.23.2 Functions/Subroutines

- `subroutine` *dpds1_nvt()*

  Applies a constant volume and temperature (NVT) ensemble by using Velocity Verlet force integration on interaction forces and first-order Shardlow splitting on DPD thermostat forces.

- `subroutine` *dpds1_lang_npt()*

  Applies a constant pressure and temperature (NPT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and first-order Shardlow splitting on DPD thermostat forces, and a Langevin barostat.

- `subroutine` *dpds1_lang_npat()*

  Applies a constant pressure, surface area and temperature (NPAT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and first-order Shardlow splitting on DPD thermostat forces, and a Langevin barostat.

- `subroutine` *dpds1_lang_nst()*

  Applies a constant pressure, surface tension and temperature (NST) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and first-order Shardlow splitting on DPD thermostat forces, and a Langevin barostat.

- `subroutine` *dpds1_berend_npt()*

  Applies a constant pressure and temperature (NPT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and first-order Shardlow splitting on DPD thermostat forces, and a Berendsen barostat.

- `subroutine` *dpds1_berend_npat()*

  Applies a constant pressure, surface area and temperature (NPAT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and first-order Shardlow splitting on DPD thermostat forces, and a Berendsen barostat.

- `subroutine` *dpds1_berend_nst()*

  Applies a constant pressure, surface tension and temperature (NST) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and first-order Shardlow splitting on DPD thermostat forces, and a Berendsen barostat.

- `subroutine` *dpds2_nvt()*

  Applies a constant volume and temperature (NVT) ensemble by using Velocity Verlet force integration on interaction forces and second-order Shardlow splitting on DPD thermostat forces.

- `subroutine` *dpds2_lang_npt()*

  Applies a constant pressure and temperature (NPT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and second-order Shardlow splitting on DPD thermostat forces, and a Langevin barostat.

- `subroutine` *dpds2_lang_npat()*

  Applies a constant pressure, surface area and temperature (NPAT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and second-order Shardlow splitting on DPD thermostat forces, and a Langevin barostat.

- `subroutine` *dpds2_lang_nst()*

  Applies a constant pressure, surface tension and temperature (NST) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and second-order Shardlow splitting on DPD thermostat forces, and a Langevin barostat.

- `subroutine` *dpds2_berend_npt()*

  Applies a constant pressure and temperature (NPT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and second-order Shardlow splitting on DPD thermostat forces, and a Berendsen barostat.

- `subroutine` *dpds2_berend_npat()*

  Applies a constant pressure, surface area and temperature (NPAT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and second-order Shardlow splitting on DPD thermostat forces, and a Berendsen barostat.

- `subroutine` *dpds2_berend_nst()*

  Applies a constant pressure, surface tension and temperature (NST) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and second-order Shardlow splitting on DPD thermostat forces, and a Berendsen barostat.

10.23.3 Function/Subroutine Documentation

dpds1_berend_npat()

```
subroutine integrate_dpd_shardlow::dpds1_berend_npat (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces, and first-order Shardlow splitting [120] to integrate DPD thermostat (dissipative and random) forces at the start of the timestep for a constant pressure, surface area and temperature (NPAT) ensemble using a Berendsen barostat. This implementation holds the x- and y-components of box dimensions and positions constant and only varies the z-component based on the zz-component of the pressure tensor. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpds1_berend_npt()

```
subroutine integrate_dpd_shardlow::dpds1_berend_npt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces, and first-order Shardlow splitting [120] to integrate DPD thermostat (dissipative and random) forces at the start of the timestep for a constant pressure and temperature (NPT) ensemble using a Berendsen barostat. This implementation keeps the system isotropic by using equal (averaged) values for pressure tensors in all three dimensions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpds1_berend_nst()

```
subroutine integrate_dpd_shardlow::dpds1_berend_nst (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces, and first-order Shardlow splitting [120] to integrate DPD thermostat (dissipative and random) forces at the start of the timestep for a constant pressure, surface tension and temperature (NST) ensemble using a Berendsen barostat. This implementation can either apply the barostat independently in all three directions or semi-isotropically by using averaged pressure tensors for x- and y-components: in either case, the surface tension terms are applied in x- and y-directions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpds1_lang_npat()

```
subroutine integrate_dpd_shardlow::dpds1_lang_npat (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces, and first-order Shardlow splitting [120] to integrate DPD thermostat (dissipative and random) forces at the start of the timestep for a constant pressure, surface area and temperature (NPAT) ensemble using a Langevin barostat. This implementation holds the x- and y-components of box dimensions and positions constant and only varies the z-component based on the zz-component of the pressure tensor. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpds1_lang_npt()

```
subroutine integrate_dpd_shardlow::dpds1_lang_npt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces, and first-order Shardlow splitting [120] to integrate DPD thermostat (dissipative and random) forces at the start of the timestep for a constant pressure and temperature (NPT) ensemble using a Langevin barostat. This implementation keeps the system isotropic by using equal (averaged) values for pressure tensors and random numbers in all three dimensions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpds1_lang_nst()

```
subroutine integrate_dpd_shardlow::dpds1_lang_nst (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces, and first-order Shardlow splitting [120] to integrate DPD thermostat (dissipative and random) forces at the start of the timestep for a constant pressure, surface tension and temperature (NST) ensemble using a Langevin barostat. This implementation can either apply the barostat independently in all three directions or semi-isotropically by using averaged pressure tensors and the same random piston force for x- and y-components: in either case, the surface tension terms are applied in x- and y-directions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpds1_nvt()

```
subroutine integrate_dpd_shardlow::dpds1_nvt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces, and first-order Shardlow splitting [120] to integrate DPD thermostat (dissipative and random) forces at the start of the timestep for a constant volume and temperature (NVT) ensemble. Particle positions are adjusted after the first integration stage for any applicable boundary conditions - reciprocal vector maps for Ewald sums or SPME are recalculated if the system undergoes Lees-Edwards shearing.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpds2_berend_npat()

```
subroutine integrate_dpd_shardlow::dpds2_berend_npat (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces, and second-order Shardlow splitting [120] to integrate DPD thermostat (dissipative and random) forces at the start of the timestep and after the second Velocity Verlet stage for a constant pressure, surface area and temperature (NPAT) ensemble using a Berendsen barostat. This implementation holds the x- and y-components of box dimensions and positions constant and only varies the z-component based on the zz-component of the pressure tensor. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpds2_berend_npt()

```
subroutine integrate_dpd_shardlow::dpds2_berend_npt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces, and second-order Shardlow splitting [120] to integrate DPD thermostat (dissipative and random) forces at the start of the timestep and after the second Velocity Verlet stage for a constant pressure and temperature (NPT) ensemble using a Berendsen barostat. This implementation keeps the system isotropic by using equal (averaged) values for pressure tensors in all three dimensions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpds2_berend_nst()

```
subroutine integrate_dpd_shardlow::dpds2_berend_nst (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces, and second-order Shardlow splitting [120] to integrate DPD thermostat (dissipative and random) forces at the start of the timestep and after the second Velocity Verlet stage for a constant pressure, surface tension and temperature (NST) ensemble using a Berendsen barostat. This implementation can either apply the barostat independently in all three directions or semi-isotropically by using averaged pressure tensors for x- and y-components: in either case, the surface tension terms are applied in x- and y-directions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpds2_lang_npat()

```
subroutine integrate_dpd_shardlow::dpds2_lang_npat (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces, and second-order Shardlow splitting [120] to integrate DPD thermostat (dissipative and random) forces at the start of the timestep and after the second Velocity Verlet stage for a constant pressure, surface area and temperature (NPAT) ensemble using a Langevin barostat. This implementation holds the x- and y-components of box dimensions and positions constant and only varies the z-component based on the zz-component of the pressure tensor. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpds2_lang_npt()

```
subroutine integrate_dpd_shardlow::dpds2_lang_npt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces, and second-order Shardlow splitting [120] to integrate DPD thermostat (dissipative and random) forces at the start of the timestep and after the second Velocity Verlet stage for a constant pressure and temperature (NPT) ensemble using a Langevin barostat. This implementation keeps the system isotropic by using equal (averaged) values for pressure tensors and random numbers in all three dimensions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpds2_lang_nst()

```
subroutine integrate_dpd_shardlow::dpds2_lang_nst (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces, and second-order Shardlow splitting [120] to integrate DPD thermostat (dissipative and random) forces at the start of the timestep and after the second Velocity Verlet stage for a constant pressure, surface tension and temperature (NST) ensemble using a Langevin barostat. This implementation can either apply the barostat independently in all three directions or semi-isotropically by using averaged pressure tensors and the same random piston force for x- and y-components: in either case, the surface tension terms are applied in x- and y-directions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

dpds2_nvt()

```
subroutine integrate_dpd_shardlow::dpds2_nvt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces, and second-order Shardlow splitting [120] to integrate DPD thermostat (dissipative and random) forces at the start of the timestep and after the second Velocity Verlet stage for a constant volume and temperature (NVT) ensemble. Particle positions are adjusted after the first integration stage for any applicable boundary conditions - reciprocal vector maps for Ewald sums or SPME are recalculated if the system undergoes Lees-Edwards shearing.

Parameters

stage	Velocity Verlet stage (1 or 2)

10.24 integrate_lowe.F90

10.24.1 Summary

Module for integrating forces with the Lowe-Andersen thermostat.

10.24.2 Functions/Subroutines

- subroutine *lowe_correct()*

  Applies corrections to relative velocities between selected particle pairs for the Lowe-Andersen thermostat.

- subroutine *lowe_nvt()*

  Applies a constant volume and temperature (NVT) ensemble by using Velocity Verlet integration on interaction forces and Lowe-Andersen thermostat.

- subroutine *lowe_lang_npt()*

  Applies a constant pressure and temperature (NPT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Lowe-Andersen thermostat, and a Langevin barostat.

- subroutine *lowe_lang_npat()*

  Applies a constant pressure, surface area and temperature (NPAT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Lowe-Andersen thermostat, and a Langevin barostat.

- subroutine *lowe_lang_nst()*

  Applies a constant pressure, surface tension and temperature (NST) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Lowe-Andersen thermostat, and a Langevin barostat.

- subroutine *lowe_berend_npt()*

  Applies a constant pressure and temperature (NPT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Lowe-Andersen thermostat, and a Berendsen barostat.

- subroutine *lowe_berend_npat()*

  Applies a constant pressure, surface area and temperature (NPAT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Lowe-Andersen thermostat, and a Berendsen barostat.

- subroutine *lowe_berend_nst()*

  Applies a constant pressure, surface tension and temperature (NST) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Lowe-Andersen thermostat, and a Berendsen barostat.

10.24.3 Function/Subroutine Documentation

lowe_berend_npat()

```
subroutine integrate_lowe::lowe_berend_npat (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces and the Lowe-Andersen thermostat [83] for a constant pressure, surface area and temperature (NPAT) ensemble using a Berendsen barostat. This implementation holds the x- and y-components of box dimensions and positions constant and only varies the z-component based on the zz-component of the pressure tensor. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

lowe_berend_npt()

```
subroutine integrate_lowe::lowe_berend_npt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces and the Lowe-Andersen thermostat [83] for a constant pressure and temperature (NPT) ensemble using a Berendsen barostat. This implementation keeps the system isotropic by using equal (averaged) values for pressure tensors in all three dimensions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

lowe_berend_nst()

```
subroutine integrate_lowe::lowe_berend_nst (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces and the Lowe-Andersen thermostat [83] for a constant pressure, surface tension and temperature (NST) ensemble using a Berendsen barostat. This implementation can either apply the barostat independently in all three directions or semi-isotropically by using averaged pressure tensors for x- and y-components: in either case, the surface tension terms are applied in x- and y-directions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

lowe_correct()

```
subroutine integrate_lowe::lowe_correct (real(kind=dp) sdxx,
                                         real(kind=dp) sdxy,
                                         real(kind=dp) sdxz,
                                         real(kind=dp) sdyy,
                                         real(kind=dp) sdyz,
                                         real(kind=dp) sdzz,
                                         real(kind=dp) srxx,
                                         real(kind=dp) srxy,
                                         real(kind=dp) srxz,
                                         real(kind=dp) sryy,
                                         real(kind=dp) sryz,
                                         real(kind=dp) srzz
                                        )
```

Collects together data for thermostatting particle pairs (obtained while calculating pairwise interaction forces) across all processors, randomises order and replaces relative velocities between particle pairs with values taken from Maxwell-Boltzmann distribution for system temperature, applying the Lowe-Andersen thermostat [83]. Velocities for particles on different processors are shared by communications between those processors to complete the calculations. Virials and stresses resulting from this thermostat are calculated from velocity corrections.

Parameters

sdxx	Resulting xx-component of dissipative stress tensor from applying Lowe-Andersen thermostat
sdxy	Resulting xy-component (and yx-component) of dissipative stress tensor from applying Lowe-Andersen thermostat
sdxz	Resulting xz-component (and zx-component) of dissipative stress tensor from applying Lowe-Andersen thermostat
sdyy	Resulting yy-component of dissipative stress tensor from applying Lowe-Andersen thermostat
sdyz	Resulting yz-component (and zy-component) of dissipative stress tensor from applying Lowe-Andersen thermostat
sdzz	Resulting zz-component of dissipative stress tensor from applying Lowe-Andersen thermostat
srxx	Resulting xx-component of random stress tensor from applying Lowe-Andersen thermostat
srxy	Resulting xy-component (and yx-component) of random stress tensor from applying Lowe-Andersen thermostat
srxz	Resulting xz-component (and zx-component) of random stress tensor from applying Lowe-Andersen thermostat
sryy	Resulting yy-component of random stress tensor from applying Lowe-Andersen thermostat
sryz	Resulting yz-component (and zy-component) of random stress tensor from applying Lowe-Andersen thermostat
srzz	Resulting zz-component of random stress tensor from applying Lowe-Andersen thermostat

lowe_lang_npat()

```
subroutine integrate_lowe::lowe_lang_npat (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces and the Lowe-Andersen thermostat [83] for a constant pressure, surface area and temperature (NPAT) ensemble using a Langevin barostat. This implementation holds the x- and y-components of box dimensions and positions constant and only varies the z-component based on the zz-component of the pressure tensor. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

lowe_lang_npt()

```
subroutine integrate_lowe::lowe_lang_npt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces and the Lowe-Andersen thermostat [83] for a constant pressure and temperature (NPT) ensemble using a Langevin barostat. This implementation keeps the system isotropic by using equal (averaged) values for pressure tensors and random numbers in all three dimensions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

lowe_lang_nst()

```
subroutine integrate_lowe::lowe_lang_nst (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces and the Lowe-Andersen thermostat [83] for a constant pressure, surface tension and temperature (NST) ensemble using a Langevin barostat. This implementation can either apply the barostat independently in all three directions or semi-isotropically by using averaged pressure tensors and the same random piston force for x- and y-components: in either case, the surface tension terms are applied in x- and y-directions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

lowe_nvt()

```
subroutine integrate_lowe::lowe_nvt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces and the Lowe-Andersen thermostat [83] to apply a constant volume and temperature (NVT) ensemble. Particle positions are adjusted after the first integration stage for any applicable boundary conditions: reciprocal vector maps for Ewald sums or SPME are recalculated if the system undergoes Lees-Edwards shearing.

Parameters

stage	Velocity Verlet stage (1 or 2)

10.25 integrate_peters.F90

10.25.1 Summary

Module for integrating forces with the Peters thermostat.

10.25.2 Functions/Subroutines

- `subroutine` *peters_correct()*

  Applies corrections to relative velocities between selected particle pairs for the Peters thermostat.

- `subroutine` *peters_nvt()*

  Applies a constant volume and temperature (NVT) ensemble by using Velocity Verlet integration on interaction forces and Peters thermostat.

- `subroutine` *peters_lang_npt()*

  Applies a constant pressure and temperature (NPT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Peters thermostat, and a Langevin barostat.

- `subroutine` *peters_lang_npat()*

  Applies a constant pressure, surface area and temperature (NPAT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Peters thermostat, and a Langevin barostat.

- `subroutine` *peters_lang_nst()*

  Applies a constant pressure, surface tension and temperature (NST) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Peters thermostat, and a Langevin barostat.

- `subroutine` *peters_berend_npt()*

  Applies a constant pressure and temperature (NPT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Peters thermostat, and a Berendsen barostat.

- `subroutine` *peters_berend_npat()*

  Applies a constant pressure, surface area and temperature (NPAT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Peters thermostat, and a Berendsen barostat.

- `subroutine` *peters_berend_nst()*

  Applies a constant pressure, surface tension and temperature (NST) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Peters thermostat, and a Berendsen barostat.

10.25.3 Function/Subroutine Documentation

peters_berend_npat()

```
subroutine integrate_peters::peters_berend_npat (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces and the Peters thermostat [100] for a constant pressure, surface area and temperature (NPAT) ensemble using a Berendsen barostat. This implementation holds the x- and y-components of box dimensions and positions constant and only varies the z-component based on the zz-component of the pressure tensor. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

peters_berend_npt()

```
subroutine integrate_peters::peters_berend_npt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces and the Peters thermostat [100] for a constant pressure and temperature (NPT) ensemble using a Berendsen barostat. This implementation keeps the system isotropic by using equal (averaged) values for pressure tensors in all three dimensions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

peters_berend_nst()

```
subroutine integrate_peters::peters_berend_nst (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces and the Peters thermostat [100] for a constant pressure, surface tension and temperature (NST) ensemble using a Berendsen barostat. This implementation can either apply the barostat independently in all three directions or semi-isotropically by using averaged pressure tensors for x- and y-components: in either case, the surface tension terms are applied in x- and y-directions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

peters_correct()

```
subroutine integrate_peters::peters_correct (real(kind=dp) sdxx,
                                              real(kind=dp) sdxy,
                                              real(kind=dp) sdxz,
                                              real(kind=dp) sdyy,
                                              real(kind=dp) sdyz,
                                              real(kind=dp) sdzz,
                                              real(kind=dp) srxx,
                                              real(kind=dp) srxy,
                                              real(kind=dp) srxz,
                                              real(kind=dp) sryy,
                                              real(kind=dp) sryz,
                                              real(kind=dp) srzz
                                             )
```

Collects together data for thermostatting particle pairs (obtained while calculating pairwise interaction forces) across all processors, randomises order and applies changes to velocities for particle pairs, applying the Peters thermostat [100]. Velocities for particles on different processors are shared by communications between those processors to complete the calculations. Virials and stresses resulting from this thermostat are calculated from velocity corrections.

Parameters

sdxx	Resulting xx-component of dissipative stress tensor from applying Peters thermostat
sdxy	Resulting xy-component (and yx-component) of dissipative stress tensor from applying Peters thermostat
sdxz	Resulting xz-component (and zx-component) of dissipative stress tensor from applying Peters thermostat
sdyy	Resulting yy-component of dissipative stress tensor from applying Peters thermostat
sdyz	Resulting yz-component (and zy-component) of dissipative stress tensor from applying Peters thermostat
sdzz	Resulting zz-component of dissipative stress tensor from applying Peters thermostat
srxx	Resulting xx-component of random stress tensor from applying Peters thermostat
srxy	Resulting xy-component (and yx-component) of random stress tensor from applying Peters thermostat
srxz	Resulting xz-component (and zx-component) of random stress tensor from applying Peters thermostat
sryy	Resulting yy-component of random stress tensor from applying Peters thermostat
sryz	Resulting yz-component (and zy-component) of random stress tensor from applying Peters thermostat
srzz	Resulting zz-component of random stress tensor from applying Peters thermostat

peters_lang_npat()

```
subroutine integrate_peters::peters_lang_npat (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces and the Peters thermostat [100] for a constant pressure, surface area and temperature (NPAT) ensemble using a Langevin barostat. This implementation holds the x- and y-components of box dimensions and positions constant and only varies the z-component based on the zz-component of the pressure tensor. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

peters_lang_npt()

```
subroutine integrate_peters::peters_lang_npt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces and the Peters thermostat [100] for a constant pressure and temperature (NPT) ensemble using a Langevin barostat. This implementation keeps the system isotropic by using equal (averaged) values for pressure tensors and random numbers in all three dimensions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

peters_lang_nst()

```
subroutine integrate_peters::peters_lang_nst (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces and the Peters thermostat [100] for a constant pressure, surface tension and temperature (NST) ensemble using a Langevin barostat. This implementation can either apply the barostat independently in all three directions or semi-isotropically by using averaged pressure tensors and the same random piston force for x- and y-components: in either case, the surface tension terms are applied in x- and y-directions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

peters_nvt()

```
subroutine integrate_peters::peters_nvt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction forces and the Peters thermostat [100] to apply a constant volume and temperature (NVT) ensemble. Particle positions are adjusted after the first integration stage for any applicable boundary conditions: reciprocal vector maps for Ewald sums or SPME are recalculated if the system undergoes Lees-Edwards shearing.

Parameters

stage	Velocity Verlet stage (1 or 2)

10.26 integrate_stoyanov.F90

10.26.1 Summary

Module for integrating forces with the Stoyanov-Groot thermostat.

10.26.2 Functions/Subroutines

- `subroutine` *stoyanov_correct()*

  Applies corrections to relative velocities between selected particle pairs for the Stoyanov-Groot thermostat.

- `subroutine` *stoyanov_nvt()*

  Applies a constant volume and temperature (NVT) ensemble by using Velocity Verlet integration on interaction forces and Stoyanov-Groot thermostat.

- `subroutine` *stoyanov_lang_npt()*

  Applies a constant pressure and temperature (NPT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Stoyanov-Groot thermostat, and a Langevin barostat.

- `subroutine` *stoyanov_lang_npat()*

  Applies a constant pressure, surface area and temperature (NPAT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Stoyanov-Groot thermostat, and a Langevin barostat.

- `subroutine` *stoyanov_lang_nst()*

  Applies a constant pressure, surface tension and temperature (NST) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Stoyanov-Groot thermostat, and a Langevin barostat.

- `subroutine` *stoyanov_berend_npt()*

  Applies a constant pressure and temperature (NPT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Stoyanov-Groot thermostat, and a Berendsen barostat.

- `subroutine` *stoyanov_berend_npat()*

  Applies a constant pressure, surface area and temperature (NPAT) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Stoyanov-Groot thermostat, and a Berendsen barostat.

- `subroutine` *stoyanov_berend_nst()*

  Applies a constant pressure, surface tension and temperature (NST) ensemble by using standard (MD) Velocity Verlet force integration on interaction forces and Stoyanov-Groot thermostat, and a Berendsen barostat.

10.26.3 Function/Subroutine Documentation

stoyanov_berend_npat()

```
subroutine integrate_stoyanov::stoyanov_berend_npat (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction and Nosé-Hoover temperature-dependent forces with the Lowe-Andersen thermostat (known as the Stoyanov-Groot thermostat) [131] for a constant pressure, surface area and temperature (NPAT) ensemble using a Berendsen barostat. This implementation holds the x- and y-components of box dimensions and positions constant and only varies the z-component based on the zz-component of the pressure tensor. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

stoyanov_berend_npt()

```
subroutine integrate_stoyanov::stoyanov_berend_npt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction and Nosé-Hoover temperature-dependent forces with the Lowe-Andersen thermostat (known as the Stoyanov-Groot thermostat) [131] for a constant pressure and temperature (NPT) ensemble using a Berendsen barostat. This implementation keeps the system isotropic by using equal (averaged) values for pressure tensors in all three dimensions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

stoyanov_berend_nst()

```
subroutine integrate_stoyanov::stoyanov_berend_nst (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction and Nosé-Hoover temperature-dependent forces with the Lowe-Andersen thermostat (known as the Stoyanov-Groot thermostat) [131] for a constant pressure, surface tension and temperature (NST) ensemble using a Berendsen barostat. This implementation can either apply the barostat independently in all three directions or semi-isotropically by using averaged pressure tensors for x- and y-components: in either case, the surface tension terms are applied in x- and y-directions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles.

Parameters

stage	Velocity Verlet stage (1 or 2)

stoyanov_correct()

```
subroutine integrate_stoyanov::stoyanov_correct (real(kind=dp) sdxx,
                                                  real(kind=dp) sdxy,
                                                  real(kind=dp) sdxz,
                                                  real(kind=dp) sdyy,
                                                  real(kind=dp) sdyz,
                                                  real(kind=dp) sdzz,
                                                  real(kind=dp) srxx,
                                                  real(kind=dp) srxy,
                                                  real(kind=dp) srxz,
                                                  real(kind=dp) sryy,
                                                  real(kind=dp) sryz,
                                                  real(kind=dp) srzz
                                                 )
```

Collects together data for thermostatting particle pairs (obtained while calculating pairwise interaction forces) across all processors, randomises order and replaces relative velocities between particle pairs with values taken from Maxwell-Boltzmann distribution for system temperature, applying the Lowe-Andersen part of the Stoyanov-Groot thermostat [131]. Velocities for particles on different processors are shared by communications between those processors to complete the calculations. Virials and stresses resulting from this thermostat are calculated from velocity corrections.

Parameters

sdxx	Resulting xx-component of dissipative stress tensor from applying Lowe-Andersen part of thermostat
sdxy	Resulting xy-component (and yx-component) of dissipative stress tensor from applying Lowe-Andersen part of thermostat
sdxz	Resulting xz-component (and zx-component) of dissipative stress tensor from applying Lowe-Andersen part of thermostat
sdyy	Resulting yy-component of dissipative stress tensor from applying Lowe-Andersen part of thermostat
sdyz	Resulting yz-component (and zy-component) of dissipative stress tensor from applying Lowe-Andersen part of thermostat
sdzz	Resulting zz-component of dissipative stress tensor from applying Lowe-Andersen part of thermostat
srxx	Resulting xx-component of random stress tensor from applying Lowe-Andersen part of thermostat
srxy	Resulting xy-component (and yx-component) of random stress tensor from applying Lowe-Andersen part of thermostat
srxz	Resulting xz-component (and zx-component) of random stress tensor from applying Lowe-Andersen part of thermostat
sryy	Resulting yy-component of random stress tensor from applying Lowe-Andersen part of thermostat
sryz	Resulting yz-component (and zy-component) of random stress tensor from applying Lowe-Andersen part of thermostat
srzz	Resulting zz-component of random stress tensor from applying Lowe-Andersen part of thermostat

stoyanov_lang_npat()

```
subroutine integrate_stoyanov::stoyanov_lang_npat (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction and Nosé-Hoover temperature-dependent forces with the Lowe-Andersen thermostat (known as the Stoyanov-Groot thermostat) [131] for a constant pressure, surface area and temperature (NPAT) ensemble using a Langevin barostat. This implementation holds the x- and y-components of box dimensions and positions constant and only varies the z-component based on the zz-component of the pressure tensor. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

stoyanov_lang_npt()

```
subroutine integrate_stoyanov::stoyanov_lang_npt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction and Nosé-Hoover temperature-dependent forces with the Lowe-Andersen thermostat (known as the Stoyanov-Groot thermostat) [131] for a constant pressure and temperature (NPT) ensemble using a Langevin barostat. This implementation keeps the system isotropic by using equal (averaged) values for pressure tensors and random numbers in all three dimensions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

stoyanov_lang_nst()

```
subroutine integrate_stoyanov::stoyanov_lang_nst (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction and Nosé-Hoover temperature-dependent forces with the Lowe-Andersen thermostat (known as the Stoyanov-Groot thermostat) [131] for a constant pressure, surface tension and temperature (NST) ensemble using a Langevin barostat. This implementation can either apply the barostat independently in all three directions or semi-isotropically by using averaged pressure tensors and the same random piston force for x- and y-components: in either case, the surface tension terms are applied in x- and y-directions. Particle positions and system volume are rescaled during the first integration stage, applying any boundary conditions. Reciprocal vector values for Ewald sums or SPME are always recalculated, as are corrective forces for any charged frozen particles. The barostat piston forces and final particle velocities are calculated iteratively [64] until the latter converge.

Parameters

stage	Velocity Verlet stage (1 or 2)

stoyanov_nvt()

```
subroutine integrate_stoyanov::stoyanov_nvt (integer stage)
```

Solves the equations of motion using the Velocity Verlet scheme [141] to integrate interaction and Nosé-Hoover temperature-dependent forces with the Lowe-Andersen thermostat (known as the Stoyanov-Groot thermostat) [131] to apply a constant volume and temperature (NVT) ensemble. Particle positions are adjusted after the first integration stage for any applicable boundary conditions: reciprocal vector maps for Ewald sums or SPME are recalculated if the system undergoes Lees-Edwards shearing.

Parameters

stage	Velocity Verlet stage (1 or 2)

CHAPTER

ELEVEN

DL_MESO_DPD INPUT AND OUTPUT 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, tabs or commas.

11.1 CONTROL

This ANSI text input file contains the control variables for running a DPD simulation. It consists 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. Keywords and numerical values can be separated from each other with spaces, tabs or commas.

While not every directive has to be included for a valid simulation - many hold default values if unspecified - the following are mandatory and must be set to values greater than zero:

- `temperature`
- `timestep`
- `volume` (if no *CONFIG* file is available or the supplied file omits volume data)

while `pressure` must also be specified if using an ensemble other than NVT (i.e. a barostat is involved), even if the value is set to zero. The maximum interaction `cutoff` should also be set greater than zero, but this value can be derived from the maximum interaction length (σ_{ij} or $r_{c,ij}$) specified in the *FIELD* file if it is omitted here.

A list of the available directives follows. Some directives may include optional words or parameters as indicated by brackets that can safely be omitted. The *read_control()* and *scan_control()* subroutines that read and scan the CONTROL file often search for only the first few letters in each directive keyword, but use of the full words and their given spellings is strongly recommended. Further details about these keywords are given in Chapter 10 of the DL_MESO User Manual. Keywords of superfluous properties for the given simulation (e.g. pressure for constant volume simulations) can be omitted, while any number left out of a directive will be assumed to be zero.

Note that if there are duplicate entries for any directive, the values associated with the last one in the file will be used.

Table 11.1: Available keywords in CONTROL input file

directive	meaning
`bjerrum (length)` f	sets the Bjerrum length $\lambda_B = \frac{\Gamma r_c}{4\pi}$ for the system to f
`boundary halo` f	sets size of boundary halo (overriding default values) as f length units
`close time` f	sets job closure time to f seconds
`config (origin) zero`	sets origin of *CONFIG* file as bottom left back corner instead of default of box centre
`cutoff` f	sets maximum interaction cutoff radius, $r_{c,max}$, to f length units

continues on next page

Table 11.1 – continued from previous page

directive	meaning
`densvar` f	allows for local variation of $\approx f$ % in the system density of particles (default $f = 0$)
`electrostatic cutoff` f	sets required short-range electrostatic cutoff radius, r_e, to f length units (default $f = r_c$)
`ensemble nvt mdvv`	selects NVT (constant volume/temperature) ensemble, DPD thermostat with standard MD-like Velocity Verlet integration (default ensemble if otherwise not specified)
`ensemble nvt dpdvv`	selects NVT ensemble, DPD thermostat with DPD Velocity Verlet integration
`ensemble nvt dpds1`	selects NVT ensemble, DPD thermostat with first-order Shardlow splitting
`ensemble nvt dpds2`	selects NVT ensemble, DPD thermostat with second-order Shardlow splitting
`ensemble nvt lowe`	selects NVT ensemble, Lowe-Andersen thermostat
`ensemble nvt peters`	selects NVT ensemble, Peters thermostat
`ensemble nvt stoyanov` α	selects NVT ensemble, Stoyanov-Groot thermostat with coupling parameter α
`ensemble npt` Q `langevin` f_1 f_2	selects NPT (constant pressure/temperature) ensemble, thermostat type Q (i.e. `mdvv`, `dpdvv`, `dpds1`, `dpds2`, `lowe`, `peters` or `stoyanov` α) with Langevin barostat, setting relaxation time (τ_p) and viscosity parameter (γ_p) as f_1 and f_2
`ensemble npt` Q `berendsen` f	selects NPT ensemble, thermostat type Q with Berendsen barostat, setting ratio of compressibility to relaxation time, $\frac{\beta}{\tau_p}$, to f
`ensemble nst` Q `langevin` f_1 f_2	selects (orthogonally constrained) NσT (constant stress/temperature) ensemble, thermostat type Q with Langevin barostat, setting relaxation time (τ_p) and viscosity parameter (γ_p) as f_1 and f_2 respectively
`ensemble nst` Q `berendsen` f	selects (orthogonally constrained) NσT ensemble, thermostat type Q with Berendsen barostat, setting ratio of compressibility to relaxation time, $\frac{\beta}{\tau_p}$, to f
`ensemble nst` Q_1 Q_2 `area`	selects NP$_n$AT (constant normal pressure/surface area/temperature) ensemble, thermostat type Q_1 and barostat type Q_2 (i.e. `langevin` f_1 f_2 or `berendsen` f)
`ensemble nst` Q_1 Q_2 `tension` γ_0	selects NP$_n\gamma$T (constant normal pressure/surface tension/temperature) anisotropic ensemble, thermostat type Q_1 and barostat type Q_2 with target surface tension γ_0
`ensemble nst` Q_1 Q_2 `tension` γ_0 `semi`	selects NP$_n\gamma$T (constant normal pressure/surface tension/temperature) semi-isotropic ensemble, thermostat type Q_1 and barostat type Q_2 with target surface tension γ_0, varying the (x, y) plane isotropically
`ensemble nst` Q_1 Q_2 `orthogonal`	selects NP$_n\gamma$T (constant normal pressure/surface tension/temperature) anisotropic ensemble, thermostat type Q_1 and barostat type Q_2 with target surface tension $\gamma_0 = 0$[1]
`ensemble nst` Q_1 Q_2 `orth semi`	selects NP$_n\gamma$T (constant normal pressure/surface tension/temperature) semi-isotropic ensemble, thermostat type Q_1 and barostat type Q_2 with target surface tension $\gamma_0 = 0$

continues on next page

Table 11.1 – continued from previous page

directive	meaning
`equilibration (steps)` n	equilibrates system for the first n timesteps (default $n = 0$)
`ewald precision` f	calculates electrostatic forces using Ewald sum, setting the real-space convergence parameter α and reciprocal space (k-vector) range (k_1, k_2, k_3) to achieve the relative potential error f
`ewald (sum)` α k_1 k_2 k_3	calculates electrostatic forces using Ewald sum with real-space convergence parameter α and reciprocal space (k-vector) range (k_1, k_2, k_3)
`finish`	closes the CONTROL file (last data record)
`frozen (wall)` i	sets frozen bead walls orthogonal to i-axis (`x`, `y`, `z`) if specified[2]
`global bonds`	calculates bonded interactions globally by storing bond data on all processors and sharing bonded particle positions (default: calculate bonded interactions locally)
`job time` f	sets maximum job time (including closure) to f seconds
`l_init`	creates *CONFIG* file (CFGINI) of initial system configuration (with velocities and forces) for new simulations[3]
`l_scr`	redirects simulation output to the standard (default) output of the machine and operating system, e.g. the screen
`manybody cutoff` f	sets required many-body DPD interaction radius, r_d, to f length units (default $f = r_c$)
`ndump` n	writes restart data to *export* and *REVIVE* files every n timesteps (default $n = 1000$)
`nfold` i j k	option to create volumetrically expanded version of current system by replicating configuration described by *CONFIG* file (i, j, k) times, preserving topology of *FIELD* file[4]
`no config`	ignores contents of *CONFIG* file and create initial configuration based purely on *FIELD* file
`no electrostatics`	ignores electrostatics in simulation
`no index`	ignores particles' indices in *CONFIG* file and sets indices according to reading order
`openmp (critical)`	uses OpenMP critical regions to assign forces in multithreaded calculations (instead of using additional memory per thread)
`permittivity (constant)` f	sets permittivity constant for system, $\Gamma = \frac{4\pi\lambda_B}{r_c}$, to f
`pressure` f	sets required system pressure to f (target pressure for constant pressure ensembles)
`print (every)` n	prints system data every n timesteps
`print partial (temperatures)`	prints partial temperatures (for each dimension) in both *OUTPUT* and *CORREL*. (Option automatically switched on if a constant force or shear is applied to the system.)
`rcut` f	see `cutoff`
`restart`	restarts job from end point of previous run (i.e. continue current simulation using *export* and *REVIVE* files)

continues on next page

Table 11.1 – continued from previous page

directive	meaning
`restart noscale`	restarts job from previous run without rescaling to system temperature (i.e. begin a new simulation from older run without temperature reset using configuration in *export* file)
`restart scale`	restarts job from previous run after rescaling to system temperature (i.e. begin a new simulation from older run with temperature reset using configuration in *export* file)
`scale` (`temperature`) (`every`) n	rescales system temperature every n steps during equilibration
`seed` n	modifies random number generator seeds used in DPD calculations
`smear` Q	applies charge smearing type to Q (`none`, `linear`, `slater` (`exact`), `slater approx`, `gauss` or `sinusoidal`)
`smear beta` f (Q)	sets Slater-type smearing parameter β: can be followed by Q (`original`, `overlap` or `distribution`) to specify how β is related to the smearing length
`smear length` f	sets smearing length (R, λ, σ_G or D) to f
`smear length` f `equal`	sets smearing length for Gaussian smearing (σ_G) to f and set Ewald real-space convergence parameter $\alpha = \frac{1}{2\sigma_G}$ to override any other specified value[5]
`smear length` f Q	sets smearing length for Slater-type smearing (λ) to f, with Q (`original`, `overlap` or `distribution`) indicating how β is related to the smearing length λ
`spme precision` f (n)	calculates electrostatic forces using Smooth Particle Mesh Ewald, setting the real-space convergence parameter α and reciprocal space (k-vector) range (k_1, k_2, k_3) to achieve the relative potential error f and maximum B-spline order n (default: 8 if omitted or less than 4)
`spme` (`sum`) α k_1 k_2 k_3 (n)	calculates electrostatic forces using Smooth Particle Mesh Ewald with real-space convergence parameter α, reciprocal space (k-vector) range (k_1, k_2, k_3)[6] and maximum B-spline order n (default: 8 if omitted or less than 4)
`stack` (`size`) n	sets rolling average stack to n timesteps
`stats` (`every`) n	accumulates statistics data and writes to *CORREL* file every n timesteps
`steps` n	runs simulation for n timesteps
`stress` i j Q	accumulates separated stress tensors and writes to *Stress_*.d* file(s) with controls: i = start timestep for writing stress tensors, j = timestep interval, Q = (up to four) separated stress tensor sets to write to files (`potential`, `dissipative`, `random`, `kinetic`, `all` four)
`surface cutoff` f	sets required maximum surface repulsive range, z_c, to f length units (default $f = r_c$)
`surface hard` i `bounceback` (f)	sets hard adsorbing walls orthogonal to i-axis (`x`, `y`, `z`) if specified with bounce-back reflections (optionally at a distance f from simulation boundaries, default $f = 0$)

continues on next page

Table 11.1 – continued from previous page

directive	meaning
`surface hard` i `(specular)` (f)	sets hard adsorbing walls orthogonal to i-axis (`x`, `y`, `z`) if specified with specular reflections (optionally at a distance f from simulation boundaries, default $f =$ 0)
`surface shear` i j	sets moving Lees-Edwards periodic walls orthogonal to i-axis (`x`, `y`, `z`) if specified[7], starting shearing from time step j
`temperature` f	sets required simulation temperature ($k_B T$) to f
`thermostat cutoff` f	sets thermostat cutoff radius, r_c, to f length units
`trajectory` i j (k)	writes trajectory data to *HISTORY* file with controls: i = start timestep for dumping configurations (default: equilibration time), j = timestep interval between configurations, k = data level to be included (default: 0; 0 = positions, 1 adds velocities, 2 adds forces)
`timestep` f	set timestep to f time units
`vacuum (gap)` f_1 f_2 f_3	sets size of vacuum gap (additional volume) for reciprocal space Ewald/SPME calculations of slab-like systems as orthorhombic dimensions (f_1, f_2, f_3)
`volume` f_1 (f_2 f_3)	sets system size to either cubic volume f_1 or orthorhombic dimensions (f_1, f_2, f_3)

This file is compulsory for a DL_MESO_DPD calculation and must be supplied in the same directory where DL_MESO_DPD is run. It can be created or modified either by hand using a text editor or by using the DL_MESO GUI (recommended when starting to use DL_MESO_DPD).

11.2 FIELD

This ANSI text input file contains the species and force field information required for both bonded and unbonded interactions in a DPD simulation. Apart from the first line indicating the simulation name (as for the *CONTROL* file), it consists of blocks of information, each headed by a directive indicating the type and (in most cases) the number of interactions to follow in specific formats for the directives in question. Keywords and numerical values can be separated from each other with spaces, tabs or commas.

[1] This option is equivalent to the orthogonal NσT ensemble and is included for equivalence to DL_POLY.

[2] This directive can only be used for new simulations and is ignored if a simulation is restarted.

[3] If simulation is started using a *CONFIG* file with velocities and forces without volumetric expansion, this option is ignored.

[4] If restarting a system with an *export* file that was originally set up this way, this option can be used to avoid modifying the *FIELD* file.

[5] The real space terms for the Ewald sum are omitted when this option is selected.

[6] This vector should be double the size of that used for the standard Ewald sum, as the k-vector is applied cubically rather than spherically.

[7] Unlike frozen bead and hard surfaces, only one shearing boundary can be specified: if multiple axes are specified, DL_MESO_DPD will use the first one.

Table 11.2: Available directives in FIELD input file

directive	meaning
`units constant`	Sets units for DPD simulation to use force and energy parameters without scaling (default)
`units kT`	Sets units for DPD simulation to use force and energy parameters scaled by temperature specified in *CONTROL* file
`species` n	Sets information for n available particle species
`interactions` n	Sets (intermolecular) non-bonded interactions between particle pairs for n pairs of particle species
`molecules` n	Specifies n molecule definitions with name, particles, bonds, angles and dihedrals (see below)
`surface` n	Sets surface interactions between solid walls and particles for n particle species
`frozen SPEC` ρ_{wall}:math:`x_{wall}	Sets frozen walls to use particles of species `SPEC` with density ρ_{wall} and thickness of :math:`x_{wall} DPD length units
`external`	Specifies external fields applied to particles: gravitional acceleration, electric fields on charged particles and/or linear shear
`close`	Indicates end of interaction data in file

Each line of the `species` data is given by 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)

where all numbers beyond the particle mass are optional and will be read as zero if not supplied, e.g. no species population contained entirely in molecules needs to be given as this can be worked out from molecule definitions. Only the first 8 characters of the species name will be used: DL_MESO_DPD will warn the user if two or more species have the same truncated names.

Each line of the `interaction` data is given as:

species 1	a8	name of species 1
species 2	a8	name of species 2
key	a4	interaction key
variables 1-7	real	interaction and thermostat parameters

where the interaction key (up to 4 characters long) indicates the type of non-bonded interaction (`lj`, `wca`, `dpd` or `mdpd`) between the two species, which requires up to six interaction parameters (including an interaction length) and a thermostat parameter, either the dissipative force parameter γ_{ij} or a collision frequency Γ_{ij}. (Full details of what is required for each interaction type are given in Chapter 10 of the DL_MESO User Manual.)

Each `surface` interaction line uses a similar format to the interactions above:

species	a8	name of species
key	a4	interaction key
variables 1 and 2	real	interaction parameters

with similar interaction keys (`dpd` and `wca`) and only two parameters required per species.

The `molecules` directive is followed by n blocks of data, one for each molecule type. Each block starts with the molecule name (a string of up to 8 characters) and is followed by the following sub-directives, all given in any order apart from `finish` which concludes the block:

Table 11.3: Available molecule sub-directives in FIELD input file

sub-directive	meaning
`nummols` n	Gives the number of molecules of the current type in the system (molecule population)
`beads` n	Indicates the number of particles per molecules (followed by details about each particle, see below)
`no isomer`	Indicates the molecule shape should not be reflected/inverted when added to a new simulation
`bonds` n	Indicates the number of bonds included in each molecule (followed by details, see below)
`angles` n	Indicates the number of bond angles included in each molecule (followed by details, see below)
`dihedrals` n	Indicates the number of bonds included in each molecule (followed by details, see below)
`finish`	Indicates end of information for current molecule type

Each line of information indicating the particles involved in each molecule type is given by the following format:

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

where the relative coordinates for the bead are used by DL_MESO_DPD when setting up a new simulation without a *CONFIG* file.

The bonds, angles and dihedrals all follow similar formats:

bond/angle/dihedral key	a4	potential key
index 1 (i)	integer	first bead index in bond/angle/dihedral
index 2 (j)	integer	second bead index in bond/angle/dihedral
index 3 (k)	integer	third bead index in angle/dihedral (omitted for bonds)
index 4 (l)	integer	fourth bead index in dihedral (omitted for bonds/angles)
variables 1-4	real	potential parameters

where the bond/angle/dihedral key indicates the potential form used and the indices indicate which particles in the molecule (given in terms of the current molecule type) are involved in the bonded interaction. (Full details of which potentials are available and what is required for each type are given in Chapter 10 of the DL_MESO User Manual.)

The `external` directive is followed by another line with a keyword (`grav`, `elec` or `shear`) indicating the type of field to be applied and the Cartesian components of the field: the gravitational acceleration applied to all particles, the constant electric field applied to all charged particles or the velocity of one of the shearing walls when using Lees-Edwards boundary conditions.

This file is compulsory for a DL_MESO_DPD calculation and is used by the *scan_field()* and *read_field()* subroutines, the former scanning in information required to allocate arrays and the latter reading in all of the required interaction data. This file can be constructed by hand using a text editor, although use of the DL_MESO GUI is recommended when creating one for the first time. The `molecule-generate.cpp` utility is also recommended for constructing data for chain molecules with branches and needs to be compiled before launching it via the DL_MESO GUI.

11.3 CONFIG

This ANSI text input file enables users to provide their own initial configurations for DL_MESO_DPD simulations. It consists mainly of blocks of data providing the species, number and at least the position for each particle in the simulation, which are read into DL_MESO_DPD at the start of the calculation[8]. Multiple numbers on lines can be separated from each other with spaces, tabs or commas.

The first line of this file gives the simulation name (up to 80 characters), while the second line includes a file key *levcfg* indicating the data available per particle (0 = positions, 1 = positions and velocities, 2 = positions, velocities and forces), a periodic boundary key *imcon* (normally an integer greater than zero) and, optionally, the number of particles in the file and the current configuration energy. If the periodic boundary key is greater than zero, the following three lines need to include the Cartesian components for the x-, y- and z-axis vectors: since DL_MESO_DPD only operates with orthogonal boxes, only the diagonal of these vectors will be read and used.

Each particle is represented by a block record, with at least two lines of information:

- The species name (8 characters) and the global particle number (integer, optional)
- The x-, y- and z-coordinates for the particle (real)
- The x-, y- and z-components of the particle velocity (real, only included if *levcfg* is 1 or 2)
- The x-, y- and z-components of force on the particle (real, only included if *levcfg* is 2)

This file is entirely optional for a DL_MESO_DPD calculation, but if it is to be used, it must be in the directory where DL_MESO_DPD is launched. CONFIG files can be created from *export* or *HISTORY* files generated from previous DL_MESO_DPD using the *export_config.F90* or *history_config.F90* utilities respectively. Alternatively, DL_MESO_DPD can create a CFGINI file using the same format (that can later be renamed as CONFIG) based on the initial configuration it creates from scratch when the `l_init` directive is included in the *CONTROL* file, which is carried out by the *write_config()* subroutine.

The *scan_config()* and *read_config()* subroutines read the CONFIG file: the former looks for the system volume, while the latter reads in and assigns the particle data to the available processor cores based on their positions. The `nfold` option in the *CONTROL* file can be used to duplicate the unit cell in the CONFIG file and produce a larger simulation. Frozen particle walls or hard surfaces can be added to a system initialised using a CONFIG file, although users need to take care that no molecules cross these boundaries (no checks for this are currently carried out by DL_MESO_DPD).

11.4 export

This binary file consists of information required for DL_MESO_DPD to restart and extend a DPD simulation - the positions, velocities and forces for all particles - which can be read in if a `restart` directive is used in the *CONTROL* file.

The file consists of a header with the name of the DPD calculation (80 characters), two (4-byte) integers giving the total number of particles and the number of particles not included in molecules, and eight double-precision real numbers (8 bytes each) giving the specified temperature, the timestep size, the dimensions of the simulation box and the current displacement used for Lees-Edwards shearing boundaries.

This is then followed by a block of integers to indicate the ordering of particles in the file. Each particle supplies three integers of data: the global particle number, its species and its molecule type. A block of double-precision real numbers follows, with each particle providing nine values for its position, velocity and force (each property given as three Cartesian components). The particle data is provided in this block using the same ordering as the integer block preceding it. This format is particularly amenable to being written concurrently (using MPI-IO) by a small group of processor cores that have previously gathered together particle data: it avoids the need to sort the particles by global numbers and enables large streams of integers or real numbers to be written in single write operations.

This file is automatically generated by DL_MESO_DPD during a simulation by the *write_export()* subroutine and either at the very end once all the timesteps have been completed or when the calculation job time (less the close

[8] This file is formatted identically to CONFIG files used in DL_POLY [127][139], with the origin set at the centre of the simulation volume.

time) specified in *CONTROL* has run out. The endianness (big or small) of the file corresponds to that of the computer used to run the calculation: it can be used to restart the simulation on another machine provided the two computers have the same endianness or DL_MESO_DPD is compiled on the second computer to use that endianness (which can be set as a compiler flag). The *read_export()* subroutine gets all processor cores to read a section of the file each and subsequently send data for individual particles to the appropriate cores based on positions: it can be used either to resume a previous simulation from where it left off or to start a new simulation using the configuration given in the export file.

The *export_config.F90* utility can be used to generate a *CONFIG* file from this file as a starting point for a new simulation, while *export_image_vtf.F90* and *export_image_xml.F90* can be used to generate VTF and GALAMOST XML-format files that can be opened in VMD and OVITO respectively to visualise the system at the point the export file was created.

11.5 REVIVE

This binary file consists of information required for DL_MESO_DPD to resume a previous simulation from where it left off - barostat parameters, statistical properties and random number generator states - which can be read in if a `restart` directive is used in the *CONTROL* file.

The file consists a header with the name of the DPD calculation (80 characters), 9 double-precision real numbers (8 bytes each) with the barostat piston velocities (or volume scaling factor) for the current timestep (*upx*, *upy*, *upz*), the previous timestep (*up1x*, *up1y* and *up1z*) and the piston forces (*fpx*, *fpy* and *fpz*), and three (4-byte) integers with the current timestep number *nstep*, the number of values used for statistical averaging *nav* and the statistical stack size for rolling average values *nstk*.

This is then followed by the current instantaneous values (all double-precision real values) of the following properties:

- Total energy per particle
- Total potential energy per particle
- Electrostatic energy per particle
- Bond energy per particle
- Angle energy per particle
- Dihedral energy per particle
- Virial per particle
- Kinetic energy per particle
- System pressure
- System volume
- Interfacial tension along the z-axis
- System temperature
- Partial temperature in the x-direction
- Partial temperature in the y-direction
- Partial temperature in the z-direction
- Mean bond length
- Mean bond angle (in radians)
- Mean bond dihedral (in radians)

and then the current array with rolling average values, *rav*, for all of the above properties (apart from mean bond lengths, angles and dihedrals). The arrays used to calculate mean and fluctuating values of the same 15 properties plus the pressure tensors separated into conservative interaction, dissipative, random and kinetic parts, *ave* and *flc*, then follow.

Arrays for statistical stacks then follow: starting with the current sums of 11 properties, *zum*, used to calculate the rolling average values:

- Total potential energy per particle
- Electrostatic energy per particle
- Bond energy per particle
- Angle energy per particle
- Dihedral energy per particle
- Virial per particle
- System volume
- Interfacial tension along the z-axis
- Kinetic energy per particle in the x-direction
- Kinetic energy per particle in the y-direction
- Kinetic energy per particle in the z-direction

and followed by the current statistical stacks for these properties - *stkpe*, *stkee*, *stkbe*, *stkae*, *stkde*, *stkvir*, *stkvlm*, *stkzts*, *stktkex*, *stktkey* and *stktkez* - each of which have *nstk* values.

The *write_revive()* subroutine mainly uses processor core 0 to write the REVIVE file - all cores have the same statistical data in memory - apart from the random number generator states, which are each written concurrently to the file by each individual core using MPI-IO. The *read_revive()* subroutine - only called when a simulation is restarted - uses processor core 0 to read the statistical data and broadcast the values to all other cores, while each core (up to the number given in the REVIVE file) reads in its own random number generator state to overwrite the previously initialised state.

No utilities currently exist to read REVIVE files, as these are only intended to be used by DL_MESO_DPD itself for simulation restarts.

11.6 OUTPUT

This ANSI text file is generated by all DL_MESO_DPD calculations and contains:

- The number of processors available (and threads per core if using the OpenMP version) and their endianness.
- The system and bond/angle/dihedral properties used for calculations.
- Domain decomposition details (parallel version only).
- The starting positions and velocities for a sample of particles (in processor core 0).
- The calculation time, current values and rolling averages of system-wide properties every *nsbpo* time steps.
- Final averages and fluctuations (standard deviations) for all reported properties and pressure tensors over all time steps after equilibration.
- The final positions and velocities for a sample of particles (in processor core 0).
- Elapsed and average times for the calculation.

Only the properties relevant to the simulation will be printed: for instance, electrostatic and bond energies will be output if the simulation includes these kinds of interactions, while the partial temperatures will be output if the system is likely to include dynamics (e.g. if a constant force or shear is applied) or the `print partial` directive is included in the *CONTROL* file.

If the `l_scr` directive is included in the *CONTROL* file, the above simulation information will be redirected to the standard output for the machine and its operating system (e.g. to the screen) and no OUTPUT file will be generated. This directive may be useful when a simulation crashes but no error messages or other information are printed to the OUTPUT file, or to use a custom-named file for this output.

11.7 HISTORY

If the `trajectory` option is specified in the *CONTROL* file, DL_MESO_DPD will generate this binary file containing snapshots of the simulation every *ntraj* timesteps starting at timestep number *straj* as a trajectory. This file is written using stream I/O or MPI-IO without beginning or end of record markers typically created by Fortran programs.

The file starts with a header beginning with a (4-byte) integer (currently equal to 1) as an endianness check used by DL_MESO_DPD and its utilities, the latter of which use this value to determine if the HISTORY file needs to be read using the opposite endianness. This is followed by another integer indicating the number of bytes per floating point (real) number used in the file (set to 8 for double precision), a long integer (8 bytes in length) with the current size of the HISTORY file in bytes (the value of which can be checked when reading or during simulation restart) and two further standard integers with the current number of trajectory frames in the file and the timestep number for the last trajectory frame.

The next part of the header describes the simulation itself: an 80-character simulation name, integers providing the numbers of particle species, molecule definitions, particles *not* involved in molecules, all particles and bonds. An integer indicating the trajectory data key (equal to 0 for positions, 1 for positions and velocities, 2 for positions, velocities and forces) then follows with three further integers indicating the boundary condition types in the x-, y- and z-directions (0 = periodic, 1 = Lees-Edwards shearing, 2 = specular reflection, 3 = bounce back reflection).

Data for the particle species then follow, with the following provided for each species (as double-precision real numbers unless otherwise indicated):

- The species name (8 character string)
- Mass per particle of the species
- Interaction length or particle radius
- Charge or valency of a particle for the species
- Flag indicating if the species is frozen (1) or not (0), given as an integer

If there is at least one molecule type, the names for the molecule types are then supplied as 8 character strings.

Data about the individual particles and bonds then follow. The following is provided for each particle (all given as integers):

- Its global particle number (index)
- Its species type
- Its molecule type
- Its molecule number (used to identify individual molecules)

and any bonds are given as pairs of global particle numbers (all integers) that are connected together. It should be noted that neither the particle nor bond data need to be written in numerical order by global particle numbers. This brings the file header to a close: its size can be determined from the data supplied inside it (mainly the total numbers of species, molecule types, particles and bonds) and used to skip past it when reading the simulation snapshots.

Each trajectory frame in the HISTORY file consists of a block of information about the simulation at the given timestep (given as double-precision real numbers unless otherwise specified):

- Time of the trajectory frame in DPD units (product of timestep size Δt and timestep number less the number of equilibration timesteps)
- Total number of particles in trajectory frame (given as an integer)
- Length of simulation box in x-dimension

- Length of simulation box in y-dimension
- Length of simulation box in z-dimension
- Displacement in x-direction for Lees-Edwards shear
- Displacement in y-direction for Lees-Edwards shear
- Displacement in z-direction for Lees-Edwards shear

This information block is then followed by a series of integers with global particle numbers to indicate the order in which the particles and their data appear in this trajectory frame. A block of double-precision real numbers follows, with each particle providing three, six or nine values for the three Cartesian components of its position, velocity and force (depending on the trajectory data key selected in the *CONTROL* file). The particle data is provided in this block using the same ordering as the integer block preceding it. This format is particularly amenable to being written concurrently (using MPI-IO) by a small group of processor cores that have previously gathered together particle data: it avoids the need to sort the particles by global numbers and enables large streams of integers or real numbers to be written in single write operations.

This file is generated by DL_MESO_DPD during a simulation, with the *write_history_header()* subroutine creating the file and writing the header to it (if required) and the *write_history()* subroutine appending each trajectory frame and updating the filesize, number of frames and the timestep number for the last frame in the header. The endianness (big or small) of the file corresponds to that of the computer used to run the calculation: it is possible to get a different computer to continue writing to the HISTORY file provided it either uses the same endianness as the original machine or DL_MESO_DPD is compiled to use that endianness (available as a compiler flag). If a simulation is restarted, the *write_history_header()* subroutine will check the HISTORY file had previously been written correctly (producing warning messages if not) and prepare DL_MESO_DPD to append new trajectory frames to the correct places in the file.

The HISTORY file can be used with twelve utilities provided with DL_MESO:

- *traject_vtf.F90* and *traject_selected_vtf.F90* produce plottable VMD files [60] with sets of bonded particles represented as residues
- *traject_xml.F90* and *traject_selected_xml.F90* produce plottable XML files formatted for the coarse-grained molecular dynamics code GALAMOST [157], which can be read into the visualisation software package OVITO [132]
- *history_config.F90* can produce a *CONFIG* file from a selected frame in the HISTORY file
- *local.F90* can calculate localised properties (e.g. temperature, composition) and produce Legacy VTK files with these properties as cell data that can be visualised and analysed using Paraview
- *isosurfaces.F90* can produce isovolumetric maps based on a given species as Legacy VTK files and calculate order parameters to determine the structure of a mesophase
- *radius.F90* calculates the end-to-end distances and radii of gyration for all molecules, as well as give normalised distributions of end-to-end distances
- *dipole.F90* calculates the dipole moments for all molecules in the system, autocorrelation functions of dipole moments and their Fourier transforms
- *rdf.F90* and *rdfmol.F90* calculate radial distribution functions (RDFs) for pairs of particle species or molecule types respectively, with the option of calculating Fourier transforms of the RDFs to give structure factors
- *widom_insertion.F90* calculates excess chemical potentials of individual particles or molecules by random insertions into particle configurations supplied from trajectory frames

All of these utilities can be run on different machines to those used for DL_MESO_DPD calculations, including those with a different endianness.

11.8 CORREL

If specified by the `stats` directive in the *CONTROL* file, DL_MESO_DPD will generate this ANSI text file with statistical data written every *iscorr* timesteps after equilibration has finished. This file can be imported into a spreadsheet program or used by graph-plotting software for visualisation and analysis.

The file is formatted predominately as fixed-width columns of numbers - given as (scientific) E notation with 6 decimal places and three characters for the exponent and its sign - with column headers in the first line starting with a # character. The number of columns printed to the file will depend upon the simulation, primarily the interactions and ensemble in use: if a particular property in the table below is not required (e.g. bond energy for a system without molecules), it will be left out.

at a timestep interval specified by the user, 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):

Table 11.4: Available properties in CORREL output file

quantity	column header	property
t	`time`	time (in DPD units)
E_{tot}	`en-total`	total energy per particle
$E_{pot,tot}$	`pe-total`	total potential energy per particle
$E_{pot.elec}$	`ee-total`	total electrostatic energy per particle
$E_{pot,bond}$	`be-total`	total (stretching) bond energy per particle
$E_{pot,angle}$	`ae-total`	total bond angle energy per particle
$E_{pot,dihed}$	`de-total`	total bond dihedral energy per particle
P	`pressure`	system pressure
P_{xx}	`p_xx`	xx-component of pressure tensor (σ_{xx}/V)[9]
P_{xy}	`p_xy`	xy-component of pressure tensor (σ_{xy}/V)
P_{xz}	`p_xz`	xz-component of pressure tensor (σ_{xz}/V)
P_{yx}	`p_yx`	yx-component of pressure tensor (σ_{yx}/V)
P_{yy}	`p_yy`	yy-component of pressure tensor (σ_{yy}/V)
P_{yz}	`p_yz`	yz-component of pressure tensor (σ_{yz}/V)
P_{zx}	`p_zx`	zx-component of pressure tensor (σ_{zx}/V)
P_{zy}	`p_zy`	zy-component of pressure tensor (σ_{zy}/V)
P_{zz}	`p_zz`	zz-component of pressure tensor (σ_{zz}/V)
V	`volume`	system volume
L_x	`L_x`	box length in dimension x
L_y	`L_y`	box length in dimension y
L_z	`L_z`	box length in dimension z
γ_z	`tension`	interfacial tension along z-axis
T	`temperature`	system temperature
T_x	`temp-x`	partial temperature in x-dimension ($\langle\sum_i m_i v_{i,x}^2\rangle$)

continues on next page

Table 11.4 – continued from previous page

quantity	column header	property
T_y	`temp-y`	partial temperature in y-dimension $(\langle\sum_i m_i v_{i,y}^2\rangle)$
T_z	`temp-z`	partial temperature in z-dimension $(\langle\sum_i m_i v_{i,z}^2\rangle)$
$\langle r_{bond}\rangle$	`bndlen-av`	average (mean) bond length
$r_{bond,max}$	`bndlen-max`	maximum bond length
$r_{bond,min}$	`bndlen-min`	minimum bond length
$\langle\theta_{angle}\rangle$	`angle-av`	average (mean) bond angle (in degrees)
$\langle\phi_{dihed}\rangle$	`dihed-av`	average (mean) bond dihedral (in degrees)

This file is written by the *write_correl()* subroutine, which will ordinarily only create a new CORREL file (with the required column headers) if one does not already exist. If a CORREL file already exists, DL_MESO_DPD will append data to it even if running a new simulation: if this is not required, the user will need to either delete or rename the existing CORREL file before running DL_MESO_DPD.

11.9 Stress_*.d

If specified by the `stress` directive in the *CONTROL* file, DL_MESO_DPD will generate at least one ANSI text file containing stress tensors separated by contributions every *nstrs* timesteps starting from timestep number *sstrs*, both of which are specified by the user. Four contributions with associated filenames are available:

- Interaction potential (conservative, many-body, bonded and electrostatic) components: Stress_pot.d
- Dissipative components: Stress_diss.d
- Random components: Stress_rn.d
- Kinetic components: Stress_kin.d

and the user can select which files are written in the *CONTROL* file. The data are formatted in a similar form to the *CORREL* file: fixed-width columns of numbers in (scientific) E notation with 6 decimal places and three characters for the exponent and its sign, plus column headers in the first line starting with `#`. The available properties in these files are as follows:

Table 11.5: Available properties in Stress_*.d output files

quantity	column header	property
t	`time`	time (in DPD units)
P	`pressure`	system pressure
P_{xx}^Q	`p_xx`	xx-component of pressure tensor (σ_{xx}^Q/V) for contribution Q
P_{xy}^Q	`p_xy`	xy-component of pressure tensor (σ_{xy}^Q/V) for contribution Q
P_{xz}^Q	`p_xz`	xz-component of pressure tensor (σ_{xz}^Q/V) for contribution Q
P_{yx}^Q	`p_yx`	yx-component of pressure tensor (σ_{yx}^Q/V) for contribution Q
P_{yy}^Q	`p_yy`	yy-component of pressure tensor (σ_{yy}^Q/V) for contribution Q
P_{yz}^Q	`p_yz`	yz-component of pressure tensor (σ_{yz}^Q/V) for contribution Q
P_{zx}^Q	`p_zx`	zx-component of pressure tensor (σ_{zx}^Q/V) for contribution Q
P_{zy}^Q	`p_zy`	zy-component of pressure tensor (σ_{zy}^Q/V) for contribution Q
P_{zz}^Q	`p_zz`	zz-component of pressure tensor (σ_{zz}^Q/V) for contribution Q
V	`volume`	system volume

where Q represents potential, dissipative, random or kinetic contributions to the pressure tensor.

[9] The pressure tensors printed in this file include all contributions (interaction potential, dissipative, random and kinetic): the average between the xx-, yy- and zz-components should equal the system pressure, while the non-diagonal components are ordinarily symmetric, i.e. $P_(xy) = P_{yx}$.

These files can be imported into a spreadsheet program or used by graph-plotting software for visualisation and analysis. They can be used to analyse the rheological behaviour of the system, e.g. integrating autocorrelation functions of the potential and dissipative components of stress tensors to find the zero-shear viscosity [30].

CHAPTER

TWELVE

ADVICE ON DEVELOPING DL_MESO_DPD

DL_MESO_DPD has been written to allow users to either use the code as-is for their DPD simulations and/or to expand the code to implement their new functionalities and run simulations in a highly-scalable manner. In order to expand upon DL_MESO_DPD's feature set, this chapter provides some advice on what changes need to be made to the code and how these could be carried out.

12.1 New conservative interaction forces and potentials

The main force calculation routines - *forces_mdvv()*, *forces_dpdvv()* etc. - call the *conservativeforce()* subroutine to obtain the pairwise force and potential for a given particle pair[1]. The functional form is specified by the *ktype* array for each pair of particle species, whose index is numbered as:

$$k = \frac{\max(i,j)\left(\max(i,j)-1\right)}{2} + \min(i,j)$$

where i and j are the species numbers for the two particles. A `SELECT CASE` block is used on the value in this array for the given species pair to select the functional form:

1. Lennard-Jones [66]
2. Weeks-Chandler-Andersen [147]
3. *Standard DPD* Groot-Warren [46]
4. Two-term many-body DPD [142]

so another value can be selected for a new functional form and implemented as a new `CASE` in the *conservative-force()* subroutine. The outputs for this subroutine are:

- `gforce`: Scalar force divided by the distance between the particle pair, $\frac{F_{ij}^C}{r_{ij}}$
- `pot`: Potential energy for particle pair, U_{ij}

where the force is related to the potential by

$$F_{ij}^C = -\frac{\partial U_{ij}}{\partial r_{ij}}$$

and the division by the distance r_{ij} is used to enable multiplication of the scalar force by the unit vector while reducing the number of required division operations.

The parameters for the conservative force and potential are stored in the two-dimensional *vvv* array, with the first index giving the available parameters (or useful numbers derived from those parameters) and the second index giving the pair of particle species k. These parameters can include a cutoff distance for the potential and force, which can be compared with the distance between the particles (`rrr`) or its square (`rsq`).

Modifications can be made to reading the *FIELD* file to include the new interaction type. These are specified in the information block under the `interactions` keyword, and require a name for the interaction type (normally

[1] The *conservativeforce()* subroutine appears in both the standard and OpenMP versions of *field_module.F90*: the user-developer should ideally make the required changes in *both* versions of the subroutine to ensure they are implemented in any available form of DL_MESO_DPD.

up to 4 characters long but no longer than 16 characters) and up to 7 parameters including a lengthscale and a thermostat parameter (dissipative force parameter γ_{ij} or collision frequency Γ_{ij}).

The *scan_field()* subroutine searches for the available interaction types (under the `interactions` keyword) to find the maximum number of parameters per interaction type, *mxprm*, and the maximum cutoff distance if not specified in the *CONTROL* file from the lengthscale.

The *read_field()* subroutine actually reads in the parameters and assigns them (and any numbers based on them that are useful for force and potential calculations) to the *vvv* array, while the thermostat parameters are placed in the *gamma* array. If the potential does not truncate to zero at or before the maximum interaction cutoff distance *rcut*, long-range corrections to potential and virials can be added to the *clr* array. The `interact` array in this subroutine is used to indicate whether or not the species pair has its parameters - energy, length scale and thermostat - correctly specified, and is checked after reading the entire *FIELD* file to see if all particle species pairs have specified parameters and/or if those missing values can be derived using mixing rules.

The *sysdef()* subroutine reports the interaction types and parameters specified in the *FIELD* file for the various species pairs, using the *ktype* array to identify the type and which parameters need printing to the *OUTPUT* file. The energy parameters, interaction lengths and thermostat parameters are printed separately, with the energy parameters indicating the potential type for each species pair. The potential and virial long-range corrections can also be printed if any of the interaction types use them.

12.2 New wall forces and potentials

The main subroutine calculating wall potentials and forces, *wallforces()*, calls the *surfaceforce()* subroutine to obtain the force and potential for each particle within the surface cutoff distance *srfzcut* of the planar surface. The functional form is specified by the *srfktype* array for each particle species, and the value from this array for the required particle species is used in the `SELECT CASE` block to select the form used in calculations, currently one of two options:

1. *Standard DPD* Groot-Warren [46][105]
2. Weeks-Chandler-Andersen [147]

Another value can thus be selected for a new functional form and implemented as a new `CASE` in the *surfaceforce()* subroutine. The outputs for this subroutine are:

- `srfforce`: Scalar force divided by the distance between the particle and the wall, $\frac{F_{wall,i}}{z_{wall,i}}$
- `srfpot`: Wall potential energy for particle, $U_{wall,i}$

where the force is related to the potential by

$$F_{wall,i} = -\frac{\partial U_{wall,i}}{\partial z_{wall,i}}$$

and the division by the distance $z_{wall,i}$ is used to enable multiplication of the scalar force by the unit vector (which points orthogonally from the wall back into the system) while reducing the number of required division operations.

The parameters for the wall forces and potentials are stored in the two-dimensional *vvsrf* array, with the first index giving the available parameters (or useful numbers derived from those parameters) and the second index giving the particle species i. These parameters can include a cutoff distance for the potential and force, which can be compared with the distance between the particle and the wall (`rrr`) or its square (`rsq`).

Modifications can be made to reading the *FIELD* file to include the new wall interaction type. These are specified in the information block under the `surfaces` keyword, and require a name for the interaction type (normally up to 4 characters long but no longer than 16 characters) and 2 parameters including a lengthscale.

The *scan_field()* subroutine searches for the available wall interaction types (under the `surfaces` keyword) to find the maximum number of parameters per interaction type, *mxsprm*, and the maximum surface cutoff distance if not specified in the *CONTROL* file from the lengthscale.

The *read_field()* subroutine actually reads in the parameters and assigns them (and any numbers based on them that are useful for force and potential calculations) to the *vvsrf* array. Unlike the conservative interactions, the omission of any species in surface interactions will not cause issues: missing particle species will effectively be omitted from wall force and potential calculations.

The *sysdef()* subroutine reports the wall interaction types and parameters specified in the *FIELD* file for the various species, using the *srfktype* array to identify the type and which parameters need printing to the *OUTPUT* file. The energy parameters and interaction lengths are printed separately, with the energy parameters indicating the potential type for each species.

12.3 New bond, angle and dihedral forces and potentials

The main subroutines calculating the bond interaction forces and potentials, *bondforceslocal()* and *bondforces-global()*, call the *bond_force()*, *angle_force()* and *dihedral_force()* subroutines to calculate the forces and potentials for a given bond, angle or dihedral (respectively)[2]. The functional form is given as an integer input to each subroutine - `bondtype`, `angtype` or `dhdtype` - with a `SELECT CASE` block used to select the form and carry out the calculations: these are stored for the available bonds/angles/dihedrals in the system in the *bdtype*, *angtype* and *dhdtype* arrays. The parameters for each bond/angle/dihedral are also input into the subroutines and stored in arrays for each available type. Up to four parameters can be specified for each bond/angle/dihedral type, with their specific meanings chosen arbitrarily when implementing the potentials and forces.

The force supplied as an output by *bond_force()* is the scalar bond force divided by the distance between the two particles, $\frac{F_{ij}}{r_{ij}}$. A maximum possible bond length `mxlen` is also supplied as an output for certain bond types to check that the current bond is within that distance. The force supplied by *angle_force()* is the scalar angle force divided by the sine of the angle formed by the two connected bonds, $\frac{F_{ijk}}{\sin\theta_{ijk}}$. This subroutine can also output the resulting virial for the angle `virial` and any additional force contributions (`dfab`, `dfcb`) if the angle interaction potential includes screening/truncation functions (currently not used in DL_MESO_DPD). The force supplied by *dihedral_force()* is the scalar dihedral force divided by the sine of the dihedral angle, $\frac{F_{ijkl}}{\sin\phi_{ijkl}}$. Since dihedrals do not contribute to virials, no such output is made from this subroutine (although the forces do contribute to the stress tensor).

Modifications can be made to reading the *FIELD* file to include the new bond/angle/dihedral type. These are specified in the information block under the `molecules` keyword within the sub-blocks indicated by `bonds`, `angles` or `dihedrals`. A name for the new bond/angle/dihedral type is required (normally up to 4 characters long but no longer than 16 characters) and up to 4 parameters can be specified.

The *scan_field()* subroutine searches for the required bond, angle and dihedral types for all molecule definitions: each of these is identified by the keyword for the type and the parameters, ending up with all of the unique bond, angle and dihedral definitions and eventually assigning the types to *bdtype*, *angtype* and *dhdtype*, and the parameters to *aabond*, *bbbond* etc.

The *read_field()* subroutine reads through the *FIELD* file again and identifies the bond/angle/dihedral types for each bond/angle/dihedral in each molecule's definition based on the keyword and the supplied parameters. The unique bond/angle/dihedral types are assigned to the arrays subsequenrly used to assign data to the book-keeping arrays: *bdinp3*, *anginp4* and *dhdinp5*.

The *sysdef()* subroutine reports the bonded interaction types and parameters specified in the *FIELD* file, using the *bdtype*, *angtype* and *dhdtype* arrays to identify the types. Each bond/angle/dihedral type is printed with all four parameters, regardless of whether or not all of them are used during calculations.

[2] The *bond_force()*, *angle_force()* and *dihedral_force()* subroutines appear in both the standard and OpenMP versions of *bond_module.F90*: the user-developer should ideally make the required changes in *both* versions of the subroutines to ensure they are implemented in any available form of DL_MESO_DPD.

12.4 Change to many-body DPD calculations

Most of the code changes required for many-body DPD interactions are carried out in the same manner as for other conservative interactions (see above). There are, however, a few additional considerations to take when applying a new many-body DPD model in DL_MESO_DPD, mainly involving changes to the *manybody_module.F90*[3].

The *local_density()* subroutine calculates localised densities for each particle species (as stored in *rhomb*): the sum over all particle species (the second index of this two-dimensional array) gives the total localised density for each particle, as used in the current two-parameter many-body DPD model. These localised densities use the screening function w^{ρ} given in *weight_rho()*, which can be changed by the user-developer provided:

- It truncates to zero at the many-body cutoff distance *rmbcut*
- The weighting function used for the pairwise forces is proportional to the derivative of this function with respect to distance between particle pairs

The self-energy terms for the potential are dependent only on localised densities and are calculated in the *many-body_potential()* subroutine for all particles and all species pairs involving at least one non-frozen particle species. The calculations in this subroutine should be modified for new many-body DPD interactions.

No changes are absolutely necessary in the *scan_field()* or *read_field()* subroutines, unless the user-developer wishes to give a new name to the interaction type. The instructions for doing this are nearly identical to those for other conservative interactions (see above), except that *lmb* should be set to true when reading the interaction in *read_field()* to ensure localised densities and self-energy terms are calculated.

12.5 Changes to electrostatic interaction calculations

Two main changes may be made to calculations of electrostatic interactions between charged particles in DL_MESO_DPD:

- Addition of a new charge smearing scheme
- Alternative method to calculate long-range (reciprocal-space) part of Ewald sum

The first change - adding a new charge smearing scheme - involves creating a new subroutine in *ewald_module.F90* to calculate pairwise real-space calculations[4], which is the only place where the scheme needs to be applied. The user-developer should copy and paste one of the existing subroutines, e.g. *ewald_real_slater_exact()*, and modify it accordingly. The variable *betaew* is available as a parameter related to the charge smearing length *chglen*. Changes will additionally need to be made to the *read_control()* subroutine for the `smear` keyword to read the new charge smearing type as the integer variable *etype* and to calculate the value of *betaew* from the charge smearing length, the *sysdef()* subroutine to print information about electrostatic interactions to the *OUTPUT* file, and the subroutines in *field_module.F90* setting up parallel link cells and calculating forces (e.g. *plcfor_mdvv()*) to include the new Ewald real-space subroutine as an option. It should be noted that the variable *etype* selects both the charge smearing scheme and the implementation of the reciprocal-space calculations (standard Ewald or SPME), so changes to numbering for this variable will be needed for the new charge smearing scheme to fit in with the current ones.

The addition of a new method to calculate the long-range part of electrostatic interactions - currently available with standard Ewald summation and Smooth Particle Mesh Ewald (SPME) - will require even more extensive changes. Specifically, the user-developer may need to create their own module (along similar lines to *spme_module.F90*) to incorporate an implementation of this new method. Similarly to adding a charge smearing scheme, changes will be needed for *CONTROL* file reading to read in the new calculation scheme (as an alternative to `ewald` or `spme`) and assign the variable *etype* to apply it for the available charge smearing schemes, to write information to the *OUTPUT* file and to select new subroutines during force calculations.

[3] This module exists in both standard and OpenMP versions: the user-developer should ideally make the required changes in *both* versions of the module to ensure they are implemented in any available form of DL_MESO_DPD.

[4] This module exists in both standard and OpenMP versions: the user-developer should ideally make the required changes in *both* versions of the module to ensure they are implemented in any available form of DL_MESO_DPD.

12.6 New integration scheme or thermostat

Each thermostat and force integration scheme is implemented by its own module:

- *integrate_dpd_mdvv.F90* for DPD using standard (MD) Velocity Verlet
- *integrate_dpd_dpdvv.F90* for DPD using DPD Velocity Verlet (recalculation of dissipative forces)
- *integrate_dpd_shardlow.F90* for DPD with Shardlow splitting, using Velocity Verlet for interaction forces
- *integrate_lowe.F90* for Lowe-Andersen using Velocity Verlet for interaction forces
- *integrate_peters.F90* for Peters using Velocity Verlet for interaction forces
- *integrate_stoyanov.F90* for Stoyanov-Groot using Velocity Verlet for interaction and pairwise thermostat forces

with subroutines for each ensemble and barostat (where applicable). The user-developer is advised to create their own module to implement their own thermostat and/or force integration scheme along the lines of the existing ones listed above. Subroutine calls for applying the communications - *deportdata()* and *exportvelocitydata()* (where applicable) - applying boundary conditions. recalculating Ewald/SPME reciprocal space vector maps and corrective forces for frozen charged particles etc. can be based on the pre-existing subroutines for the various ensembles.

12.7 New or modified output file format

The simplest type of new output file to create is one that reports on system-wide values of a property. Noting that obtaining these values in parallel running might require a global communication step, e.g. summation over all processor cores using e.g. *global_sum_dble()*, the results can be printed to a file by a single processor core. By convention and to ensure compatibility for serial running, this core should be numbered 0. An example of this approach includes *write_correl()*, which writes statistical data collected together by the *statis()* subroutine.

If data for individual particles need to be written to a new file, a similar approach to that used for *export* and *HISTORY* files can be taken. This involves gathering together data among groups of processor cores - as in *gather_write_data()* - before the writing cores in each group writes the data concurrently to the file (e.g. *write_history()*). Since the cores in each group and the writing cores are likely to send/write different amounts of data, the cores in each gathering and writing group should share a measure of the amount of data they are sending or writing (e.g. the number of particles), which will allow these cores to specify where the data should end up in the receive array (i.e. the starting index) or the file (the starting byte number where the data should be placed). No changes need to be made to the I/O groups set up in *init_output_groups()* for the *HISTORY*, *export* and *CFGINI* files, although an additional MPI communicator should be created for the new file to enable it to be written (and a file handle for MPI-IO to be created) without interrupting writing to other existing files.

12.8 Modifications to input file reading

If the new feature has been tested and the user-developer wishes to implement it more fully into DL_MESO_DPD, they will need to make modifications to reading input files - *CONTROL* for simulation parameters, *FIELD* for new interactions - to read in options and parameters for the new feature. The main subroutines to modify are *read_control()* and *scan_control()*, which read keywords and values in individual lines, and *scan_control()* and *scan_field()* to prepare any arrays for reading in data from these files.

Once the user-developer has devised the new keywords for the *CONTROL* or *FIELD* file, they can then add them to the main `DO WHILE` loops going through each line of the file with `IF` statements to check for the strings. The *getword()* function can be used to obtain a word from a given line that can be compared with a string (with the *lowercase()* subroutine available to change all uppercase letters to lowercase ones), while *getdble()* and *getint()* can read a double-precision real number and an integer from the same 'word' respectively.

When comparing strings with possible values, care should be taken to ensure enough characters in the string being compared are used. If the proposed keyword is long, only the first few characters may be compared to allow for variant spellings or abbreviations.

CHAPTER

THIRTEEN

DL_MESO_DPD ERROR MESSAGES

This chapter 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.

13.1 Messages related to input files

13.1.1 Message 1: cutoff radius value not set

A valid cutoff radius (either $r_{c,max}$ or r_c) for all (non-electrostatic) 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 either the `cutoff` directive or the `thermostat cutoff` directive is included with a non-zero value.

13.1.2 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 `temperature` directive is included with a non-zero value.

13.1.3 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.

13.1.4 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.

13.1.5 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.

Action: This error is unlikely to happen as the maximum number of particles per node 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.

13.1.6 Message 6: system is not charge neutral

The overall charge for the system is non-zero.

Action: None immediately required to ensure the simulation runs as this is a warning message, but the user may wish to adjust the numbers of charged particles to balance out positive and negative charges. (Ewald sums do not tend to work for periodic systems with overall charges.)

13.1.7 Message 7: at least one interaction length larger than cutoff radius

The interaction length for at least one interacting pair of species ($r_{c,ij}$) specified in the `FIELD` file exceeds the (global) cutoff radius (r_c) specified in the `CONTROL` file: any overly-long interactions will be truncated at the global cutoff radius.

Action: None required to ensure the simulation runs as this is a warning message, but the user may wish to increase the specified cutoff radius to prevent truncation of interactions at larger separations.

13.1.8 Message 8: boundary halo size larger than subdomain size - cannot apply SPME

The size of the boundary halo (either specified by the user or determined from required interaction and bond lengths) exceeds the length of at least one dimension of the subdomain volume assigned to each processor. This prevents correct charge grid assignment for Smooth Particle Mesh Ewald calculations when using distributed charge grids.

Action: The user should either increase the size of the maximum reciprocal vector or otherwise reduce the size of the boundary halo.

13.1.9 Message 9: volume not specified in CONTROL or CONFIG file

The system volume cannot be found in either the CONTROL file (for systems initialized from scratch) or the supplied CONFIG file.

Action: The user should check that either the CONTROL file includes a `volume` directive or the CONFIG file includes lines specifying the system volume.

13.1.10 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.

13.1.11 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.

13.1.12 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.

13.1.13 Message 22: no charge smearing length defined in CONTROL file

The supplied `CONTROL` file has specified a charge smearing type but no charge smearing length (or reciprocal).

Action: Check the `CONTROL` file includes a `smear length` or `smear beta` directive with a non-zero value. If the `CONTROL` file was originally created for earlier versions of DL_MESO_DPD, it will need modification to specify both charge smearing type and lengthscale in separate lines: see Appendix A in the DL_MESO User Manual for further details.

13.1.14 Message 23: no pressure specified in CONTROL file for barostat

The supplied `CONTROL` file has specified an ensemble involving a barostat (constant pressure, surface area or surface tension) but no pressure.

Action: Check the `CONTROL` file includes a `pressure` directive.

13.1.15 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.

13.1.16 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.

13.1.17 Message 32: unrecognised bond type defined in FIELD file

A bond type not included in the available range 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.

13.1.18 Message 33: unrecognised bond angle type defined in FIELD file

A bond angle type not included in the available range 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.

13.1.19 Message 34: unrecognised bond dihedral type defined in FIELD file

A bond dihedral type not included in the available range 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.

13.1.20 Message 35: name for species number i in FIELD file truncated to 8 characters

The given name for the i-th species in the `FIELD` file exceeds 8 characters and has had to be truncated.

Action: None required to ensure the simulation runs as this is a warning message, but the user may wish to check the `FIELD` file to ensure this species cannot be confused with another.

13.1.21 Message 36: name for molecule number i in FIELD file truncated to 8 characters

The given name for the i-th molecule type in the `FIELD` file exceeds 8 characters and has had to be truncated.

Action: None required to ensure the simulation runs as this is a warning message, but the user may wish to check the `FIELD` file to ensure this molecule type cannot be confused with another.

13.1.22 Message 37: missing finish directive in FIELD file for molecule i

No `finish` directive can be found for the i-th molecule type in the `FIELD` file.

Action: Check the `FIELD` file, particularly the i-molecule type, and ensure each molecule type ends with a `finish` directive.

13.1.23 Message 38: 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.

13.1.24 Message 39: 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.

13.1.25 Message 40: out-of-range bead number for bond in FIELD file for molecule i

At least one of the bond definitions for the i-th molecule type refers to a bead number out of range (either below 1 or above the number of beads for the molecule).

Action: Check the `FIELD` file, particularly the i-th molecule type, and check that the bond definitions refer to bead numbers within range.

13.1.26 Message 41: 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.

13.1.27 Message 42: out-of-range bead number for angle in FIELD file for molecule i

At least one of the angle definitions for the i-th molecule type refers to a bead number out of range (either below 1 or above the number of beads for the molecule).

Action: Check the `FIELD` file, particularly the i-th molecule type, and check that the angle definitions refer to bead numbers within range.

13.1.28 Message 43: 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.

13.1.29 Message 44: out-of-range bead number for dihedral in FIELD file for molecule i

At least one of the dihedral definitions for the i-th molecule type refers to a bead number out of range (either below 1 or above the number of beads for the molecule).

Action: Check the `FIELD` file, particularly the i-th molecule type, and check that the dihedral definitions refer to bead numbers within range.

13.1.30 Message 45: 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.

13.1.31 Message 46: 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.

13.1.32 Message 47: 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.

13.1.33 Message 48: 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.

13.1.34 Message 49: 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.

13.1.35 Message 50: zero reciprocal vector range for ewald sum

The maximum reciprocal vector, $\vec{k}_{max}$, has not been defined for systems requiring Ewald sum or SPME electrostatics.

Action: Look in the `CONTROL` file and make sure the `ewald` or `spme` directive includes the convergence parameter α and the extents of the maximum reciprocal vector, k_1, k_2 and k_3.

13.1.36 Message 51: cannot read restart file named export

No valid file called `export` can be found or read to restart a simulation.

Action: Check for the existence of the `export` file in the working directory; if no file exists or is needed, remove the **restart** directive from the `CONTROL` file.

13.1.37 Message 52: too many beads needed per node for export file

The total number of particles from the `export` file for a particular process exceeds the predicted maximum number of particles per node.

Action: This error is unlikely to happen as the maximum number of particles per node 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.

13.1.38 Message 53: cannot read REVIVE file - no previous statistical data available

No valid `REVIVE` file with statistical accumulators, barostat properties and random number generator states can be found or read.

Action: No immediate action is necessary to get DL_MESO to run as this message is a warning, but the user may need to consider either ensuring there is a `REVIVE` file available in the working directory or using the `restart noscale` directive in the `CONTROL` file (which starts a new simulation based on the configuration given in the `export` file).

13.1.39 Message 54: at least one molecule type incorrectly defined in FIELD file

The `FIELD` file contains a molecule definition that is incorrectly formatted, preventing other definitions from being read.

Action: Check the molecule definitions in the `FIELD` file to ensure each of them includes the number of molecules to be used in the simulation, the number of beads per molecule and finishes with the `finish` directive (which indicates the end of the molecule definition).

13.1.40 Message 55: insufficient number of beads per node allocated for required CONFIG file

The total number of particles from the `CONFIG` file for a particular process exceeds the predicted maximum number of particles per node.

Action: This error is unlikely to happen as the maximum number of particles per node 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.

13.1.41 Message 56: non-existent species given in CONFIG file

No valid species can be found for an entry in the `CONFIG` file: this can happen either when the file is being scanned to find the start of a particle entry or when the particles are being read (in the latter case, additional messages will be produced stating which particles have invalid species identifiers).

Action: Check the species definitions in the `FIELD` file and the i-th particle in the `CONFIG` file (if given as an additional message) to ensure that species is defined.

13.1.42 Message 57: out-of-range particle index given in CONFIG file

At least one of the particle indices given in the `CONFIG` file is out of range compared with the contents of the `FIELD` file.

Action: Check the particle numbering in the `CONFIG` file and the contents of the `FIELD` file, and adjust where necessary. If the contents of the `CONFIG` file match up with the `FIELD` file, consider using the `no index` option in the `CONTROL` file to ignore the supplied particle numbering.

13.1.43 Message 58: mismatch in species particle counts between FIELD and CONFIG files

The numbers of particles for a given species in the `CONFIG` file does not match those given in the `FIELD` file: a table itemising the total numbers for each species and for all species in both files is given before this message.

Action: Check the `CONFIG` and `FIELD` files to ensure the species compositions of both files match up.

13.1.44 Message 59: out-of-range particle index given in export file

At least one of the particle indices given in the `export` file is out of range compared with the contents of the `FIELD` file.

Action: Check the contents of the `FIELD` file to make sure it matches up with that used when the `export` file was created, and adjust where necessary. The `export_config` utility might be useful here to determine the contents of the `export` file.

13.1.45 Message 60: out-of-range species type given in export file

At least one of the species types given in the `export` file is out of range compared with the contents of the `FIELD` file.

Action: Check the contents of the `FIELD` file to make sure it matches up with that used when the `export` file was created, and adjust where necessary. The `export_config` utility might be useful here to determine the contents of the `export` file.

13.1.46 Message 61: out-of-range molecule type given in export file

At least one of the molecule types given in the `export` file is out of range compared with the contents of the `FIELD` file.

Action: Check the contents of the `FIELD` file to make sure it matches up with that used when the `export` file was created, and adjust where necessary. The `export_config` utility might be useful here to determine the contents of the `export` file, although the resulting `CONFIG` file would not directly show which particles belong to particular molecules.

13.1.47 Message 62: mismatch in species particle counts between FIELD and export files

The numbers of particles for a given species in the `export` file does not match those given in the `FIELD` file (with any cell duplication from the previous simulation taken into account): a table itemising the total numbers for each species and for all species in both files is given before this message.

Action: Check the `export` and `FIELD` files to ensure the species compositions of both files match up. If the `nfold` option was used for the simulation that created the `export` file, make sure the relevant values are specified in the `CONTROL` file.

13.1.48 Message 65: insufficient number of beads per node allocated for required initialization

The maximum number of particles per node 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 the maximum number of particles per node 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.

13.1.49 Message 66: discrepancy 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 new simulations without `CONFIG` files or restarted simulations without frozen bead walls, 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. If restarting a simulation that used frozen bead walls, remove the directives for frozen bead walls in the `CONTROL` and `FIELD` files and include the number of frozen beads used in the species totals in the `FIELD` file.

13.1.50 Message 67: molecule i bigger than domain - cannot insert into system

The maximum extent of molecule i, which is represented as a cuboid, is larger than the defined size of the system in all three dimensions. This message will only appear for systems with hard surfaces or frozen walls, and the molecule cannot therefore be inserted into the system without crossing these surfaces or walls.

Action: Either the system size must be increased to accommodate the defined molecule or the molecule needs to be made smaller to fit inside the system dimensions (which are reduced by the specified positions of hard surfaces or the thicknesses of frozen walls).

13.1.51 Message 68: molecule i bigger than domain in at least one dimension

The maximum extent of molecule i, which is represented as a cuboid, is larger than the defined size of the system in at least one dimension.

Action: No immediate action is necessary as this is a warning message, but the user may wish to either increase the system size or reduce the molecule size in future simulations. In the case of systems with hard surfaces or frozen walls, the molecule size may restrict how it can be randomly inserted into the system volume and it may take longer to initialize such systems.

13.2 Messages for processor-to-processor communication

13.2.1 Message 71: 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 the maximum number of particles per node and thus accommodate larger numbers of particles.

13.2.2 Message 72: 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 the maximum number of particles per node and thus accommodate larger numbers of particles.

13.2.3 Message 81: 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 the maximum number of particles per node and thus accommodate larger numbers of particles.

13.2.4 Message 82: 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 the maximum number of particles per node and thus accommodate larger numbers of particles.

13.2.5 Message 91: 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 the maximum number of particles per node and thus accommodate larger numbers of particles.

13.2.6 Message 92: 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 the maximum number of particles per node and thus accommodate larger numbers of particles.

13.2.7 Message 93: 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. (This error message can only be invoked when running the serial version of DL_MESO.)

Action: This error should never occur! If it does, please contact the authors of DL_MESO.

13.2.8 Message 94: cannot correctly export data to boundary halos for density calculations

Particle data for calculating localised densities (needed for many-body DPD) cannot be exported to the boundary halos as the number of particles created would be greater than that accounted for in memory. (This error message can only be invoked when running the serial version of DL_MESO.)

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 the maximum number of particles per node and thus accommodate larger numbers of particles.

13.2.9 Message 95: cannot correctly export data to boundary halos for density calculations with shear

Particle data for calculating localised densities (needed for many-body DPD) cannot be exported to the boundary halos with Lees-Edwards shearing as the number of particles created would be greater than that accounted for in memory. (This error message can only be invoked when running the serial version of DL_MESO.)

Action: This error message suggests non-constant particle densities across the system. The user may wish to use the `densvar` directive in the `CONTROL` file to increase the value of the maximum number of particles per node and thus accommodate larger numbers of particles.

13.3 Messages for runtime issues

13.3.1 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.

13.3.2 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.

13.3.3 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.

Action: The user may wish to use the `densvar` directive in the `CONTROL` file to increase the maximum numbers of particles per node and pairs per node, thus accommodating larger numbers of interacting pairs.

13.4 Messages related to output files

13.4.1 Message 301: incorrect endianness in HISTORY file - cannot append additional data

The endianness used for a pre-existing HISTORY file does not match the endianness currently used for the calculation: data appended to the end of the file will therefore not be readable.

Action: The endianness used in DL_MESO can be swapped by re-compiling with appropriate compiler flags in the Makefile.

13.4.2 Message 302: incorrect real number size in HISTORY file - cannot append additional data

The number of bytes per real number used in the HISTORY does not match the number of bytes per real number used by DL_MESO: there is a mismatch in real number types.

Action: The kind of real number used in DL_MESO can be modified by changing the definition of dp in the constants module.

13.4.3 Message 303: data corruption in HISTORY file

The beginning of a trajectory frame (the header containing the number of particles and time) in the HISTORY file cannot be read due to a previous interruption in writing it.

Action: Greater care must be taken to ensure there is enough closing time to complete writing the last trajectory frame to the HISTORY file at the end of a simulation.

13.4.4 Message 304: HISTORY file missing i frames of trajectories

The HISTORY file ends before the timestep at which the simulation is being restarted.

Action: None immediately required as this message is a warning and does not prevent the simulation from being restarted, but the user should be aware there will be a gap in the sequence of trajectories in the HISTORY file when it is visualised or analysed with post-processing utilities.

13.5 Messages related to memory usage

13.5.1 Messages 1001-1256: allocation/deallocation errors

Allocation or deallocation of arrays for DPD calculations (including reading of input data, transfer buffers for communications between processors, global arrays 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.

Action: 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 to hold particle data), under-populating multicore processors (i.e. using fewer cores per processor than the maximum available) or upgrading your machine. Alternatively, try running a smaller simulation.

CHAPTER

FOURTEEN

DL_MESO GUI CODE DESCRIPTION

This chapter lists and describes the classes, subroutines, functions etc. in the DL_MESO GUI, based on output generated using Doxygen with annotations in the code. Many of the classes describing windows (e.g. Define LBE System, *setlbeSys.java*) have event counterparts (e.g. *setlbeSysEvt.java*) that deal with user-driven actions, including changing the states of pull-down (combo) boxes and reading values from text boxes and checkboxes when buttons are pressed: these latter classes are also used to deal with related pop-up windows.

14.1 dlmesogui.java

Main window for DL_MESO GUI.

Sets up main window for DL_MESO GUI, including welcome message in centre and buttons at top for LBE, DPD and SPH simulations, Manual, Help and Exit. (Note that SPH button currently not a working option due to lack of Smooth Particle Hydrodynamics code in DL_MESO, but is available for future versions.)

14.1.1 Classes

- `public class dlmesogui`

14.1.2 Function/Subroutine Documentation

dlmesogui()

```
public dlmesogui ( )
```

Sets up main window for DL_MESO GUI, laying out elements onto a grid.

main()

```
public static void main (String[] arguments)
```

Creates new instance of graphical user interface when program is run.

14.2 dlmesoguiEvt.java

Events for main window of DL_MESO GUI.

Implements actions for LBE, DPD, SPH, Manual, Help and Exit buttons, including adding side bar on left hand side for LBE and DPD simulations.

14.2.1 Classes

- `class dlmesoguiEvt`

14.2.2 Function/Subroutine Documentation

dlmesoguiEvt

```
public dlmesoguiEvt(dlmesogui in)
public dlmesoguiEvt(dllbe lbein)
public dlmesoguiEvt(dldpd dpdin)
```

Sets currently open DL_MESO GUI (including side bars for LBE and DPD) as window to check for user actions and implement actions in response.

Parameters

in	dlmesogui	Instance of DL_MESO GUI
in	dllbe	Instance of LBE side bar
in	dldpd	Instance of DPD side bar

changecode

```
void changecode()
```

Opens window for Change LBE Code.

compilel

```
void compilel()
```

Opens window for Compile LBE Code.

runprogram

```
void runprogram()
```

Opens window for Run LBE Program.

gatherdata

```
void gatherdata()
```

Opens window for Gather LBE Data.

plotresult

```
void plotresult()
```

Opens window for Plot LBE Results.

setspa

```
void setspa(int dim)
```

Opens window for Set LBE Space.

Parameters

dim	int	Number of spatial dimensions required for LBE simulation (obtained from Define LBE System)

defsy

```
void defsy()
```

Opens window for Define LBE System.

dpdchangecode

```
void dpdchangecode()
```

Opens window for Change DPD Code.

dpdcompile

```
void dpdcompile()
```

Opens window for Compile DPD Code.

dpdrunprogram

```
void dpdrunprogram()
```

Opens window for Run DPD Program.

dpdgatherdata

```
void dpdgatherdata()
```

Opens window for Process DPD Data.

dpdplotresult

```
void dpdplotresult()
```

Opens window for Plot DPD Results.

dpdsetinteract

```
void dpdsetinteract()
```

Opens window for Set DPD Interactions.

dpddefsys

```
void dpddefsys()
```

Opens window for Define DPD System.

closelbe

```
void closelbe()
```

Closes current window for LBE simulations.

closedpd

```
void closedpd()
```

Closes current window for DPD simulations.

lbe

```
void lbe()
```

Opens side bar for LBE simulations on left-hand side of GUI window.

dpd

```
void dpd()
```

Opens side bar for DPD simulations on left-hand side of GUI window.

sph

```
void sph()
```

Opens side bar for SPH simulations on left-hand side of GUI window: currently not an active option, so will open pop-up window with error message.

help

```
void help()
```

Opens a pop-up window advising the user to visit the DL_MESO website.

manual

```
void manual()
```

Attempts to open the DL_MESO User Manual using Acrobat Reader (if installed on computer): if not available, opens a pop-up window to advise user to open file manually.

lbeUpdatePan0

```
void lbeUpdatePan0()
```

Updates side bar (panel) with buttons for LBE simulation without modifying the main window in the GUI.

lbeUpdatePan

```
void lbeUpdatePan(Component cc)
```

Updates side bar (panel) with buttons for LBE simulation and updates main window in the GUI with the required window (taken as an input).

Parameters

cc	Component	Window to be opened in main window of GUI

dpdUpdatePan0

```
void dpdUpdatePan0()
```

Updates side bar (panel) with buttons for DPD simulation without modifying the main window in the GUI.

dpdUpdatePan

```
void dpdUpdatePan(Component cc)
```

Updates side bar (panel) with buttons for DPD simulation and updates main window in the GUI with the required window (taken as an input).

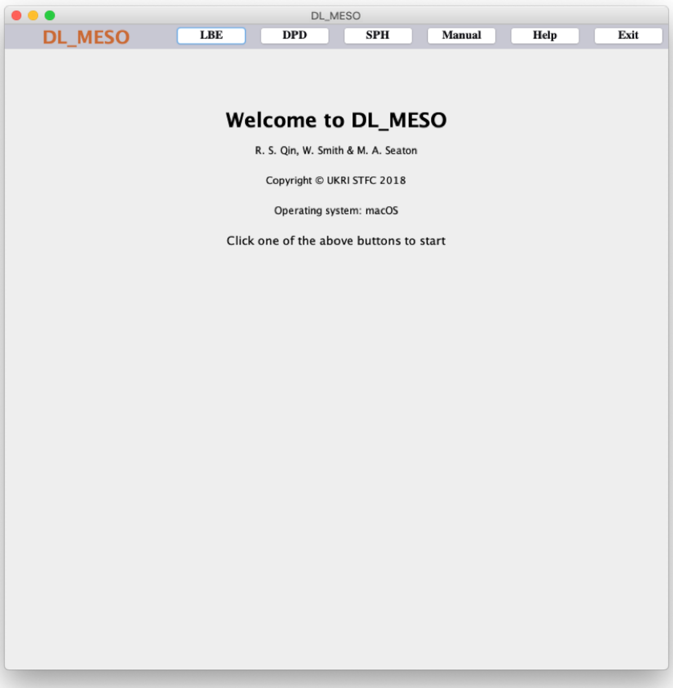
Parameters

cc	Component	Window to be opened in main window of GUI

14.3 dlwelcome.java

Welome message for DL_MESO GUI.

Displays welcome message for DL_MESO GUI, including an indicator of the identified operating system, in the centre of the main window.



Welcome to DL_MESO

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

Copyright © UKRI STFC 2018

Operating system: macOS

Click one of the above buttons to start

14.3.1 Classes

- `class dlwelcome`

14.3.2 Function/Subroutine Documentation

dlwelcome

```
public dlwelcome()
```

Sets up welcome message in middle of DL_MESO GUI window.

14.4 dllbe.java

Side panel for LBE.

Sets up panel on left hand side of main window for LBE (Lattice Boltzmann Equation) simulations, with buttons for Define LBE System, Set LBE Space, Change LBE Code, Compile LBE Code, Run LBE Program, Gather LBE Data, Plot LBE Results and Close LBE Panel.

14.4.1 Classes

- `class dllbe`

14.4.2 Function/Subroutine Documentation

dllbe

```
public dllbe()
```

Sets up LBE side panel for DL_MESO GUI, laying out buttons in a single column.

14.5 lbesysdim.java

System boundaries for LBE simulations.

Sets grid size, numbers of dimensions, fluids, solutes and temperature fields, boundary condition types and gradient order (using 0 for all values as default apart from 1 for gradient order), and provides arrays for boundary condition values at main boundaries around the outside of the grid (constant fluid densities, constant velocities, oscillating velocities, oscillation frequencies/periods, switches for boundary oscillations, constant concentrations, constant temperatures, constant heating rates): used to store global properties required for both Define LBE System and Set LBE Space windows.

14.5.1 Classes

- `class lbesysdim`

14.5.2 Variables

- `static int lbnx`

  Number of grid points in x-dimension
- `static int lbny`

  Number of grid points in x-dimension
- `static int lbnz`

  Number of grid points in z-dimension
- `static int led`

  Number of spatial dimensions
- `static int lbf`

  Number of fluids
- `static int lbc`

  Number of solutes
- `static int lbt`

  Number of temperature fields
- `static int bctyp`

  Boundary condition type for fluids
- `static int sbctyp`

  Boundary condition type for solutes
- `static int tbctyp`

  Boundary condition type for temperature
- `static int gradord`

  Order of gradient calculations at surfaces
- `static double densf[]`

  Fixed fluid densities at main boundaries on outside of simulation box
- `static double vel[]`

  Fixed velocities at main boundaries on outside of simulation box
- `static double velos[]`

  Oscillating velocities (amplitudes) at main boundaries on outside of simulation box
- `static velospf[]`

  Period or frequency for oscillating velocities at main boundaries
- `static int pf[]`

  Flags to indicate whether period or frequency is specified for oscillating velocities at main boundaries
- `static double conc[]`

  Fixed solute concentrations at main boundaries on outside of simulation box

- `static double temp[]`

  Fixed temperatures at main boundaries on outside of simulation box

- `static double tempdt[]`

  Heating rate at main boundaries on outside of simulation box

14.6 dldpd.java

Side panel for DPD.

Sets up panel on left hand side of main window for DPD (Dissipative Particle Dynamics) simulations, with buttons for Define DPD System, Set DPD Interactions, Change DPD Code, Compile DPD Code, Run DPD Program, Process DPD Data, Plot DPD Results and Close DPD Panel.

14.6.1 Classes

- `class dldpd`

14.6.2 Function/Subroutine Documentation

dldpd

```
public dldpd()
```

Sets up DPD side panel for DL_MESO GUI, laying out buttons in a single column.

14.7 dpdsysdim.java

System boundaries for DPD simulations.

Sets boundary conditions for DPD simulation, using periodic boundaries without frozen particle walls as the default: used to store global properties required for both Define DPD System and Set DPD Interactions windows.

14.7.1 Classes

- `class dpdsysdim`

14.7.2 Variables

- `static int srftyp`

  Selected surface type (corresponds to *srftype* in DL_MESO_DPD)

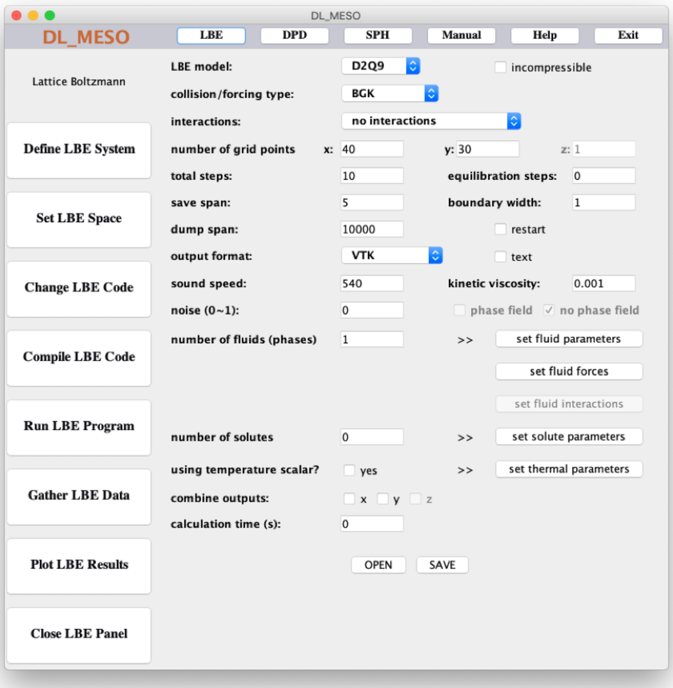
- `static boolean frzwalls`

  Switch for frozen bead walls (corresponds to *lfrzwall* in DL_MESO_DPD)

14.8 setlbeSys.java

Window for Define LBE System.

Sets up window for Define LBE System, including pull-down (combo) boxes for lattice scheme, collision/forcing type, interaction type and output file format, checkboxes for options to use incompressible fluids, simulation restart, text format for output files, temperature scalar and combining outputs for parallel calculations, text boxes to type in values, and buttons to open existing *lbin.sys* file, save information to lbin.sys file and open pop-up windows for fluid parameters, fluid forces, solute parameters and thermal parameters.



LBE model: D2Q9 incompressible
collision/forcing type: BGK
interactions: no interactions
number of grid points x: 40 y: 30 z: 1
total steps: 10 equilibration steps: 0
save span: 5 boundary width: 1
dump span: 10000 restart
output format: VTK text
sound speed: 540 kinetic viscosity: 0.001
noise (0~1): 0 phase field ✓ no phase field
number of fluids (phases) 1
number of solutes 0
using temperature scalar? yes
combine outputs: x y z
calculation time (s): 0

14.8.1 Classes

- `class setlbeSys`

14.8.2 Function/Subroutine Documentation

setlbeSys

```
public setlbeSys()
```

Sets up Define LBE System window for DL_MESO GUI, laying out elements onto a grid.

14.9 setlbeSysEvt.java

Events for Define LBE System.

Implements actions for Define LBE System (including pop-up windows for setting fluid parameters, forces on fluids, solute and thermal parameters): opening *lbin.sys* file into GUI on clicking OPEN button, reading values from text fields, pull-down (combo) boxes and checkboxes, and saving information to lbin.sys file on clicking SAVE button (and saving values from pop-up boxes into memory when relevant save buttons are clicked).

14.9.1 Classes

- `class setlbeSysEvt`

14.9.2 Function/Subroutine Documentation

setlbeSysEvt

```
public setlbeSysEvt(setlbeSys ini)
public setlbeSysEvt(setFluid fluini)
public setlbeSysEvt(setFluidForce fluforini)
public setlbeSysEvt(setFluidInteract fluinterini)
public setlbeSysEvt(setSolute solini)
public setlbeSysEvt(setThermal theini)
```

Sets currently open Define LBE System window and pop-up windows to check for user actions and implement actions in response.

Parameters

ini	setlbeSys	Instance of current Define LBE System window
fluini	setFluid	Instance of LBE Fluid Properties pop-up window
fluforini	setFluidForce	Instance of LBE Fluid Forces pop-up window
fluinterini	setFluidInteract	Instance of LBE Fluid Interaction Properties pop-up window
solini	setSolute	Instance of LBE Solute Properties pop-up window
theini	setThermal	Instance of LBE Thermal Properties pop-up window

saveflu

```
void saveflu()
```

Reads values for initial velocity, fluid densities, relaxation times etc. from text boxes in LBE Fluid Properties pop-up window and stores them in memory before closing window.

saveflufor

```
void saveflufor()
```

Reads values for body, oscillating and Boussinesq forces acting on fluids and oscillation period/frequency from text boxes and pull-down (combo) box in LBE Fluid Forces pop-up window and stores them in memory before closing window.

savefluint

```
void savefluint()
```

Reads values for interaction parameters, types and parameters for equations of state etc. from text boxes and pull-down (combo) boxes in LBE Fluid Interaction Properties pop-up window and stores them in memory before closing window.

savesol

```
void savesol()
```

Reads values for initial solute concentrations and solute relaxation times from text boxes in LBE Solute Properties pop-up window and stores them in memory before closing window.

savethe

```
void savethe()
```

Reads values for initial temperature and heating rate, Boussinesq temperatures and thermal relaxation time from text boxes in LBE Thermal Properties pop-up window and stores them in memory before closing window.

savelbe

```
void savelbe()
```

Creates new *lbin.sys* file (or overwrites existing one) after reading in values from text boxes, pull-down (combo) boxes and checkboxes in Define LBE System window, also using values previously obtained from pop-up windows.

openlbe

```
void openlbe()
```

Opens and reads *lbin.sys* file, storing values required for pop-up windows (on fluid properties, fluid forces, interaction properties, solute properties and thermal properties) in memory and writing or setting values in text boxes, pull-down (combo) boxes and checkboxes in Define LBE System window. (Some values read from this file - particularly those related to boundary conditions - are also used in the Set LBE Space window.)

setfluitparameter

```
void setfluitparameter(int totf,
                       int nd,
                       int nq,
                       int coll,
                       int inter,
                       double[] densfic,
                       double[] initv,
                       double[] relaxtime,
                       double[] mrtrelaxtime,
                       double trtm,
                       int[] rheo,
                       double[] rheoa,
```

(continues on next page)

(continued from previous page)

```
                                   double[] rheob,
                                   double[] rheoc,
                                   double[] rheod,
                                   double[] rheon)
```

Launches LBE Fluid Properties pop-up window with number of fluids, lattice scheme, collision and interaction types, initial and constant densities, initial velocity, relaxation times/frequencies etc.

Parameters

totf	int	Number of fluids
nd	int	Number of spatial dimensions
nq	int	Number of lattice links (based on lattice scheme pull-down box in Define LBE System)
coll	int	Selected collision scheme (based on pull-down box in Define LBE System window)
inter	int	Selected interaction type (based on pull-down box in Define LBE System window)
densfic	double[]	Initial and constant densities for fluids
initv	double[]	Initial velocity for system
relaxtime	double[]	Relaxation times for fluids (including bulk and CLBE third and fourth order values)
mrtrelax-time	double[]	System-wide relaxation frequencies for MRT collisions
trtm	double	TRT magic number
rheo	int[]	Rheological models for fluids (selecting item for pull-down box)
rheoa	double[]	Parameters a for rheological models
rheob	double[]	Parameters b for rheological models
rheoc	double[]	Parameters c for rheological models
rheod	double[]	Parameters d for rheological models
rheon	double[]	Parameters n (power law indices) for rheological models

setfluitforceparameter

```
void setfluitforceparameter(int totf,
                            int dim,
                            int coll,
                            double[] bdf,
                            double[] oscf,
                            double[] bousf,
                            double ocfp,
                            int fp)
```

Launches LBE Fluid Forces pop-up window with numbers of fluids and dimensions, collision type, body, oscillating and Boussinesq forces acting on fluids and oscillating period/frequency.

Parameters

totf	int	Number of fluids
dim	int	Number of spatial dimensions
coll	int	Selected collision scheme (based on pull-down box in Define LBE System window)
bdf	double[]	Body (constant) forces acting on fluids
oscf	double[]	Oscillating forces (amplitudes) acting on fluids
bousf	double[]	Boussinesq (buoyancy) forces acting on fluids
ocfp	double	Frequency or period for oscillating forces
pf	int	Switch indicating frequency or period for oscillating forces (used for pull-down box)

setfluitintparameter

```
void setfluitintparameter(int totf,
                          int inter,
                          double[] fluidinteract,
                          int[] eos,
                          int eoscrit,
                          double[] eosa,
                          double[] eosb,
                          double[] acentric,
                          double[] psi0,
                          double[] quadw,
                          int[] wettype,
                          double[] wallinteract,
                          double segregate,
                          double gascon,
                          double tempsys,
                          double kappa,
                          double taumob,
                          double mobparam)
```

Launches LBE Fluid Interaction Properties pop-up window with numbers of fluids, interaction type, interaction parameters, selected equations of state and related parameters etc.

Parameters

totf	int	Number of fluids
inter	int	Selected interaction type (based on pull-down box in Define LBE System window)
fluidin-teract	dou-ble[]	Interaction parameters between fluid pairs
eos	int[]	Selected equations of states for fluids (selecting items for pull-down boxes)
eoscrit	int	Switch to indicate using equation-of-state parameters or critical properties
eosa	dou-ble[]	Parameters a for equations of state or critical temperatures T_c for each fluid
eosb	dou-ble[]	Parameters b for equations of state or critical pressures P_c for each fluid
acentric	dou-ble[]	Acentric parameters ω for equations of state, for each fluid
psi0	dou-ble[]	Maximum pseudopotential parameters ψ_0 for equations of state with 1994 Shan-Chen thermodynamically consistent model
quadw	dou-ble[]	Shan-Chen quadratic weighting parameter for fluid pairs
wettype	int[]	Surface wetting type for fluids (selecting items for pull-down boxes)
wallinter-act	dou-ble[]	Wall interaction parameters for fluids
segre-gate	dou-ble	Segregation parameter β between fluid pairs for Lishchuk interactions
gascon	dou-ble	Universal gas constant R used for equations of state
tempsys	dou-ble	Isothermal system temperature used for equations of state without temperature field
kappa	dou-ble	Surface tension parameter κ for Swift free-energy interactions
taumob	dou-ble	Mobility relaxation time τ_ϕ for Swift free-energy interactions
mob-param	dou-ble	Mobility parameter Γ for Swift free-energy interactions

setsoluteparameter

```
void setsoluteparameter(int totc,
                        int dim,
                        double[] solrelax,
                        double[] concs)
```

Launches LBE Solute Properties pop-up window with numbers of solutes and dimensions, solute relaxation times and initial solute concentrations.

Parameters

totc	int	Number of solutes
dim	int	Number of spatial dimensions
solrelax	double[]	Solute relaxation times
concs	double[]	Initial solute concentrations

setthermalparameter

```
void setthermalparameter(int dim,
                         double heatrelax,
                         double boustemph,
                         double boustempl,
                         double tempsi,
                         double heatratei)
```

Launches LBE Thermal Properties pop-up window with number of spatial dimensions, thermal relaxation time, high and low Boussinesq temperatures, initial temperature and heating rate.

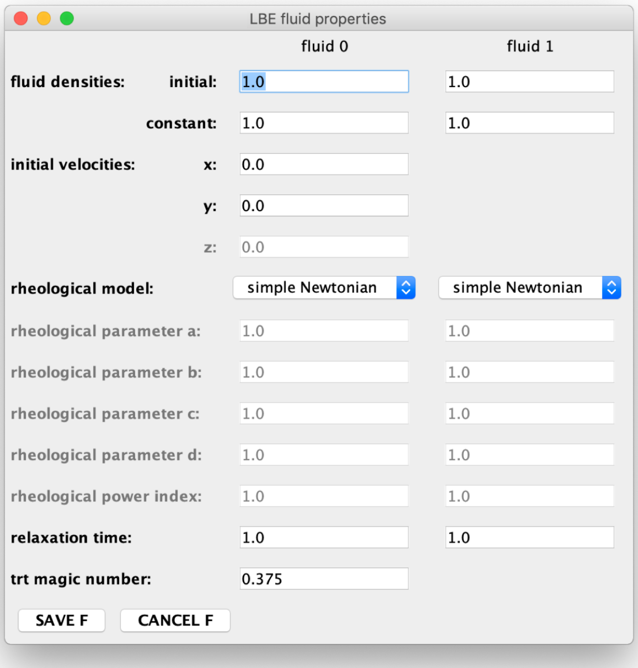
Parameters

dim	int	Number of spatial dimensions
heatrelax	double	Thermal relaxation time τ_t
boustemph	double	High temperature for Boussinesq approximation
boustempl	double	Low temperature for Boussinesq approximation
tempsi	double	Initial system temperature
heatratesi	double	Initial heating rate

14.10 setFluid.java

Pop-up window for LBE Fluid Properties.

Sets up pop-up window for LBE Fluid Properties (obtained from button in Define LBE System), setting labels and adding text boxes for initial and constant fluid densities, initial velocity, relaxation times etc. based on number of fluids (given as an input), filling values into text boxes based on either default values or those obtained from reading *lbin.sys* file (*setlbeSysEvt.java* reads values from these boxes when saving lbin.sys file).



		fluid 0	fluid 1
fluid densities:	initial:	1.0	1.0
	constant:	1.0	1.0
initial velocities:	x:	0.0	
	y:	0.0	
	z:	0.0	
rheological model:		simple Newtonian	simple Newtonian
rheological parameter a:		1.0	1.0
rheological parameter b:		1.0	1.0
rheological parameter c:		1.0	1.0
rheological parameter d:		1.0	1.0
rheological power index:		1.0	1.0
relaxation time:		1.0	1.0
trt magic number:		0.375	

14.10.1 Classes

- `class setFluid`

14.10.2 Function/Subroutine Documentation

setFluid

```
public setFluid(int totf,
                int nd,
                int nq,
                int coll,
                int inter,
                double[] densf,
                double[] iniv,
                double[] relaxtime,
                double[] mrtrelaxfreq,
                double trtm,
                int[] rheomod,
                double[] rheoa,
                double[] rheob,
                double[] rheoc,
                double[] rheod,
                double[] rheon)
```

Sets up LBE Fluid Properties pop-up window for DL_MESO GUI, laying out elements onto a grid and using supplied numbers of fluids, dimensions, lattice links (for lattice scheme), collision and interaction types, initial and constant fluid densities, initial velocity, relaxation times, rheology model and parameters to set up the text fields, pull-down (combo) boxes etc.

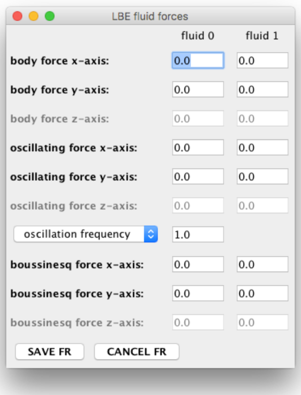
Parameters

totf	int	Number of fluids
nd	int	Number of spatial dimensions
nq	int	Number of lattice links (based on lattice scheme pull-down box in Define LBE System)
coll	int	Selected collision scheme (based on pull-down box in Define LBE System window)
inter	int	Selected interaction type (based on pull-down box in Define LBE System window)
densf	double[]	Initial and constant densities for fluids
iniv	double[]	Initial velocity for system
relaxtime	double[]	Relaxation times for fluids (including bulk and CLBE third and fourth order values)
mrtrelaxfreq	double[]	System-wide relaxation frequencies for MRT collisions
trtm	double	TRT magic number
rheomod	int[]	Rheological models for fluids (selecting item for pull-down box)
rheoa	double[]	Parameters a for rheological models
rheob	double[]	Parameters b for rheological models
rheoc	double[]	Parameters c for rheological models
rheod	double[]	Parameters d for rheological models
rheon	double[]	Parameters n (power law indices) for rheological models

14.11 setFluidForce.java

Pop-up window for LBE Fluid Forces.

Sets up pop-up window for LBE Fluid Forces (obtained from button in Define LBE System), setting labels and adding text boxes for body (constant), sinusoid oscillating and Boussinesq forces based on number of fluids (given as an input) and pull-down (combo) box to specify oscillating frequency or time period, filling values into text boxes based on either default values or those obtained from reading *lbin.sys* file (*setlbeSysEvt.java* reads values from these boxes when saving lbin.sys file).



	fluid 0	fluid 1
body force x-axis:	0.0	0.0
body force y-axis:	0.0	0.0
body force z-axis:	0.0	0.0
oscillating force x-axis:	0.0	0.0
oscillating force y-axis:	0.0	0.0
oscillating force z-axis:	0.0	0.0
oscillation frequency	1.0	
boussinesq force x-axis:	0.0	0.0
boussinesq force y-axis:	0.0	0.0
boussinesq force z-axis:	0.0	0.0

14.11.1 Classes

- `class setFluidForce`

14.11.2 Function/Subroutine Documentation

setFluidForce

```
public setFluidForce(int totf,
                     int dim,
                     int coll,
                     double[] bdf,
                     double[] oscf,
                     double[] bousf,
                     double oscfrpe,
                     int fp)
```

Sets up LBE Fluid Forces pop-up window for DL_MESO GUI, laying out elements onto a grid and using supplied numbers of fluids, dimensions, collision type, body, oscillating and Boussinesq (buoyancy) forces, oscillating period/frequency and switch to set up the text fields, pull-down (combo) boxes etc.

Parameters

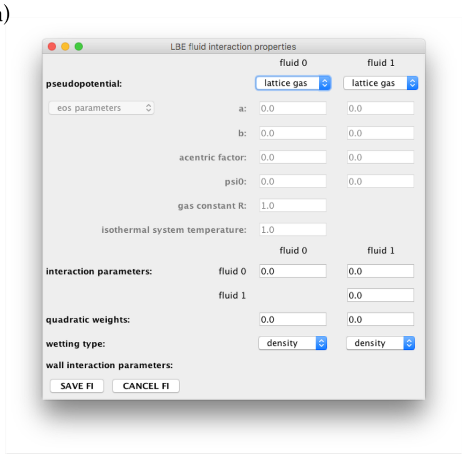
totf	int	Number of fluids
dim	int	Number of spatial dimensions
coll	int	Selected collision scheme (based on pull-down box in Define LBE System window)
bdf	double[]	Body (constant) forces acting on fluids
oscf	double[]	Oscillating forces (amplitudes) acting on fluids
bousf	double[]	Boussinesq (buoyancy) forces acting on fluids
oscfrpe	double	Frequency or period for oscillating forces
pf	int	Switch indicating frequency or period for oscillating forces (used for pull-down box)

14.12 setFluidInteract.java

Pop-up window for LBE Fluid Interaction Properties.

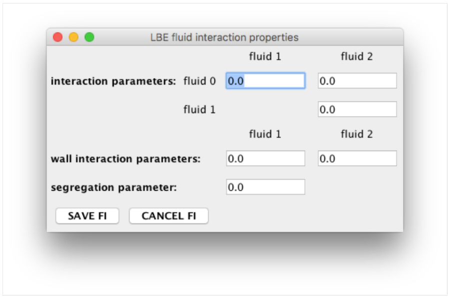
Sets up pop-up window for LBE Fluid Interaction Properties (obtained from button in Define LBE System), setting labels and adding text boxes and pull-down (combo) boxes for interaction parameters, pseudopotential type and/or equation of state, surface wetting etc. based on interaction type and number of fluids (given as inputs), filling values into text boxes and selecting values in pull-down boxes based on either default values or those obtained from reading *lbin.sys* file (*setlbeSysEvt.java* reads values from these boxes when saving lbin.sys file).

(a)



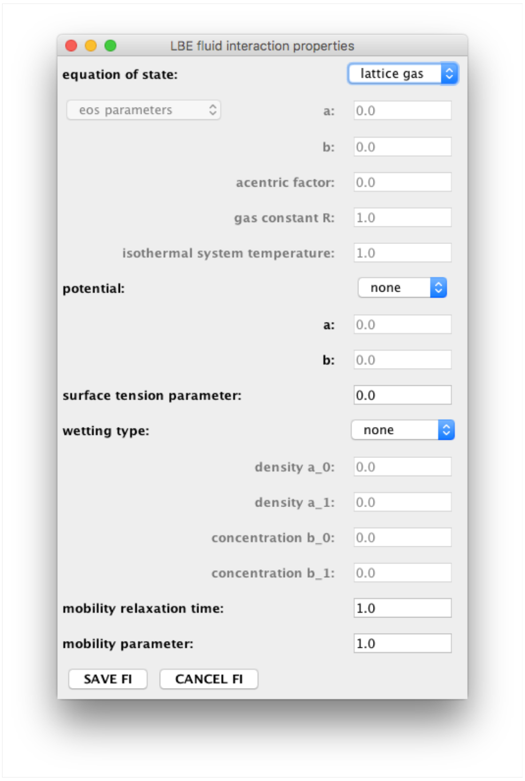
		fluid 0	fluid 1
pseudopotential:		lattice gas	lattice gas
eos parameters	a:	0.0	0.0
	b:	0.0	0.0
	acentric factor:	0.0	0.0
	psi0:	0.0	0.0
	gas constant R:	1.0	
	isothermal system temperature:	1.0	
		fluid 0	fluid 1
interaction parameters:	fluid 0	0.0	0.0
	fluid 1		0.0
quadratic weights:		0.0	0.0
wetting type:		density	density
wall interaction parameters:			

(b)



		fluid 1	fluid 2
interaction parameters:	fluid 0	0.0	0.0
	fluid 1		0.0
		fluid 1	fluid 2
wall interaction parameters:		0.0	0.0
segregation parameter:		0.0	

(c)



equation of state:		lattice gas
eos parameters	a:	0.0
	b:	0.0
	acentric factor:	0.0
	gas constant R:	1.0
	isothermal system temperature:	1.0
potential:		none
	a:	0.0
	b:	0.0
surface tension parameter:		0.0
wetting type:		none
	density a_0:	0.0
	density a_1:	0.0
	concentration b_0:	0.0
	concentration b_1:	0.0
mobility relaxation time:		1.0
mobility parameter:		1.0

Fig. 14.1: Fluid interaction pop-up windows: (a) Shan/Chen pseudopotential interactions, (b) Lishchuk continuum-based interactions, (c) Swift free-energy interactions

14.12.1 Classes

- `class setFluidInteract`

14.12.2 Function/Subroutine Documentation

setFluidInteract

```
public setFluidInteract(int totf,
                        int inter,
                        double[] interact,
                        int[] eos,
                        int eoscrt,
                        double[] eosa,
```

(continues on next page)

(continued from previous page)

```
                        double[] eosb,
                        double[] acentric,
                        double[] psi0,
                        double[] quadw,
                        int[] wettyp,
                        double[] wallinteract,
                        double segregate,
                        double gascon,
                        double tempsys,
                        double kappa,
                        double taumob,
                        double mobparam)
```

Sets up LBE Fluid Interaction Properties pop-up window for DL_MESO GUI, laying out elements onto a grid and using supplied number of fluids, interaction type, interaction parameters, fluid equations of state and parameters, wetting type and parameters etc. to set up the text fields, pull-down (combo) boxes etc.

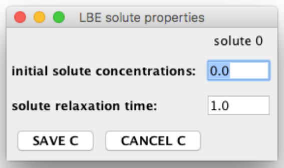
Parameters

totf	int	Number of fluids
inter	int	Selected interaction type (based on pull-down box in Define LBE System window)
interact	double[]	Interaction parameters between fluid pairs
eos	int[]	Selected equations of states for fluids (selecting items for pull-down boxes)
eoscrt	int	Switch to indicate using equation-of-state parameters or critical properties
eosa	double[]	Parameters a for equations of state or critical temperatures T_c for each fluid
eosb	double[]	Parameters b for equations of state or critical pressures P_c for each fluid
acentric	double[]	Acentric parameters ω for equations of state, for each fluid
psi0	double[]	Maximum pseudopotential parameters ψ_0 for equations of state with 1994 Shan-Chen thermodynamically consistent model
quadw	double[]	Shan-Chen quadratic weighting parameter for fluid pairs
wettyp	int[]	Surface wetting type for fluids (selecting items for pull-down boxes)
wallinteract	double[]	Wall interaction parameters for fluids
segregate	double	Segregation parameter β between fluid pairs for Lishchuk interactions
gascon	double	Universal gas constant R used for equations of state
tempsys	double	Isothermal system temperature used for equations of state without temperature field
kappa	double	Surface tension parameter κ for Swift free-energy interactions
taumob	double	Mobility relaxation time τ_ϕ for Swift free-energy interactions
mobparam	double	Mobility parameter Γ for Swift free-energy interactions

14.13 setSolute.java

Pop-up window for LBE Solute Properties.

Sets up pop-up window for LBE Solute Properties (obtained from button in Define LBE System), setting labels and adding text boxes for initial solute concentrations and solute relaxation times based on number of solutes (given as an input), filling values into text boxes based on either default values or those obtained from reading *lbin.sys* file (*setlbeSysEvt.java* reads values from these boxes when saving lbin.sys file)



solute 0
initial solute concentrations: 0.0
solute relaxation time: 1.0

14.13.1 Classes

- `class setSolute`

14.13.2 Function/Subroutine Documentation

setSolute

```
public setSolute(int totc,
                 int dim,
                 double[] solrelax,
                 double[] concs)
```

Sets up LBE Solute Properties pop-up window for DL_MESO GUI, laying out elements onto a grid and using supplied numbers of solutes and dimensions, solute relaxation times and initial concentrations to set up the text fields etc.

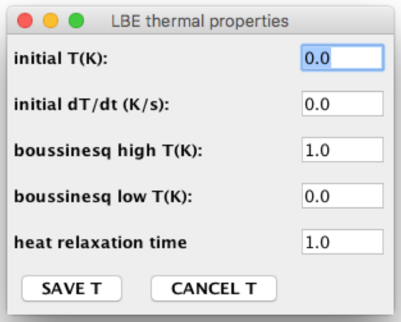
Parameters

totc	int	Number of solutes
dim	int	Number of spatial dimensions
solrelax	double[]	Solute relaxation times
concs	double[]	Initial solute concentrations

14.14 setThermal.java

Pop-up window for LBE Thermal Properties.

Sets up pop-up window for LBE Thermal Properties (obtained from button in Define LBE System), setting labels and adding text boxes for initial temperature and heating rate, Boussinesq high and low temperatures and thermal relaxation time, filling values into text boxes based on either default values or those obtained from reading *lbin.sys* file (*setlbeSysEvt.java* reads values from these boxes when saving lbin.sys file).



initial T(K): 0.0
initial dT/dt (K/s): 0.0
boussinesq high T(K): 1.0
boussinesq low T(K): 0.0
heat relaxation time 1.0

14.14.1 Classes

- `class setThermal`

14.14.2 Function/Subroutine Documentation

setThermal

```
public setThermal(int dim,
                  double heatrelax,
                  double boustemph,
                  double boustempl,
                  double tempsi,
                  double heatratesi)
```

Sets up LBE Thermal Properties pop-up window for DL_MESO GUI, laying out elements onto a grid and using supplied number of dimensions, thermal relaxation time, high and low Boussinesq temperatures, initial temperature and heating rate to set up the text fields etc.

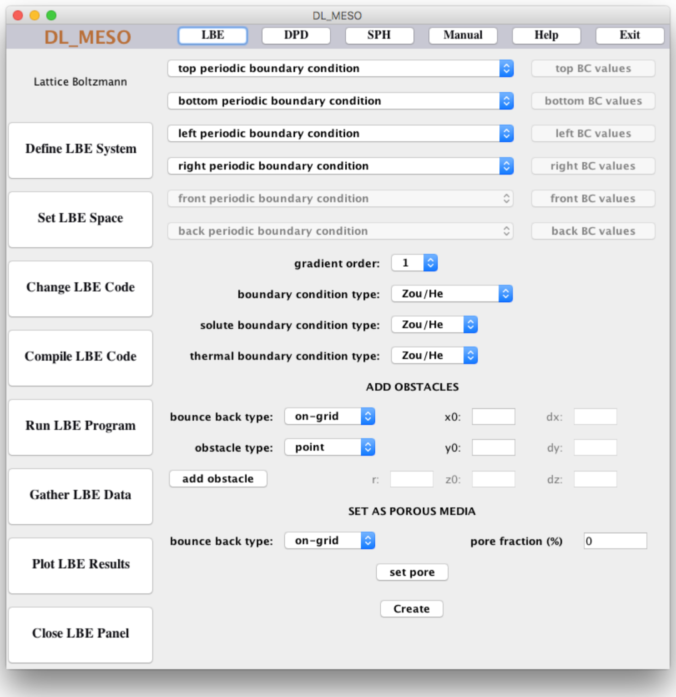
Parameters

dim	int	Number of spatial dimensions
heatrelax	double	Thermal relaxation time τ_t
boustemph	double	High temperature for Boussinesq approximation
boustempl	double	Low temperature for Boussinesq approximation
tempsi	double	Initial system temperature
heatratesi	double	Initial heating rate

14.15 setlbeSpa.java

Window for Set LBE Space.

Sets up window for Set LBE Space, including pull-down (combo) boxes to specify boundary conditions at planar/edge boundaries, order for calculating gradients (for surface interactions), boundary condition types, solid obstacle specifications (bounce back and obstacle types) and bounce back type for porous media, text boxes for position and sizes of solid obstacles and pore fraction, and buttons to open pop-up windows to specify properties at planar/edge boundaries, to add solid obstacle, set pore for porous media, create *lbin.spa* file and add required values for boundary conditions to *lbin.sys* file. (Note that the system dimensions, including grid size and numbers of fluids/solutes/temperature fields, are required as an input.)



top periodic boundary condition
bottom periodic boundary condition
left periodic boundary condition
right periodic boundary condition
front periodic boundary condition
back periodic boundary condition
gradient order: 1
boundary condition type: Zou/He
solute boundary condition type: Zou/He
thermal boundary condition type: Zou/He
bounce back type: on-grid
obstacle type: point
bounce back type: on-grid
pore fraction (%) 0

14.15.1 Classes

- `class setlbeSpa`

14.15.2 Function/Subroutine Documentation

setlbeSpa

```
public setlbeSpa(int dim)
```

Sets up Set DPD Interactions window for DL_MESO GUI, laying out elements onto a grid and using number of spatial dimensions to determine which elements should be active (usable).

Parameters

dim	int	Number of spatial dimensions

14.16 setlbeSpaEvt.java

Events for Set LBE Space.

Implements actions for Set LBE Space (including pop-up windows for setting properties for planar/edge boundaries): saving information to new *lbin.spa* file and adding information to *lbin.sys* file, launching pop-up windows, obtaining velocities/fluid densities/solute concentrations/temperatures from text boxes, adding solid obstacles and setting porous media (both to memory) on button clicks, enabling buttons and text boxes based on drop-down (combo) box selections.

14.16.1 Classes

- `class setlbeSpaEvt`

14.16.2 Function/Subroutine Documentation

setlbeSpaEvt

```
public setlbeSpaEvt(setlbeSpa setspace)
public setlbeSpaEvt(setBound boundlbe)
```

Sets currently open Set LBE Space window and LBE Boundary Conditions pop-up window to check for user actions and implement actions in response.

Parameters

setspace	setdpdInteract	Instance of current Change DPD Code window
boundlbe	setBound	Instance of LBE Boundary Conditions pop-up window

topBoundIssue

```
void topBoundIssue(String topbis)
```

Adds boundary condition codes to *lbin.spa* file for top boundary of simulation box: used for all boundary conditions based on the start of a 'word' specifying a general type, e.g. constant velocity/density, constant solute concentration/bounceback and constant temperature/bounceback.

Parameters

topbis	String	Start of boundary condition 'word' for top boundary

downBoundIssue

```
void downBoundIssue(String topbis)
```

Adds boundary condition codes to *lbin.spa* file for bottom boundary of simulation box: used for all boundary conditions based on the start of a 'word' specifying a general type, e.g. constant velocity/density, constant solute concentration/bounceback and constant temperature/bounceback.

Parameters

topbis	String	Start of boundary condition 'word' for bottom boundary

leftBoundIssue

```
void leftBoundIssue(String topbis)
```

Adds boundary condition codes to *lbin.spa* file for left boundary of simulation box: used for all boundary conditions based on the start of a 'word' specifying a general type, e.g. constant velocity/density, constant solute concentration/bounceback and constant temperature/bounceback.

Parameters

topbis	String	Start of boundary condition 'word' for left boundary

rightBoundIssue

```
void rightBoundIssue(String topbis)
```

Adds boundary condition codes to *lbin.spa* file for right boundary of simulation box: used for all boundary conditions based on the start of a 'word' specifying a general type, e.g. constant velocity/density, constant solute concentration/bounceback and constant temperature/bounceback.

Parameters

topbis	String	Start of boundary condition 'word' for left boundary

frontBoundIssue

```
void frontBoundIssue(String topbis)
```

Adds boundary condition codes to *lbin.spa* file for front boundary of simulation box: used for all boundary conditions based on the start of a 'word' specifying a general type, e.g. constant velocity/density, constant solute concentration/bounceback and constant temperature/bounceback. (Only applied for three-dimensional simulations.)

Parameters

topbis	String	Start of boundary condition 'word' for front boundary

backBoundIssue

```
void backBoundIssue(String topbis)
```

Adds boundary condition codes to *lbin.spa* file for back boundary of simulation box: used for all boundary conditions based on the start of a 'word' specifying a general type, e.g. constant velocity/density, constant solute concentration/bounceback and constant temperature/bounceback. (Only applied for three-dimensional simulations.)

Parameters

topbis	String	Start of boundary condition 'word' for back boundary

checkexist

```
void checkexist()
```

Writes boundary condition codes to *lbin.spa* file based on pull-down (combo) boxes for all main boundaries, and write boundary condition types and values (velocities, fluid densities, solute concentrations, temperatures and heating rates) to *lbin.sys* file. (Instigated by clicking

addpore

```
void addpore(String str1,
             double porefr,
             int tx,
             int ty,
             int tz)
```

Determines lattice points to apply bounceback boundary condition for a porous medium based on a given porosity and randomised selection, and writes relevant boundary condition codes to *lbin.spa* file.

Parameters

str1	String	Boundary condition 'word' describing required type of bounceback boundary condition
porefr	double	Pore fraction for fluid points in porous medium, given as percentage
tx	int	Total number of grid points in x-dimension
ty	int	Total number of grid points in y-dimension
tz	int	Total number of grid points in z-dimension

addBlock

```
void addBlock(String str1,
              int xpos,
              int ypos,
              int xdis,
              int ydis)
```

Determines lattice points to apply bounceback and blank site boundary conditions for a two-dimensional rectangular region starting at the bottom-left corner and extending by set numbers of lattice points in horizontal and vertical directions, and writes relevant boundary condition codes to *lbin.spa* file.

Parameters

str1	String	Boundary condition 'word' describing required type of (bounceback) boundary condition for outside of rectangular box
xpos	int	Position of bottom-left corner of rectangle (x-coordinate)
ypos	int	Position of bottom-left corner of rectangle (y-coordinate)
xdis	int	Extent of rectangle in x-dimension
ydis	int	Extent of rectangle in y-dimension

addblock

```
void addblock(String str1,
              int xpos,
              int ypos,
              int zpos,
              int xdis,
              int ydis,
              int zdis)
```

Determines lattice points to apply boundary conditions for a three-dimensional cuboidal region starting at the bottom-left-back corner and extending by set numbers of lattice points in all three Cartesian directions, and writes relevant boundary condition codes to *lbin.spa* file. This subroutine is used to add both cuboidal obstacles and main boundaries on the outside of the simulation box.

Parameters

str1	String	Boundary condition 'word' describing required type of boundary condition for outside of cuboidal box
xpos	int	Position of bottom-left-back corner of cuboid (x-coordinate)
ypos	int	Position of bottom-left-back corner of cuboid (y-coordinate)
zpos	int	Position of bottom-left-back corner of cuboid (z-coordinate)
xdis	int	Extent of cuboid in x-dimension
ydis	int	Extent of cuboid in y-dimension
zdis	int	Extent of cuboid in z-dimension

addsphere

```
void addsphere(String str1)
```

Determines lattice points to apply bounceback and blank site boundary conditions for a sphere, reading the coordinates of the centre and the radius from text boxes in Set LBE Space window, and writes relevant boundary condition codes to *lbin.spa* file.

Parameters

str1	String	Boundary condition 'word' describing required type of bounceback boundary condition

addcylinder

```
void addcylinder(String str1)
```

Determines lattice points to apply bounceback and blank site boundary conditions for a cylinder, reading the coordinates of the centre and the radius from text boxes in Set LBE Space window, and writes relevant boundary condition codes to *lbin.spa* file.

Parameters

str1	String	Boundary condition 'word' describing required type of bounceback boundary condition

addpoint

```
void addpoint(String str1)
```

Applies a bounceback boundary condition to an individual lattice point, reading its coordinates from text text boxes in Set LBE Space window, and writes the relevant boundary condition code to *lbin.spa* file.

Parameters

str1	String	Boundary condition 'word' describing required type of bounceback boundary condition

setboundarycond

```
void setboundarycond(int bc,
                     int[] bcsel,
                     int dim,
                     int totf,
                     int totc,
                     int tott,
                     double[] vel,
                     double[] velos,
                     double[] velospf,
                     int[] pf,
                     double[] dens,
                     double[] conc,
                     double[] temp,
                     double[] tempdt)
```

Launches LBE Boundary Conditions pop-up window with supplied boundary, boundary condition, numbers of dimensions, fluids, solutes and temperature fields, and properties at boundaries.

Parameters

bc	int	Boundary condition (0 = top, 1 = bottom, 2 = left, 3 = right, 4 = front, 5 = back)
bcsel	int[]	Selected boundary condition type (based on pull-down box in Set LBE Space window)
dim	int	Number of spatial dimensions
totf	int	Number of fluids
totc	int	Number of solutes
tott	int	Number of temperature fields
vel	double[]	Fixed velocities at boundaries
velos	double[]	Oscillating velocity amplitudes at boundaries
velospf	double[]	Oscillating frequencies or periods at boundaries
pf	int[]	Switches to indicate using frequencies or periods for oscillating boundaries
dens	double[]	Fixed fluid densities at boundaries
conc	double[]	Fixed solute concentrations at boundaries
temp	double[]	Fixed temperatures at boundaries
tempdt	double[]	Heating rates at boundaries

fConvert

```
int fConvert(String str1)
```

Function to convert a boundary condition ‘word’ into a numerical code for writing to a *lbin.spa* file.

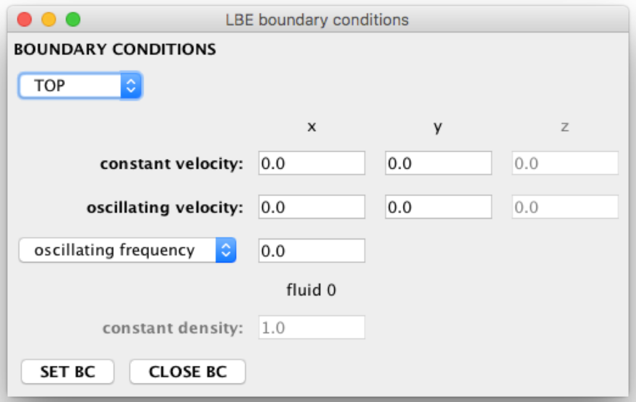
Parameters

str1	String	Boundary condition ‘word’

14.17 setBound.java

Pop-up window for LBE Boundary Conditions.

Sets up pop-up window for LBE Boundary Conditions (obtained from buttons in Set LBE Space), setting labels, adding and filling text boxes for specified values for constant velocities, fluid densities etc. (either default values or taken from lbin.sys file read in by Define LBE System window), a pull-down (combo) box to specify planar/edge boundary (initially selected based on the button pressed in Set LBE Space window, but can be changed while pop-up window is open), and buttons to set boundary values to memory and close window (*setlbeSpaEvt.java* uses values stored in memory when appending to *lbin.sys* file).



	x	y	z
constant velocity:	0.0	0.0	0.0
oscillating velocity:	0.0	0.0	0.0
oscillating frequency	0.0		
	fluid 0		
constant density:	1.0		

14.17.1 Classes

- `class setBound`

14.17.2 Function/Subroutine Documentation

setBound

```
public setBound(int bc,
                int[] bcsel,
                int dim,
                int totf,
                int totc,
                int tott,
                double[] vel,
                double[] velos,
                double[] velospf,
                int[] pf,
                double[] dens,
                double[] conc,
                double[] temp,
                double[] tempdt)
```

Sets up LBE Boundary Conditions pop-up window for DL_MESO GUI, laying out elements onto a grid and using supplied boundary, boundary condition, numbers of dimensions, fluids, solutes and temperature fields, and properties at boundaries to set up the text fields, pull-down (combo) boxes etc.

Parameters

bc	int	Boundary condition (0 = top, 1 = bottom, 2 = left, 3 = right, 4 = front, 5 = back)
bcsel	int[]	Selected boundary condition type (based on pull-down box in Set LBE Space window)
dim	int	Number of spatial dimensions
totf	int	Number of fluids
totc	int	Number of solutes
tott	int	Number of temperature fields
vel	double[]	Fixed velocities at boundaries
velos	double[]	Oscillating velocity amplitudes at boundaries
velospf	double[]	Oscillating frequencies or periods at boundaries
pf	int[]	Switches to indicate using frequencies or periods for oscillating boundaries
dens	double[]	Fixed fluid densities at boundaries
conc	double[]	Fixed solute concentrations at boundaries
temp	double[]	Fixed temperatures at boundaries
tempdt	double[]	Heating rates at boundaries

14.18 changelbecode.java

Window for Change LBE Code.

Sets up window for Change LBE Code, including pull-down (combo) boxes to select text editor and DL_MESO_LBE code module file, text fields for editors and files not included in pull-down boxes, and button to edit selected file with editor.

14.18.1 Classes

- `class changelbecode`

14.18.2 Function/Subroutine Documentation

changelbecode

```
public changelbecode()
```

Sets up Change LBE Code window for DL_MESO GUI, laying out elements onto a grid.

14.19 changelbeEvt.java

Events for Change LBE Code.

Implements actions for Change LBE Code, including selecting text editor and DL_MESO_LBE code module file in pull-down (combo) boxes and text boxes, and pressing EDIT button.

14.19.1 Classes

- `class changelbeEvt`

14.19.2 Function/Subroutine Documentation

changedlbeEvt

```
public changelbeEvt(changelbecode in)
```

Sets currently open Change LBE Code window as window to check for user actions and implement actions in response.

Parameters

in	changelbecode	Instance of current Change LBE Code window

14.20 compilelbe.java

Window for Compile LBE Code.

Sets up window for Compile LBE Code, including pull-down (combo) boxes to select C++ compiler and DL_MESO_LBE code version, text fields for compilers not included in pull-down box and compiler flags, and a button to compile code.

14.20.1 Classes

- `class compilelbe`

14.20.2 Function/Subroutine Documentation

compilelbe

```
public compilelbe()
```

Sets up Compile LBE Code window for DL_MESO GUI, laying out elements onto a grid.

14.21 compilelbeEvt.java

Events for Compile LBE Code.

Implements actions for Compile LBE Code, including selecting C++ compiler, compiler flags and DL_MESO_LBE code version, and pressing button to compile code.

14.21.1 Classes

- `class compilelbeEvt`

14.21.2 Function/Subroutine Documentation

compilelbeEvt

```
public compilelbeEvt(compilelbe in)
```

Sets currently open Compile LBE Code window as window to check for user actions and implement actions in response.

Parameters

in	compilelbe	Instance of current Compile LBE Code window

14.22 rublbe.java

Window for Run LBE Program.

Sets up window for Run LBE Program, including pull-down (combo) box and text field to select command required to launch DL_MESO_LBE, and button to Run LBE code.

14.22.1 Classes

- `class rublbe`

14.22.2 Function/Subroutine Documentation

rublbe

```
public rublbe()
```

Sets up Plot LBE Results window for DL_MESO GUI, laying out elements onto a grid.

14.23 rublbeEvt.java

Events for Run LBE Program.

Implements actions for Run LBE Program, including selecting command for launching DL_MESO_LBE, and pressing Run LBE button.

14.23.1 Classes

- `class rublbeEvt`

14.23.2 Function/Subroutine Documentation

rublbeEvt

```
public rublbeEvt(rublbe in)
```

Sets currently open Run LBE Program window as window to check for user actions and implement actions in response.

Parameters

in	rublbe	Instance of current Run LBE Program window

14.24 gatherlbe.java

Window for Gather LBE Data.

Sets up window for Gather LBE Data, including pull-down (combo) boxes to select file type (utility to launch) and available data in files, and a button to Gather data.

14.24.1 Classes

- `class gatherlbe`

14.24.2 Function/Subroutine Documentation

gatherlbe

```
public gatherlbe()
```

Sets up Gather LBE Data window for DL_MESO GUI, laying out elements onto a grid.

14.25 gatherlbeEvt.java

Events for Gather LBE Data.

Implements actions for Gather LBE Data, including selecting utility and command-line options from pull-down (combo) boxes, and running utility when Gather data button is clicked.

14.25.1 Classes

- `class gatherlbeEvt`

14.25.2 Function/Subroutine Documentation

gatherlbeEvt

```
public gatherlbeEvt(gatherlbe in)
```

Sets currently open Gather LBE Data window as window to check for user actions and implement actions in response.

Parameters

in	gatherlbe	Instance of current Gather LBE Data window

14.26 plotlbe.java

Window for Plot LBE Results.

Sets up window for Plot LBE Results, including pull-down (combo) box to select pre-installed software package to launch, text field for other packages not included in pull-down box, check box to run program in terminal window, and button to launch package for plotting results from DPD calculation.

14.26.1 Classes

- `class plotlbe`

14.26.2 Function/Subroutine Documentation

plotlbe

```
public plotlbe()
```

Sets up Plot LBE Results window for DL_MESO GUI, laying out elements onto a grid.

14.27 plotlbeEvt.java

Events for Plot LBE Results.

Implements actions for Plot LBE Results, including selecting plotting software package in pull-down (combo) and text box, and pressing Plot data button.

14.27.1 Classes

- `class plotlbeEvt`

14.27.2 Function/Subroutine Documentation

plotlbeEvt

```
public plotlbeEvt(plotlbe in)
```

Sets currently open Plot LBE Results window as window to check for user actions and implement actions in response.

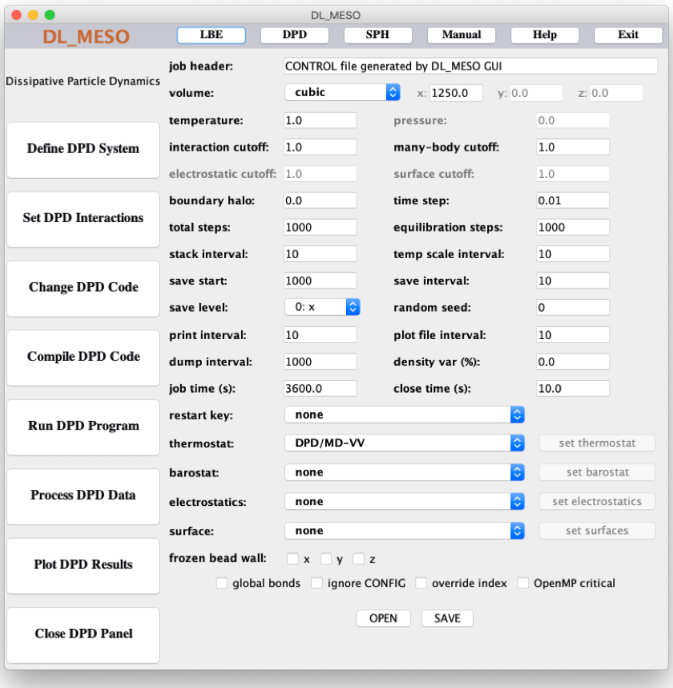
Parameters

in	plotlbe	Instance of current Plot LBE Results window

14.28 setdpdSys.java

Window for Define DPD System.

Sets up window for Define DPD System, including pull-down (combo) boxes for system volume, trajectory save level, restart, thermostat, barostat, electrostatics and surface options, checkboxes for frozen bead walls, global bonds, ignoring CONFIG, overriding particle indices and OpenMP critical options, text boxes to type in values, and buttons to open existing *CONTROL* file, save information to CONTROL file and open pop-up windows for thermostat, barostat, electrostatics and surface parameters.



job header: CONTROL file generated by DL_MESO GUI
volume: cubic x: 1250.0 y: 0.0 z: 0.0

temperature:	1.0	pressure:	0.0
interaction cutoff:	1.0	many-body cutoff:	1.0
electrostatic cutoff:	1.0	surface cutoff:	1.0
boundary halo:	0.0	time step:	0.01
total steps:	1000	equilibration steps:	1000
stack interval:	10	temp scale interval:	10
save start:	1000	save interval:	10
save level:	0: x	random seed:	0
print interval:	10	plot file interval:	10
dump interval:	1000	density var (%):	0.0
job time (s):	3600.0	close time (s):	10.0

restart key: none
thermostat: DPD/MD-VV
barostat: none
electrostatics: none
surface: none
frozen bead wall: ☐ x ☐ y ☐ z
☐ global bonds ☐ ignore CONFIG ☐ override index ☐ OpenMP critical

14.28.1 Classes

- `class setdpdSys`

14.28.2 Function/Subroutine Documentation

setdpdSys

```
public setdpdSys()
```

Sets up Define DPD System window for DL_MESO GUI, laying out elements onto a grid.

14.29 setdpdSysEvt.java

Events for Define DPD System.

Implements actions for Define DPD System (including pop-up windows for setting thermostat, barostat, electrostatics and surface parameters): opening *CONTROL* file into GUI on clicking OPEN button, reading values from text fields, pull-down (combo) boxes and checkboxes, and saving information to CONTROL file on clicking SAVE button (and saving values from pop-up boxes into memory when relevant save buttons are clicked).

14.29.1 Classes

- `class setdpdSysEvt`

14.29.2 Function/Subroutine Documentation

setdpdSysEvt

```
public setdpdSysEvt(setdpdSys ini)
public setdpdSysEvt(setThermostat theini)
public setdpdSysEvt(setBarostat barini)
public setdpdSysEvt(setElectrostatic eleini)
public setdpdSysEvt(setSurface surini)
```

Sets currently open Define DPD System window and pop-up windows to check for user actions and implement actions in response.

Parameters

ini	setdpdSys	Instance of current Define DPD System window
theini	setThermostat	Instance of DPD Thermostat Properties pop-up window
barini	setBarostat	Instance of DPD Barostat Properties pop-up window
eleini	setElectrostatic	Instance of DPD Electrostatic Properties pop-up window
surini	setSurface	Instance of DPD Surface Properties pop-up window

setthermostatparameter

```
void setthermostatparameter(int ityp,
                            double at)
```

Launches DPD Thermostat Properties pop-up window with thermostat type and current value of parameter.

Parameters

ityp	int	Thermostat type (needs to be equal to 4 to open window for Stoyanov-Groot thermostat)
at	double	Stoyanov-Groot thermostat parameter α for pairwise Nosé-Hoover forces

setbarostatparameter

```
void setbarostatparameter(int btyp,
                          double ab,
                          double bb,
                          double cb,
                          boolean iso)
```

Launches DPD Barostat Properties pop-up window with barostat type, current values of parameters and isotropic switch.

Parameters

btyp	int	Barostat type
ab	double	First barostat parameter
bb	double	Second barostat parameter
cb	double	Third barostat parameter
iso	boolean	Switch indicating if using isotropic/semi-isotropic ensemble

setelectroparameter

```
void setelectroparameter(int etyp,
                         int pb,
                         int coe,
                         int bl,
                         int blrel,
                         double ae,
                         double be,
                         double ce,
                         int k1,
                         int k2,
                         int k3,
                         int mxs,
                         boolean ge)
```

Launches DPD Electrostatic Properties pop-up window with electrostatics/smearing type, switches, permittivity coefficient or Bjerrum length, smearing length etc.

Parameters

etyp	int	Selected electrostatics/smearing type (based on pull-down box in Define DPD System window)
pb	int	Switch for pull-down box selecting permittivity coefficient or Bjerrum length
coe	int	Switch for pull-down box selecting real-space convergence or relative error
bl	int	Switch for pull-down box indicating length or beta for charge smearing
blrel	int	Switch for pull-down box indicating how charge smearing length and beta are related
ae	double	Permittivity coefficient/Bjerrum length
be	double	Real-space convergence/relative error
ce	double	Smearing length/beta
k1	int	Maximum reciprocal vector (x-component)
k2	int	Maximum reciprocal vector (y-component)
k3	int	Maximum reciprocal vector (z-component)
mxs	int	B-spline interpolation order for SPME
ge	boolean	Switch for matching real-space convergence and charge smearing length for Gaussian smearing

setsurfaceparameter

```
void setsurfaceparameter(int sftyp,
                         int sx,
                         int sy,
                         int sz,
                         double woff,
                         double vx,
                         double vy,
                         double vz)
```

Launches DPD Surface Properties pop-up window with barostat type, current values of parameters and isotropic switch.

Parameters

sftyp	int	Surface type (0 = frozen bead walls, 1 = Lees-Edwards shearing boundaries, 2 = hard surfaces with specular reflection, 3 = hard surfaces with bounceback reflection)
sx	int	Flag indicating presence of surfaces orthogonal to x-axis
sy	int	Flag indicating presence of surfaces orthogonal to y-axis
sz	int	Flag indicating presence of surfaces orthogonal to z-axis
woff	double	Distance offset from box boundaries to apply surface reflections
vx	double	Vacuum gap for electrostatic interactions in x-direction
vy	double	Vacuum gap for electrostatic interactions in y-direction
vz	double	Vacuum gap for electrostatic interactions in z-direction

savethermo

```
void savethermo()
```

Reads value for thermostat parameter from text box in DPD Thermostat Properties pop-up window and stores it in memory before closing window.

savebaro

```
void savebaro()
```

Reads values for barostat parameters from text boxes and checkbox in DPD Barostat Properties pop-up window and stores them in memory before closing window.

saveelectro

```
void saveelectro()
```

Reads values for electrostatic parameters from text boxes, pull-down (combo) boxes and checkboxes in DPD Electrostatics Properties pop-up window and stores them in memory before closing window.

savesurface

```
void savesurface()
```

Reads values for surface parameters from text boxes and checkboxes in DPD Surface Properties pop-up window and stores them in memory before closing window.

savedpd

```
void savedpd()
```

Creates new *CONTROL* file (or overwrites existing one) after reading in values from text boxes, pull-down (combo) boxes and checkboxes in Define DPD System window, also using values previously obtained from pop-up windows.

opendpd

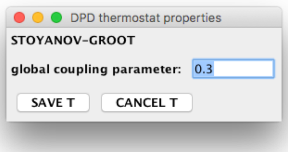
```
void opendpd()
```

Opens and reads *CONTROL* file, storing values required for pop-up windows (on thermostat, barostat, electrostatics and surfaces) in memory and writing or setting values in text boxes, pull-down (combo) boxes and checkboxes in Define DPD System window.

14.30 setThermostat.java

Pop-up window for DPD Thermostat Properties.

Sets up pop-up window for DPD Thermostat Properties (obtained from button in Define DPD System), setting labels for text boxes based on selected thermostat type (only used for Stoyanov-Groot thermostat) and filling text box with any already provided value from *CONTROL* file (*setdpdSysEvt.java* reads value from this box when saving CONTROL file).



STOYANOV-GROOT

global coupling parameter: 0.3

14.30.1 Classes

- `class setThermostat`

14.30.2 Function/Subroutine Documentation

setThermostat

```
public setThermostat(int it,
                     double at)
```

Sets up DPD Thermostat Properties pop-up window for DL_MESO GUI, laying out elements onto a grid and using supplied thermostat type and parameter to set up the text fields etc.

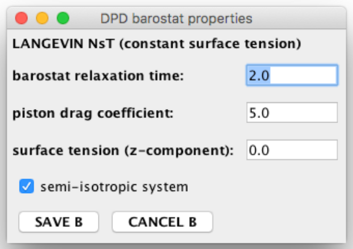
Parameters

it	int	Thermostat type (needs to be equal to 4 to open window for Stoyanov-Groot thermostat)
at	double	Stoyanov-Groot thermostat parameter α for pairwise Nosé-Hoover forces

14.31 setBarostat.java

Pop-up window for DPD Barostat Properties.

Sets up pop-up window for DPD Barostat Properties (obtained from button in Define DPD System), setting labels for text boxes based on selected barostat type and filling text boxes and isomer checkbox with any already provided values from CONTROL file (*setdpdSysEvt.java* reads values from these boxes when saving CONTROL file).



LANGEVIN NsT (constant surface tension)

barostat relaxation time: 2.0

piston drag coefficient: 5.0

surface tension (z-component): 0.0

☑ semi-isotropic system

14.31.1 Classes

- `class setBarostat`

14.31.2 Function/Subroutine Documentation

setBarostat

```
public setBarostat(int bt,
                   double ab,
                   double bb,
                   double cb,
                   boolean iso)
```

Sets up DPD Barostat Properties pop-up window for DL_MESO GUI, laying out elements onto a grid and using supplied barostat type, parameters and isotropic switch to determine its contents.

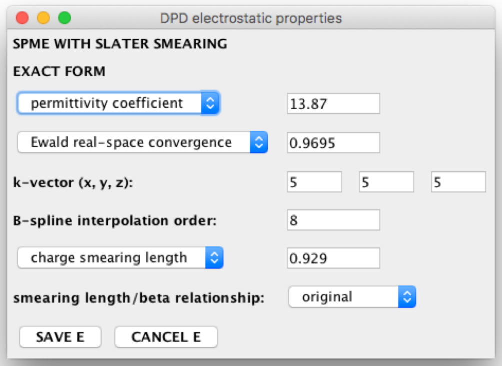
Parameters

bt	int	Barostat type
aa	double	First barostat parameter
bb	double	Second barostat parameter
cc	double	Third barostat parameter
iso	boolean	Switch indicating if using isotropic/semi-isotropic ensemble

14.32 setElectrostatic.java

Pop-up window for DPD Electrostatic Properties.

Sets up pop-up window for DPD Electrostatic Properties (obtained from button in Define DPD System), setting labels for text boxes (including pull-down combo boxes to select permittivity/Bjerrum length, reciprocal vector extent/relative error, charge smearing length/beta parameter, relationship between charge smearing length and beta parameter) based on selected electrostatic type and filling text boxes with any already provided values from *CONTROL* file (*setdpdSysEvt.java* reads values from these boxes when saving CONTROL file).



SPME WITH SLATER SMEARING

EXACT FORM

permittivity coefficient 13.87

Ewald real-space convergence 0.9695

k-vector (x, y, z): 5 5 5

B-spline interpolation order: 8

charge smearing length 0.929

smearing length/beta relationship: original

14.32.1 Classes

- `class setElectrostatic`

14.32.2 Function/Subroutine Documentation

setBound

```
public setElectrostatic(int et,
                        int pb,
                        int coe,
                        int bl,
                        int blrel,
                        double ae,
                        double be,
                        double ce,
                        int k1,
                        int k2,
                        int k3,
                        int mxs,
                        boolean ge)
```

Sets up DPD Electrostatic Properties pop-up window for DL_MESO GUI, laying out elements onto a grid and using supplied electrostatics/smearing type, parameters, selections for relationships between parameters etc. to set up the text fields, pull-down (combo) boxes etc.

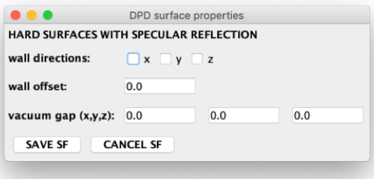
Parameters

et	int	Selected electrostatics/smearing type (based on pull-down box in Define DPD System window)
pb	int	Switch for pull-down box selecting permittivity coefficient or Bjerrum length
coe	int	Switch for pull-down box selecting real-space convergence or relative error
bl	int	Switch for pull-down box indicating length or beta for charge smearing
blrel	int	Switch for pull-down box indicating how charge smearing length and beta are related
ae	double	Permittivity coefficient/Bjerrum length
be	double	Real-space convergence/relative error
ce	double	Smearing length/beta
k1	int	Maximum reciprocal vector (x-component)
k2	int	Maximum reciprocal vector (y-component)
k3	int	Maximum reciprocal vector (z-component)
mxs	int	B-spline interpolation order for SPME
ge	boolean	Switch for matching real-space convergence and charge smearing length for Gaussian smearing

14.33 setSurface.java

Pop-up window for DPD Surface Properties.

Sets up pop-up window for DPD Surface Properties (obtained from button in Define DPD System), setting labels for text boxes and checkboxes based on selected surface type, and filling text boxes and checkboxes with any already provided values from *CONTROL* file (*setdpdSysEvt.java* reads values from these boxes when saving CONTROL file).


DPD surface properties

HARD SURFACES WITH SPECULAR REFLECTION

wall directions: x y z

wall offset: 0.0

vacuum gap (x,y,z): 0.0 0.0 0.0

14.33.1 Classes

- `class setSurface`

14.33.2 Function/Subroutine Documentation

setSurface

```
public setSurface(int srft,
                  int sx,
                  int sy,
                  int sz,
                  double walloff,
                  double vgpx,
                  double vgpy,
                  double vgpz)
```

Sets up DPD Surface Properties pop-up window for DL_MESO GUI, laying out elements onto a grid and using supplied surface type, dimensions with surfaces, distance offset for reflections and electrostatics vacuum gap to set up the text fields etc.

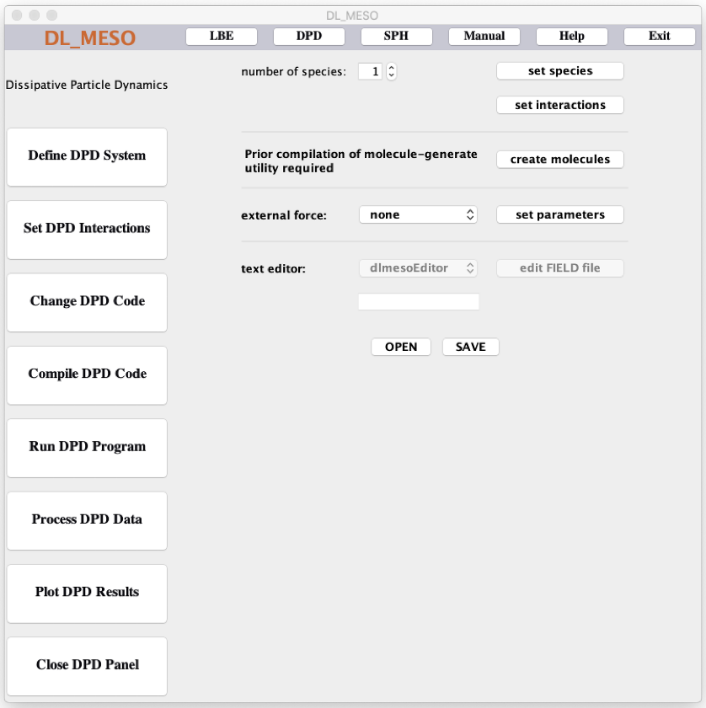
Parameters

srft	int	Surface type (0 = frozen bead walls, 1 = Lees-Edwards shearing boundaries, 2 = hard surfaces with specular reflection, 3 = hard surfaces with bounceback reflection)
sx	int	Flag indicating presence of surfaces orthogonal to x-axis
sy	int	Flag indicating presence of surfaces orthogonal to y-axis
sz	int	Flag indicating presence of surfaces orthogonal to z-axis
walloff	double	Distance offset from box boundaries to apply surface reflections
vgpx	double	Vacuum gap for electrostatic interactions in x-direction
vgpy	double	Vacuum gap for electrostatic interactions in y-direction
vgpz	double	Vacuum gap for electrostatic interactions in z-direction

14.34 setdpdInteract.java

Window for Set DPD Interactions.

Sets up window for Set DPD Interactions, including a spinner for the number of particle species, buttons to open pop-up windows for particle species, interactions and external fields, to launch utility to create molecular data, to open, edit and save *FIELD* file (needed for DL_MESO_DPD), pull-down (combo) boxes to select external fields and editor for modifying FIELD file.



number of species: 1

Prior compilation of molecule-generate utility required

external force: none

text editor: dlmesoEditor

14.34.1 Classes

- `class setdpdInteract`

14.34.2 Function/Subroutine Documentation

setdpdInteract

```
public setdpdInteract()
```

Sets up Set DPD Interactions window for DL_MESO GUI, laying out elements onto a grid.

14.35 setdpdInteractEvt.java

Events for Set DPD Interactions.

Implements actions for Set DPD Interactions (including pop-up windows for setting species, interactions and external field parameters): opening *FIELD* file into GUI, running molecule data generation utility, editing FIELD file and saving information to FIELD file on button clicks, saving species, interaction and external field paramters in memory on button clicks in pop-up windows, selecting options in drop-down (combo) boxes and reading values from text fields and drop-down boxes.

14.35.1 Classes

- `class setdpdInteractEvt`

14.35.2 Function/Subroutine Documentation

setdpdInteractEvt

```
public setdpdInteractEvt(setdpdInteract setinteractions)
public setdpdInteractEvt(setSpecies speini)
public setdpdInteractEvt(setInteraction interactdpd)
public setdpdInteractEvt(setExternal externdpd)
```

Sets currently open Set DPD Interactions window and pop-up windows to check for user actions and implement actions in response.

Parameters

setinteractions	setdpdInteract	Instance of current Change DPD Code window
speini	setSpecies	Instance of Set DPD Species Properties pop-up window
interactdpd	setInteraction	Instance of Set DPD Interaction Properties pop-up window
externdpd	setExternal	Instance of Set DPD External Field Properties pop-up window

setspecies

```
void setspecies(int totspe,
                String[] names,
                double[] masses,
                double[] charges,
                int[] numbers,
                int[] frozen)
```

Launches Set DPD Species Properties pop-up window with current values of species numbers, names, masses, charges, unbonded populations and frozen flags.

Parameters

totspe	int	Number of particle species
names	String[]	Names of particle species
masses	double[]	Masses of particle species
charges	double[]	Charges (valencies) of particle species
numbers	int[]	Population of particle species excluding molecules
frozen	int[]	Flags indicating if particles species are frozen or not

savespecies

```
void savespecies()
```

Reads values for species names, masses, charges, unbonded populations and frozen flags from text boxes in Set DPD Species Properties pop-up window and stores them in memory before closing window.

setinteract

```
void setinteract(int totspe,
                 String[] names,
                 int ktyp,
                 double aa,
                 double bb,
                 double cc,
                 double dd,
                 double ee,
                 double gamm,
                 double dist,
                 int sftyp,
                 int srfktyp,
                 double srfaa,
                 double srfdist,
                 boolean froz,
                 int wallspe,
                 double wallrho,
                 double wallthick)
```

Launches DPD Interaction Properties pop-up window with number and names of species, and interaction parameter values for the first species pair (between two particles of the first particle species).

Parameters

totspe	int	Number of particle species
names	String[]	Names of particle species (as previously set in DPD Set Species pop-up window
ktyp	int	Interaction type for specified species pair
aa	double	Interaction energy parameter A_{ij}
bb	double	Interaction energy parameter B_{ij}
cc	double	Interaction energy parameter C_{ij}
dd	double	Interaction energy parameter D_{ij}
ee	double	Interaction energy parameter E_{ij}
gamm	double	Dissipative force parameter γ_{ij} or collision frequency Γ_{ij}
dist	double	Distance parameter σ_{ij} or interaction cutoff distance $r_{c,ij}$
sftyp	int	Surface type (selecting item for pull-down box)
srfktyp	int	Surface/wall interaction type
srfaa	double	Surface interaction energy parameter $A_{wall,i}$
srfdist	double	Surface distance parameter $\sigma_{wall,i}$ or $z_{c,i}$
froz	boolean	Switch for frozen bead walls
wallspe	int	Species number for frozen bead walls
wallrho	double	Particle density for frozen bead walls
wallthick	double	Thickness of frozen bead walls

setexternal

```
void setexternal(int exttype,
                 double ex,
                 double ey,
                 double ez)
```

Launches DPD External Field Properties pop-up window with the external field type and value.

Parameters

et	int	Selected external field type (based on pull-down box in Set DPD Interactions window)
ex	double	External field value (x-component)
ey	double	External field value (y-component)
ez	double	External field value (z-component)

saveexternal

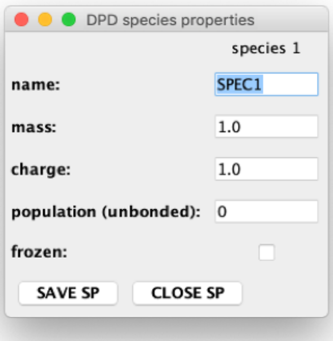
```
void saveexternal()
```

Reads values for external field from text boxes in Set DPD External Field Properties pop-up window and stores them in memory before closing window.

14.36 setSpecies.java

Pop-up window for DPD Species Properties.

Sets up pop-up window for DPD Species Properties (obtained from button in Set DPD Interactions after specifying number of species), setting labels, adding text boxes for species names, masses, charges and unbonded number (i.e. not included in molecules) and checkboxes to indicate if species is frozen or not - all either using default values or those previously read from *FIELD* file - and buttons to save values in memory and close window (*setdpdInteractEvt.java* uses values stored in memory when saving FIELD file).



14.36.1 Classes

- `class setSpecies`

14.36.2 Function/Subroutine Documentation

setSpecies

```
public setSpecies(int totsp,
                  String[] name,
                  double[] mass,
                  double[] charge,
                  int[] pop,
                  int[] frozen)
```

Sets up DPD Species Properties pop-up window for DL_MESO GUI, laying out elements onto a grid and using supplied number, names, masses, charges, unbonded populations and frozen parameters of particle species to set up the text fields etc.

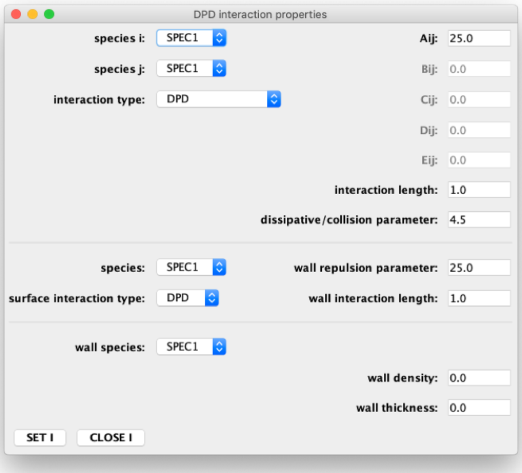
Parameters

totsp	int	Number of particle species
name	String[]	Names of particle species
mass	double[]	Masses of particle species
charge	double[]	Charges (valencies) of particle species
pop	int[]	Population of particle species excluding molecules
frozen	int[]	Flags indicating if particles species are frozen or not

14.37 setInteraction.java

Pop-up window for DPD Interaction Properties.

Sets up pop-up window for DPD Interaction Properties (obtained from button in Set DPD Interactions), setting labels, adding pull-down (combo) boxes with available particle species and interaction types, filling text boxes with current parameter values (if opening for first time, either default values or those previously read from *FIELD* file), and buttons to set values for each species pair in memory and to close window (*setdpdInteractEvt.java* uses values set in memory when saving FIELD file).



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

species: SPEC1
surface interaction type: DPD
wall repulsion parameter: 25.0
wall interaction length: 1.0

wall species: SPEC1
wall density: 0.0
wall thickness: 0.0

14.37.1 Classes

- `class setInteraction`

14.37.2 Function/Subroutine Documentation

setInteraction

```
public setInteraction(int totsp,
                      String[] name,
                      int ktype,
                      double aaparam,
                      double bbparam,
                      double ccparam,
                      double ddparam,
                      double eeparam,
                      double gammaparam,
                      double distparam,
                      int srftyp,
                      int srfktype,
                      double srfaaparam,
                      double srfdistparam,
                      boolean wall,
                      int wallspe,
                      double wallrho,
                      double wallthic)
```

Sets up DPD Interaction Properties pop-up window for DL_MESO GUI, laying out elements onto a grid and using supplied number and names of particle species, interaction type and parameters for a specified species pair, surface type and parameters etc. to set up the text fields, pull-down (combo) boxes etc.

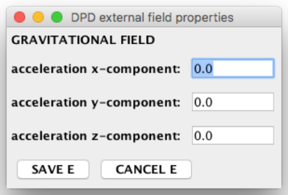
Parameters

totsp	int	Number of particle species
name	String[]	Names of particle species (as previously set in DPD Set Species pop-up window
ktype	int	Interaction type for specified species pair
aaparam	double	Interaction energy parameter A_{ij}
bbparam	double	Interaction energy parameter B_{ij}
ccparam	double	Interaction energy parameter C_{ij}
ddparam	double	Interaction energy parameter D_{ij}
eeparam	double	Interaction energy parameter E_{ij}
gammaparam	double	Dissipative force parameter γ_{ij} or collision frequency Γ_{ij}
distparam	double	Distance parameter σ_{ij} or interaction cutoff distance $r_{c,ij}$
srftyp	int	Surface type (selecting item for pull-down box)
srfktyp	int	Surface/wall interaction type
srfaaparam	double	Surface interaction energy parameter $A_{wall,i}$
srfdistparam	double	Surface distance parameter $\sigma_{wall,i}$ or $z_{c,i}$
wall	boolean	Switch for frozen bead walls
wallspe	int	Species number for frozen bead walls
wallrho	double	Particle density for frozen bead walls
wallthic	double	Thickness of frozen bead walls

14.38 setExternal.java

Pop-up window for DPD External Field Properties.

Sets up pop-up window for DPD External Field Properties (obtained from button in Set DPD Interactions), setting labels based on selected external field type, filling text boxes with current parameter values (if opening for first time, either default values or those previously read from FIELD file), and buttons to save to memory and to close window (*setdpdInteractEvt.java* uses values set in memory when saving FIELD file).



acceleration x-component: 0.0

acceleration y-component: 0.0

acceleration z-component: 0.0

14.38.1 Classes

- `class setExternal`

14.38.2 Function/Subroutine Documentation

setExternal

```
public setExternal(int et,
                   double ex,
                   double ey,
                   double ez)
```

Sets up DPD External Field Properties pop-up window for DL_MESO GUI, laying out elements onto a grid and using supplied external field type and values to set up the text fields etc.

Parameters

et	int	Selected external field type (based on pull-down box in Set DPD Interactions window)
ex	double	External field value (x-component)
ey	double	External field value (y-component)
ez	double	External field value (z-component)

14.39 changedpdcode.java

Window for Change DPD Code.

Sets up window for Change DPD Code, including pull-down (combo) boxes to select text editor and DL_MESO_DPD code module file, text fields for editors and files not included in pull-down boxes, and button to edit selected file with editor.

14.39.1 Classes

- `class changedpdcode`

14.39.2 Function/Subroutine Documentation

changedpdcode

```
public changedpdcode()
```

Sets up Change DPD Code window for DL_MESO GUI, laying out elements onto a grid.

14.40 changedpdEvt.java

14.40.1 Summary

Events for Change DPD Code.

Implements actions for Change DPD Code, including selecting text editor and DL_MESO_DPD code module file in pull-down (combo) boxes and text boxes, and pressing EDIT button.

14.40.2 Classes

- `class changedpdEvt`

14.40.3 Function/Subroutine Documentation

changeddpdEvt

```
public changedpdEvt(changedpdcode in)
```

Sets currently open Change DPD Code window as window to check for user actions and implement actions in response.

Parameters

in	changedpdcode	Instance of current Change DPD Code window

14.41 compiledpd.java

Window for Compile DPD Code.

Sets up window for Compile DPD Code, including pull-down (combo) boxes to select Fortran compiler, DL_MESO_DPD code version and FFT solver, text fields for compilers not included in pull-down box and compiler flags, and buttons to create Makefile and compile code.

14.41.1 Classes

- `class compiledpd`

14.41.2 Function/Subroutine Documentation

compiledpd

```
public compiledpd()
```

Sets up Compile DPD Code window for DL_MESO GUI, laying out elements onto a grid.

14.42 compiledpdEvt.java

Events for Compile DPD Code.

Implements actions for Compile DPD Code, including selecting Fortran compiler and compiler flags, DL_MESO_DPD code version and FFT solver, and pressing buttons to create Makefile and compile code.

14.42.1 Classes

- `class compiledpdEvt`

14.42.2 Function/Subroutine Documentation

compiledpdEvt

```
public compiledpdEvt(compiledpd in)
```

Sets currently open Compile DPD Code window as window to check for user actions and implement actions in response.

Parameters

in	compiledpd	Instance of current Compile DPD Code window

makefilemake

```
public void makefilemake()
```

Creates Makefile for compiling DL_MESO_DPD based on selected inputs in Compile DPD Code window.

14.43 rubdpd.java

Window for Run DPD Program.

Sets up window for Run DPD Program, including pull-down (combo) box and text field to select command required to launch DL_MESO_DPD, and button to Run DPD code.

14.43.1 Classes

- `class rubdpd`

14.43.2 Function/Subroutine Documentation

rubdpd

```
public rubdpd()
```

Sets up Run DPD Program window for DL_MESO GUI, laying out elements onto a grid.

14.44 rubdpdEvt.java

Events for Run DPD Program.

Implements actions for Run DPD Program, including selecting command for launching DL_MESO_DPD, and pressing Run DPD button.

14.44.1 Classes

- `class rubdpdEvt`

14.44.2 Function/Subroutine Documentation

rubdpdEvt

```
public rubdpdEvt(rubdpd in)
```

Sets currently open Run DPD Program window as window to check for user actions and implement actions in response.

Parameters

in	rubdpd	Instance of current Run DPD Program window

14.45 gatherdpd.java

Window for Process DPD Data.

Sets up window for Process DPD Data, including pull-down (combo) boxes to select property to calculate (utility to launch), available species and molecule types (taken from available FIELD file) and data level option for creating CONFIG files, text boxes for user-inputted numbers, a checkbox for utility-dependent option and a button to Process data.

14.45.1 Classes

- `class gatherdpd`

14.45.2 Function/Subroutine Documentation

gatherdpd

```
public gatherdpd()
```

Sets up Process DPD Data window for DL_MESO GUI, laying out elements onto a grid, and reading *FIELD* file to obtain species and molecule names for pull-down (combo) boxes.

14.46 gatherdpdEvt.java

Events for Process DPD Data.

Implements actions for Process DPD Data, including selecting utility from pull-down (combo) boxes, applying utility-dependent labels and activating other pull-down boxes based on this choice, and running utility when Process data button is clicked, taking in selections and user-specified values to provide command-line options.

14.46.1 Classes

- `class gatherdpdEvt`

14.46.2 Function/Subroutine Documentation

gatherdpdEvt

```
public gatherdpdEvt(gatherdpd in)
```

Sets currently open Process DPD Data window as window to check for user actions and implement actions in response.

Parameters

in	gatherdpd	Instance of current Process DPD Data window

14.47 plotdpd.java

Window for Plot DPD Results.

Sets up window for Plot DPD Results, including pull-down (combo) box to select pre-installed software package to launch, text field for other packages not included in pull-down box, check box to run program in terminal window, and button to launch package for plotting results from DPD calculation.

14.47.1 Classes

- `class plotdpd`

14.47.2 Function/Subroutine Documentation

plotdpd

```
public plotdpd()
```

Sets up Plot DPD Results window for DL_MESO GUI, laying out elements onto a grid.

14.48 plotdpdEvt.java

Events for Plot DPD Results.

Implements actions for Plot DPD Results, including selecting plotting software package in pull-down (combo) and text box, and pressing Plot data button.

14.48.1 Classes

- `class plotdpdEvt`

14.48.2 Function/Subroutine Documentation

plotdpdEvt

```
public plotdpdEvt(plotdpd in)
```

Sets currently open Plot DPD Results window as window to check for user actions and implement actions in response.

Parameters

in	plotdpd	Instance of current Plot DPD Results window

14.49 dlmesoeditor.java

Window for DL_MESO GUI Text Editor.

Sets up pop-up window for DL_MESO GUI internal text editor with scrollable text area for file contents and buttons to save file and exit editor (close window).

14.49.1 Classes

- `class dlmesoeditor`

14.49.2 Function/Subroutine Documentation

dlmesoeditor

```
public dlmesoeditor(String str1)
```

Sets up DL_MESO GUI Text Editor window (including area for text with scrollbars, save and exit buttons) and opens specified file.

Parameters

in	str1	Name of file to open in DL_MESO GUI text editor

14.50 dlmesoeditEvt.java

Events for DL_MESO GUI Text Editor.

Implements actions for DL_MESO GUI internal text editor, including pressing buttons to save file and exit (to close window).

14.50.1 Classes

- `class dlmesoeditEvt`

14.50.2 Function/Subroutine Documentation

dlmesoeditEvt

```
public dlmesoeditEvt(dlmesoeditor in)
```

Sets currently open DL_MESO GUI Text Editor as window to check for user actions and implement actions in response.

Parameters

in	dlmesoeditor	Instance of current DL_MESO GUI Text Editor window

14.51 msgPanel.java

Message panel.

Provides pop-up window to provide error, warning and information messages to the user in reaction to events or selected values for supplied properties.

14.51.1 Classes

- `class msgPanel`

14.51.2 Function/Subroutine Documentation

msgPanel

```
public msgPanel(String phrase)
```

Opens pop-up window with supplied message and close button.

Parameters

phrase	String	Message to display in pop-up window

14.52 msgPanelEvt.java

Events for message panel.

Implements actions for pop-up message window, particularly pressing button to close window.

14.52.1 Classes

- `class msgPanelEvt`

14.52.2 Function/Subroutine Documentation

msgPanelEvt

```
public msgPanelEvt(msgPanel in)
```

Sets currently open message window as window to check for user actions and implement actions in response.

Parameters

in	msgPanel	Instance of current message pop-up window

14.53 Additional subroutines/functions available in multiple classes

14.53.1 addItem

```
private void addItem(JPanel p,
                     JComponent c,
                     int x,
                     int y,
                     int width,
                     int height,
                     int align)
```

Adds item to a window using GridBag at a given position within a grid with a width, height and alignment.

Parameters

p	JPanel	Window (panel) to which item is to be added
c	JComponent	Component (item) to add to window
x	int	x-coordinate of grid location (column number)
y	int	y-coordinate of grid location (row number)
width	int	Width of item in grid blocks (enables span)
height	int	Height of item in grid blocks (enables span)
align	int	Alignment of item (given as GridBagConstraints)

14.53.2 addItemFill

```
private void addItemFill(JPanel p,
                         JComponent c,
                         int x,
                         int y,
                         int width,
                         int height,
                         int align,
                         int fill)
private void addItemFill(JPanel p,
                         JComponent c,
                         int x,
                         int y,
                         int width,
                         int height,
                         double weightx,
                         double weighty,
                         int align,
                         int fill)
```

Adds item to a window using GridBag at a given position within a grid with a width, height and alignment, filling the grid block horizontally or vertically and optionally adding space horizontally and vertically.

Parameters

p	JPanel	Window (panel) to which item is to be added
c	JComponent	Component (item) to add to window
x	int	x-coordinate of grid location (column number)
y	int	y-coordinate of grid location (row number)
width	int	Width of item in grid blocks (enables span)
height	int	Height of item in grid blocks (enables span)
weightx	double	Additional horizontal space
weighty	double	Additional vertical space
align	int	Alignment of item (given as GridBagConstraints)
fill	int	Item fill (given as GridBagConstraints)

14.53.3 addItemPadding

```
private void addItemPadding(JPanel p,
                            JComponent c,
                            int x,
                            int y,
                            int width,
                            int height,
                            int align,
                            int padding)
```

Adds item to a window using GridBag at a given position within a grid with a width, height and alignment, filling the grid block horizontally or vertically and adding a specified amount of padding space around the item (constant in all directions).

Parameters

p	JPanel	Window (panel) to which item is to be added
c	JComponent	Component (item) to add to window
x	int	x-coordinate of grid location (column number)
y	int	y-coordinate of grid location (row number)
width	int	Width of item in grid blocks (enables span)
height	int	Height of item in grid blocks (enables span)
align	int	Alignment of item (given as GridBagConstraints)
padding	int	Amount of padding to add (in pixels)

14.53.4 addItemSpacing

```
private void addItemSpacing(JPanel p,
                            JComponent c,
                            int x,
                            int y,
                            int width,
                            int height,
                            int align,
                            int inset1,
                            int inset2)
```

Adds item to a window using GridBag at a given position within a grid with a width, height and alignment, filling the grid block horizontally or vertically and adding a specified amount of padding space above and below the item.

Parameters

p	JPanel	Window (panel) to which item is to be added
c	JComponent	Component (item) to add to window
x	int	x-coordinate of grid location (column number)
y	int	y-coordinate of grid location (row number)
width	int	Width of item in grid blocks (enables span)
height	int	Height of item in grid blocks (enables span)
align	int	Alignment of item (given as GridBagConstraints)
inset1	int	Amount of padding to add above item (in pixels)
inset2	int	Amount of padding to add below item (in pixels)

14.53.5 dledit

```
void dledit(String str)
```

Launches DL_MESO GUI Text Editor (*dlmesoeditor.java*) and opens specified text file.

Parameters

str	String	Name of file to open in DL_MESO GUI Text Editor

14.53.6 ierr

```
void ierr(String errinfo)
```

Opens message panel (*msgPanel.java*) with specified error message for user.

Parameters

errinfo	String	Error message to display to user in panel

14.53.7 isMac

```
public static boolean isMac()
```

Returns true if current operating system is macOS (Mac OS X).

14.53.8 isUnix

```
public static boolean isUnix()
```

Returns true if current operating system is Unix, Linux or similar.

14.53.9 isWindows

```
public static boolean isWindows()
```

Returns true if current operating system is Windows.

CHAPTER

FIFTEEN

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 and analysis. 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.

15.1 DL_MESO_LBE

All utilities for the LBE code can be run at the command line with optional arguments, e.g.

```
utility.exe [arguments]
```

for Windows machines, or

```
./utility.exe [arguments]
```

for machines running Unix, Linux or Mac (Mac OS X or macOS) operating systems. One command line argument all LBE utilities have in common is `-h` (for help), which will give a brief description of the utility and its available command line arguments before quitting. All utilities that work on input or output files should be run in the same directory as those files, which should have the default names for those types (*lbin.sys*, *lbout*.vts*, *lbout*.q* etc.).

15.1.1 lbeinitcreate.cpp

lbeinitcreate.cpp 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.

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

[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*.

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 a single fluid is specified in the *lbin.sys* file with different initial and constant densities, the utility will ask for the number of drops to be added to the system: for each drop, the user will need to specify its radius, where its centre is located on the lattice grid and its density. 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, where its centre is located on the lattice grid and its density. (Note that it is possible for a drop to extend beyond the grid boundaries if periodic boundaries are in use, but the drop centre must be within those boundaries.)

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.

15.1.2 lbeplot3dgather.cpp

lbeplot3dgather.cpp 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 visualisation 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.

15.1.3 lbevtkgather.cpp

lbevtkgather.cpp 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 visualisation 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 writing processors used for the simulation (i.e. the number of files per frame) and the extents of each piece.

The executable for this utility can be run with any of the following command line arguments:

- `-a`
  Create linking files for output files that contain all properties from the LBE simulation (fluid densities, fluid mass fractions, solute concentrations and temperatures)

- `-d`
  Create linking files for output files that contain a single fluid density
- `-f`
  Create linking files for output files that contain a single fluid mass fraction
- `-c`
  Create linking files for output files that contain a single solute concentration
- `-t`
  Create linking files for output files that just contain temperatures

If no command-line argument is specified, the utility will assume all properties are contained in the output files. (If the GUI is used, the level of data to link together can be selected using the pulldown list in the **Gather LBE Data** panel.) 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.

15.1.4 lbedumpvtk.cpp

lbedumpvtk.cpp is a utility written in C++ to create an XML-formatted structured VTK file (lbdump.vts) from the *lbout.dump* restart file generated by DL_MESO_LBE. The resulting file can be used to visualise the system at the point when the restart data was written (e.g. when the simulation was terminated) and verify the simulation is proceeding as expected, even when no other output files are generated.

If `c++` is the command for the available C++ compiler, the executable `dump_to_vtk.exe` can be produced by typing

```
c++ -o dump_to_vtk.exe lbedumpvtk.cpp
```

and either run at the command line or via the GUI under **Gather LBE Data**. No other files are needed to create the VTK file, as the *lbout.dump* file includes all required data for visualisation.

15.1.5 lbedumpinit.cpp

lbedumpinit.cpp is a utility written in C++ to create an initialisation file *lbin.init* from the *lbout.dump* restart file generated by DL_MESO_LBE. The resulting file can be used to start a new simulation from the state of a previous one.

If `c++` is the command for the available C++ compiler, the executable `dump_to_init.exe` can be produced by typing

```
c++ -o dump_to_init.exe lbedumpinit.cpp
```

and either run at the command line or via the GUI under **Gather LBE Data**. No other files are needed to create the *lbin.init* file, as the *lbout.dump* file includes all required data for determining the state of the simulation (i.e. macroscopic properties at each grid point).

15.2 DL_MESO_DPD

All utilities for the DPD code can be run at the command line with optional arguments, e.g.

```
utility.exe [arguments]
```

for Windows machines, or

```
./utility.exe [arguments]
```

for machines running Unix, Linux or Mac (Mac OS X or macOS) operating systems. One command line argument all DPD utilities have in common is `-h` (for help), which will give a brief description of the utility and its available command line arguments before quitting. All utilities that work on input or output files should be run in the same directory as those files, which should have the default names for those types (*CONTROL*, *FIELD*, *export*, *HISTORY* etc.).

15.2.1 convert-input.cpp

convert-input.cpp 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 2.7 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 any (or all) of these optional command line arguments:

- `-c [CONTROL]`
  Uses a CONTROL file as input called `[CONTROL]`
- `-f [FIELD]`
  Uses a FIELD file as input called `[FIELD]`
- `-m [MOLECULE]`
  Uses a MOLECULE file as input called `[MOLECULE]`
- `-v`
  Verbose option: writes out data read from input files on to the screen or standard output

Note that if the default names are used for input, the old CONTROL and FIELD files are renamed after being read (to CONTROL.old and FIELD.old) 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.)

15.2.2 molecule-generate.cpp

molecule-generate.cpp 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).

This utility can be run with any (or all) of these optional command line arguments:

- `-p`
  Write data to a separate file (molecule) instead of a *FIELD* file: this is used by the GUI to insert molecular data into the *FIELD* file it generates
- `-s n [SPEC1] [SPEC2] ... [SPECn]`
  Define n particle species and provide their names (`[SPEC1]`, `[SPEC2]` to `[SPECn]`)

If a *FIELD* file exists in the same directory as the executable with species data and the `-s` command-line option is not used, 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 (if the separate file is not specified). 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 either be appended to the *FIELD* file in the correct format with positions for the beads relative to each molecule's centre of mass, or the data will be written to the separate *molecule* file. Note that the *FIELD* file will not be quite complete after running this utility: data for unbonded interactions and external force fields may be required (if it is created from scratch using the utility) and a `close` directive will be required at the end.

15.2.3 export_config.F90

export_config.F90 is a utility written in Fortran to produce a configuration file (*CONFIG*) from DL_MESO_DPD restart files (*export*), which can be used as a starting point for new simulations. Since a limited amount of data is included in restart files, the *FIELD* file for the simulation is also needed to provide some additional information.

The source code for this utility can be used for *export* files created by both the serial and parallel versions of DL_MESO_DPD. If the utility is to be run on a different machine to the one used for DPD calculations, care should be taken to ensure the utility can read files with the same endianness by using a compiler flag to force big or little endianness.

If the available Fortran compiler is invoked by the command `f90`, the executable `export_config.exe` can be produced by typing

```
f90 -o export_config.exe export_config.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.

The executable for this utility can be run with any of the following command line arguments:

- `-k` i
  Set *CONFIG* file key, *levcfg*, to i (0 = positions only, 1 = positions and velocities, 2 = positions, velocities and forces)
- `-s`
  Write particles to *CONFIG* file after sorting them in order by particle number
- `-u`
  Write particles to *CONFIG* file without sorting them

If no command-line argument is given, the utility will ask the user to type in the *CONFIG* file key and assume the particles are not sorted before writing to the file. (Note that particles in *export* files produced by the serial version of DL_MESO_DPD will be automatically in numerical order.)

15.2.4 export_image_vtf.F90

export_image_vtf.F90 is a utility written in Fortran to produce a VTF format trajectory file (export.vtf) from DL_MESO_DPD restart files (*export*) that can be visualized in VMD [60] 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 can be used for *export* files created by both the serial and parallel versions of DL_MESO_DPD: if the utility is to be run on a different machine to the one used for DPD calculations, care should be taken to ensure the utility can read files with the same endianness by using a compiler flag to force big or little endianness.

If the available Fortran compiler is invoked by the command `f90`, the executable `export_image_vtf.exe` can be produced by typing

```
f90 -o export_image_vtf.exe export_image_vtf.F90
```

and either run at the command line or by using the GUI in **Process DPD Data**. The optional commend-line argument `-s` can be used to sort the particles into order of global particle index. So long as both the *export* and *FIELD* files are available in the same directory, the utility will produce the VTF trajectory file without any prompting.

15.2.5 export_image_xml.F90

export_image_xml.F90 is a utility written in Fortran to produce a trajectory file in GALAMOST [157] XML format (export.xml) from DL_MESO_DPD restart files (*export*) that can be visualized in OVITO [132] 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 can be used for *export* filez created by both the serial and parallel versions of DL_MESO_DPD: if the utility is to be run on a different machine to the one used for DPD calculations, care should be taken to ensure the utility can read files with the same endianness by using a compiler flag to force big or little endianness.

If the available Fortran compiler is invoked by the command `f90`, the executable `export_image_xml.exe` can be produced by typing

```
f90 -o export_image.exe export_image_xml.f90
```

and either run at the command line or by using the GUI in **Process DPD Data**. So long as both the *export* and *FIELD* files are available in the same directory, the utility will produce the XML trajectory file without any prompting.

15.2.6 history_config.F90

history_config.F90 is a utility written in Fortran to take DL_MESO_DPD trajectory output data files (*HISTORY*) and produce a configuration file (*CONFIG*) from them, which can be used as a starting point for new simulations.

The source code for this utility reads *HISTORY* files generated by both the serial and parallel versions of DL_MESO_DPD respectively: because *HISTORY* files include a number in their headers as an endianness check, the utility should be able to read these files created on different machines. The utility outputs data for every particle for the selected trajectory frame: the maximum level of data (particle positions, velocities and forces) is fixed by the *HISTORY* file but the user can choose up to that particular level to write to the *CONFIG* file.

If `f90` is the command for the available Fortran compiler, the executable `history_config.exe` can be produced by typing

```
f90 -o history_config.exe history_config.F90
```

and either run at the command line or via the GUI in **Process DPD Data**.

The executable for this utility can be run with any of the following command line arguments:

- `-k` i

  Set *CONFIG* file key, *levcfg*, to i (0 = positions only, 1 = positions and velocities, 2 = positions, velocities and forces)
- `-f` i

  Use trajectory data in frame i of *HISTORY* file for configuration in *CONFIG* file
- `-l` i

  Use last frame of trajectory data in *HISTORY* file for configuration in *CONFIG* file
- `-s`

  Write particles to *CONFIG* file after sorting them in order by particle number
- `-u`

  Write particles to *CONFIG* file without sorting them

If no command-line argument is given or the *CONFIG* file key or trajectory frame number are out of range for the given *HISTORY* file, the utility will ask the user to type in valid numbers for the *CONFIG* file key (if more than positions are available) and trajectory frame number (unless the last one is selected at the command line), as well as assume that the particles are not sorted when writing to the *CONFIG* file.

15.2.7 traject_vtf.F90

The Fortran utility *traject_vtf.F90* 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 using VMD [60], such that snapshots at the recorded timesteps and animations can be produced. An option exists to produce separate VTF format trajectory files for unbonded particles (traject_bead.vtf) and bonded particles in molecules (traject_mole.vtf), and another option exists to write the structure and coordinates into separate files (traject.vsf and traject.vcf respectively).

The source code for this utility reads *HISTORY* files generated by the serial and parallel versions of DL_MESO_DPD: since *HISTORY* files include a number in their headers as an endianness check, the utility should be able to read these files created on different machines. The utility outputs every particle for all recorded timesteps, including bond data for particles in molecules represented as residues.

If `f90` is the command for the available Fortran compiler, the executable `traject_vtf.exe` can be produced by typing

```
f90 -o traject_vtf.exe traject_vtf.F90
```

and either run at the command line or via the GUI in **Process DPD Data**.

The executable for this utility can be run with the following optional command line arguments:

- `-b`

  Write trajectory data to separate files for bonded and unbonded particles
- `-s`

  Write particles to traject.vtf file after sorting them in order by particle number
- `-u`

  Write particles to traject.vtf file without sorting them
- `-sc`

Separate structure and coordinates into traject.vsf and traject.vcf files respectively (option can be used with `-b`)

By default, a single trajectory file will be written and the particles will not be sorted into order of global index number.

15.2.8 traject_selected_vtf.F90

The Fortran utility *traject_selected_vtf.F90* works in a similar fashion to *traject_vtf.F90* but allows the user to select which particles and timesteps to output to the trajectory file. If `f90` is the command for the available Fortran compiler, the executable `trajects_vtf.exe` can be produced by typing

```
f90 -o trajects_vtf.exe traject_selected_vtf.F90
```

and run at the command line (note that it cannot be invoked using the GUI). The executable for this utility can be run with the following optional command line arguments:

- `-s`
  Write particles to traject.vtf file after sorting them in order by particle number
- `-u`
  Write particles to traject.vtf file without sorting them
- `-sc`
  Separate structure and coordinates into traject.vsf and traject.vcf files respectively

By default the particles will not be sorted into order of global index number. Beyond these, the user will be shown information about the contents of the *HISTORY* file and asked to type in what is required in the traject.vtf file.

The user will then need to choose which particles are to be written to the VTF file (or VSF and VCF files): the utility will first show the user how many particles (total and unbonded) are in each frame of the *HISTORY* file, as well as the names and numbers of particle species and molecule types. The user will then be asked to select one of the following options to include in the file(s):

1. All particles
2. A contiguous range of particles based on global particle indices
3. All particles belonging to particular species and molecule types
4. Individual particle numbers
5. Individual molecule numbers

and based on this choice, further questions will be asked to select the range of particle indices, species or molecule types, particle numbers or molecule numbers respectively. The user will then be shown the number of available trajectory frames in the *HISTORY* file and asked to select which one to start with, which one to end with and the frequency to write to the output file(s) (i.e. whether or not to skip over frames).

15.2.9 traject_xml.F90

The Fortran utility *traject_xml.F90* reads in *HISTORY* output data files generated by DL_MESO_DPD and produces a series of numbered trajectory files in GALAMOST [157] XML format (traject_*.xml) that can be used to visualize the simulation using OVITO [132]. If molecules are present, the bond data are included in these files and can be shown in snapshots and animations generated by OVITO.

The source code for this utility reads *HISTORY* files generated by the serial and parallel versions of DL_MESO_DPD: since *HISTORY* files include a number in their headers as an endianness check, the utility should be able to read these files created on different machines. The utility outputs every particle for all recorded timesteps, including bond data for particles in molecules. (Each molecule type is used to identify the types of bonds between particles.)

If `f90` is the command for the available Fortran compiler, the executable `traject_xml.exe` can be produced by typing

```
f90 -o traject_xml.exe traject_xml.F90
```

and either run at the command line or via the GUI in **Process DPD Data**.

No command line arguments are required to run this utility. A series of trajectory files will be written: because global index numbers cannot be explicitly assigned in this format, the particles' data are sorted by global index before being written to the files. If velocities are included in the *HISTORY* file, these will also be written to the trajectory files.

15.2.10 traject_selected_xml.F90

The Fortran utility *traject_selected_xml.F90* works in a similar fashion to *traject_xml.F90* but allows the user to select which particles and timesteps to output to the trajectory files. If `f90` is the command for the available Fortran compiler, the executable `trajects_xml.exe` can be produced by typing

```
f90 -o trajects_xml.exe traject_selected_xml.F90
```

and run at the command line (note that it cannot be invoked using the GUI). The user will be shown information about the contents of the *HISTORY* file and asked to type in what is required in the traject_*.xml files.

The user will then need to choose which particles are to be written to the XML files: the utility will first show the user how many particles (total and unbonded) are in each frame of the *HISTORY* file, as well as the names and numbers of particle species and molecule types. The user will then be asked to select one of the following options to include in the file(s):

1. All particles
2. A contiguous range of particles based on global particle indices
3. All particles belonging to particular species and molecule types
4. Individual particle numbers
5. Individual molecule numbers

and based on this choice, further questions will be asked to select the range of particle indices, species or molecule types, particle numbers or molecule numbers respectively. The user will then be shown the number of available trajectory frames in the *HISTORY* file and asked to select which one to start with, which one to end with and the frequency to write to the output file(s) (i.e. whether or not to skip over frames).

15.2.11 isosurfaces.F90

The Fortran utility *isosurfaces.F90* reads in *HISTORY* output data files generated by DL_MESO_DPD and produces grid-based density maps at each recorded timestep for a specified species in Legacy VTK format for visualization of isosurfaces. It also calculates the second moment of the isosurface normal distribution, whose eigenvalues can be used to determine the mesophase for the system.

The source code for this utility can read *HISTORY* files generated by the serial and parallel versions of DL_MESO_DPD: an endianness check to their headers will allow the utility to work out which endianness to use for reading them. An executable `isosurfaces.exe` can be created by typing

```
f90 -o isosurfaces.exe isosurfaces.F90
```

if `f90` is the command for the available Fortran compiler. This can be run either at the command line or via the GUI in **Process DPD Data**.

The executable for this utility can be run with the following command line arguments:

- `-b [SPECIES]`

Use particle species [SPECIES] to create density maps (either name or number based on its order in the *FIELD* file)

- -p f
  Set spacing between grid points to f (overriding default value of 0.25)
- -s f
  Set Gaussian standard deviation σ to f (overriding default value of 0.4)
- -sf i
  Start collecting and writing data from frame i of *HISTORY* file (overriding default value of 1)
- -sl i
  Finish collecting and writing data at frame i of *HISTORY* file (overriding default value of last available frame)
- -tf i
  Set frequency of frames used from *HISTORY* file to i (overriding default value of 1)

For instance, if isosurfaces of species TAIL are required with a target grid spacing of 0.2 and a Gaussian standard deviation of 0.5 between frames 4 and 10 every 2 frames, the utility can be launched using one of the following commands (depending on operating system):

```
isosurfaces.exe -b TAIL -p 0.2 -s 0.5 -sf 4 -sl 10 -tf 2
./isosurfaces.exe -b TAIL -p 0.2 -s 0.5 -sf 4 -sl 10 -tf 2
```

Note that if no species is specified at the command line, the user will be asked to choose one based on the contents of the *HISTORY* file.

For each recorded timestep, the volume of every particle is smeared using a Gaussian function of standard deviation σ:

$$f(\vec{r}) = \frac{1}{(2\pi\sigma^2)^{\frac{3}{2}}} \exp\left(-\frac{|\vec{r}-\vec{r}_i|^2}{2\sigma^2}\right)$$

where $\vec{r}_i$ is the position of particle i. All points on a regular orthogonal grid within a distance of 3σ from the particle position are assigned contributions from this smearing function. The resulting totals for all particles of the specified species are subsequently written to Legacy VTK files named density_.vtk. The densities can then be used to construct the isosurface normal distribution $p(\vec{n})$, using the mean value of density over the system as the threshold for isosurfaces. Its second moment

$$\mathbf{M} = \int \vec{n}\vec{n}p(\vec{n})d\vec{n}$$

gives an indication of how the particles in the species are arranged in the system. The three eigenvalues of the second moment (μ_1, μ_2, μ_3) can be used as mesophase order [105][143]: $\mu_1 \approx \mu_2 \approx \mu_3$ indicates an isotropic mesophase, $\mu_1 \ll \mu_2, \mu_3$ indicates a hexagonal mesophase and $\mu_1, \mu_2 \ll \mu_3$ indicates a lamellar mesophase. The eigenvalues are written to a file named *moment*, which contains columns for the time and the three eigenvalues in numerical order: this file can be imported into graph plotting software to display how the system mesophase changes over time.

15.2.12 radius.F90

The Fortran utility *radius.F90* reads in *HISTORY* output data files generated by DL_MESO_DPD and calculates the end-to-end distances and radii of gyration at each recorded timestep for all molecules in the system, as well as finding the time-averaged distributions of end-to-end distances for each molecule type.

The source code for this utility can read *HISTORY* files generated by the serial and parallel versions of DL_MESO_DPD. If f90 is the command for the available Fortran compiler, the executable radius.exe can be produced by typing

```
f90 -o radius.exe radius.F90
```

and run either at the command line or via the GUI in **Process DPD Data**.

The executable for this utility can be run with the following command line arguments:

- `-c` f
  Set maximum end-to-end distance for distribution to f (overriding default value of 2.0)
- `-d` f
  Set histogram spacing for distribution to f (overriding default value of 0.05)
- `-sf` i
  Start collecting and writing data from frame i of *HISTORY* file (overriding default value of 1)
- `-sl` i
  Finish collecting and writing data at frame i of *HISTORY* file (overriding default value of last available frame)
- `-tf` i
  Set frequency of frames used from *HISTORY* file to i (overriding default value of 1)

Note that if the *HISTORY* file includes no molecules, the utility will stop with an error message.

For each recorded timestep and all molecules, the end-to-end distance along the main molecular branch and the radius of gyration

$$R_g^2 = \frac{1}{N} \sum_i^N \left(\vec{r}_i - \vec{r}_{mean}\right)^2$$

are calculated, where N is the number of particles in the molecule and $\vec{r}_{mean} = \frac{\sum_i^N m_i \vec{r}_i}{\sum_i^N m_i}$ is the centre-of-mass for the molecule. Files named *radius_* * are produced for each molecule type: these text files contain columns for the time, root mean squared end-to-end distance, the mean squared end-to-end distance and root mean squared radius of gyration over all molecules of the specified type, which can be plotted using graph plotting software.

A single file named *MOLDIST* is also produced to give the time-averaged distributions of end-to-end distance for every molecule type, normalised to give $\int_0^\infty 4\pi r^2 dr\, g(r) = 1$. The file starts with two lines, the first giving the name of the simulation and the second with the number of timesteps used and the number of distance divisions used (the number of shells). The distributions for each molecule type are then given, starting with a line giving the molecule name and followed by columns with the radius r (the mid-point for each shell) and the distribution $g(r)$: each type is separated by two blank lines.

15.2.13 dipole.F90

The Fortran utility *dipole.F90* reads in *HISTORY* output data files generated by DL_MESO_DPD and calculates the total dipole moments for each molecule type, autocorrelation functions of dipole moments and (optionally) their Fourier transforms.

The source code for this utility can read *HISTORY* files generated by the serial and parallel versions of DL_MESO_DPD, regardless of which endianness is used. If `f90` is the command for the available Fortran compiler, the executable `dipole.exe` can be produced by typing

```
f90 -o dipole.exe dipole.F90
```

and run either at the command line or via the GUI in **Process DPD Data**.

The executable for this utility can be run with the following command line arguments:

- `-n` i
  Set number of bins to calculate dipole autocorrelation functions (DAFs) to i
- `-fft`

Calculate Fourier Transforms of dipole autocorrelation functions

- `-fc` i

  Set number of bins for DAFs to i (overriding default value of maximum between 500 and double the number of bins for DAF calculations)
- `-sf` i

  Start collecting and writing data from frame i of *HISTORY* file (overriding default value of 1)
- `-sl` i

  Finish collecting and writing data at frame i of *HISTORY* file (overriding default value of last available frame)
- `-tf` i

  Set frequency of frames used from *HISTORY* file to i (overriding default value of 1)

If no value for the number of bins for dipole autocorrelation functions is given in the command line or it exceeds the number of frames in the *HISTORY* file (or the number used based on starting/finishing frames and frequency), the user will be asked to type one in. Note that no Fourier transforms for the dipole autocorrelation functions will be calculated unless the `-fft` command line argument is used.

For each recorded timestep, the total dipole moment for each molecule is calculated using the formula $\vec{p} = \sum_i q_i \vec{r}_i$, where q_i is the charge on particle i (obtained from the *HISTORY* file) and $\vec{r}_i$ is the position of that particle with adjustments made for periodic boundary conditions where necessary. These dipole moments are summed up for each molecule type to give the total dipole moment $\vec{P}$. Files named `dipole_` are produced for each molecule type: these text files contain columns for the time, x-, y- and z-components of the dipole moment, the squared dipole moment P^2 and the ratio of the squared dipole moment P^2 to volume ($\frac{P^2}{V}$).

The dipole autocorrelation function for each molecule type is calculated as $C(t) = \left\langle \vec{P}(0) \cdot \vec{P}(t) \right\rangle$ over the number of time steps given by the user, ensemble averaging across all possible samples from the given trajectory data. A single file called *DIPOLEDAT* is produced, containing the DAFs for all available molecule types. Starting with two lines – the first with the simulation name, the second with the total number of timesteps used and the number of time steps used for DAFs – the names and data for each molecule type is given, the latter in columns for time t, the absolute value of the ensemble averaged DAF $C(t)$ and the value scaled with the value at $t = 0$ (i.e. $Z(t) = \frac{C(t)}{C(0)}$). The data for each molecule type are separated by two blank lines.

If the Fourier transforms are requested, this appears in an additional file called *DIPOLEFFT*, which is similarly formatted to DIPOLEDAT apart from the data for each molecule type, which are the frequency k and the real and imaginary terms of the Fourier transform $S(k)$ (i.e. $\Re(S(k))$ and $\Im(S(k))$).

15.2.14 rdf.F90

rdf.F90 is a utility written in Fortran that can read in *HISTORY* output data files generated by DL_MESO_DPD and determine radial distribution functions (RDFs) between all pairs of particle species (including self-interactions).

The source code for this utility can read *HISTORY* files generated by the serial and parallel versions of DL_MESO_DPD. It can exploit OpenMP multithreading to speed up RDF calculations: the source code includes directives to use different blocks of code depending on whether or not it is compiled with OpenMP, which require preprocessing. (The source code filename extension `.F90` should automatically invoke the compiler's C preprocessor.)

If `f90` is the command for the available Fortran compiler, the executable `rdf.exe` can be produced by typing either

```
f90 -o rdf.exe rdf.F90
```

for the serial (single thread) version, or by typing

```
f90 -o rdf.exe -openmp rdf.F90
```

for the OpenMP multithreaded version, substituting `-openmp` with the required compiler flag for invoking OpenMP (e.g. `-fopenmp` for `gfortran`).

The utility can either be run at the command line or via the GUI in **Process DPD data**. A number of command line arguments can be used after the command:

- `-c` f
  Set maximum distance between pairs of particles to f (overriding default value of 2.0)
- `-d` f
  Set histogram spacing for radial distribution function calculations to f (overriding default value of 0.05)
- `-fft`
  Calculate Fourier Transforms of radial distribution functions (structure factors)
- `-fc` i
  Set number of bins for structure factors to i (overriding default value of maximum between 500 and double the number of bins for RDF calculations)
- `-sf` i
  Start collecting and writing data from frame i of *HISTORY* file (overriding default value of 1)
- `-sl` i
  Finish collecting and writing data at frame i of *HISTORY* file (overriding default value of last available frame)
- `-tf` i
  Set frequency of frames used from *HISTORY* file to i (overriding default value of 1)

Note that no Fourier transforms for the radial distribution functions will be calculated unless the `-fft` command line argument is used.

At each recorded timestep, linked-cell lists for the particles are created and used to determine the particle-particle distances between pairs within the maximum distance r_c. These distances are used to increase counters for the appropriate histogram bins (of width Δr) for each species pair. The sums of these histograms are averaged over time to give $n(r)$ (the mean number of particles in the bin at distance r) and the radial distribution function is given as

$$g(r) = \frac{n(r)}{4\pi r^2 \Delta r \rho}$$

where ρ is the mean particle density. The Fourier transform of $g(r)$ gives the structure factor:

$$S(k) = 1 + \frac{4\pi\rho}{k} \int_0^\infty (g(r) - 1) r \sin(kr) dr$$

which is a property that can be measured experimentally using e.g. X-ray diffraction.

A single file *RDFDAT* containing all radial distribution function data is produced. Starting with two lines – the first giving the simulation name, the second giving the number of timesteps and the number of histogram bins used – the names and the data for each species pair are given, the latter in columns for the radius r (the mid-point for each shell), the RDF $g(r)$ and the sum of $g(r)$ (the average number of particles of one type within r around a particle of the other). The data for each species pair are separated by two blank lines and an additional data set of the same form is given for all particles regardless of species type at the end of the file.

If the Fourier transforms are requested, an additional *RDFFFT* file is created. This file has a similar format to RDFDAT, except that the data is given in two columns: one for the frequency k and the other for $S(k)$ (the Fourier transform of $g(r) - 1$).

15.2.15 rdfmol.F90

rdfmol.F90 is a utility written in Fortran that can read in *HISTORY* output data files generated by DL_MESO_DPD and determine radial distribution functions (RDFs) between all pairs of molecules by type (including self-interactions).

The source code for this utility can read *HISTORY* files generated by the serial and parallel versions of DL_MESO_DPD. It can exploit OpenMP multithreading to speed up calculations: the source code includes directives to use different blocks of code depending on whether or not it is compiled with OpenMP.

If `f90` is the command for the available Fortran compiler, the executable `rdfmol.exe` can be produced by typing either

```
f90 -o rdfmol.exe rdfmol.F90
```

for the serial (single thread) version, or by typing

```
f90 -o rdfmol.exe -openmp rdfmol.F90
```

for the OpenMP multithreaded version, substituting `-openmp` with the required compiler flag for invoking OpenMP.

The utility can either be run at the command line or via the GUI in **Process DPD data**. A number of command line arguments can be used after the command:

- `-c` f
  Set maximum distance between pairs of particles to f (overriding default value of 2.0)
- `-d` f
  Set histogram spacing for radial distribution function calculations to f (overriding default value of 0.05)
- `-fft`
  Calculate Fourier Transforms of radial distribution functions (structure factors)
- `-fc` i
  Set number of bins for DAFs to i (overriding default value of maximum between 500 and double the number of bins for RDF calculations)
- `-sf` i
  Start collecting and writing data from frame i of *HISTORY* file (overriding default value of 1)
- `-sl` i
  Finish collecting and writing data at frame i of *HISTORY* file (overriding default value of last available frame)
- `-tf` i
  Set frequency of frames used from *HISTORY* file to i (overriding default value of 1)

Note that no Fourier transforms for the radial distribution functions will be calculated unless the `-fft` command line argument is used.

At each recorded timestep, the centres of mass for each molecule are determined (taking periodic or shearing boundaries into account) and these coordinates are assigned to linked-cell lists, which are used to determine the distances between pairs of molecules within the maximum distance r_c. These distances are used to fill histograms with bin sizes of Δr and are later time-averaged and divided by the shell volumes (as for standard RDFs) to give the radial distribution functions between molecule types. Fourier transforms of molecular RDFs give structure factors for the molecules.

The standard output from this utility is the file *RDFMOLDAT*: this is formatted in a similar manner to RDFDAT (as described for *rdf.F90*) except the RDF data is given by molecule type pairs. If Fourier transforms are requested, these are given in the file *RDFMOLFFT* and formatted similarly to RDFFFT.

15.2.16 local.F90

local.F90 is a utility written in Fortran that can read in *HISTORY* output data files generated by DL_MESO_DPD and produce series of Legacy VTK format files containing statistical properties – number of beads, density, compositions per particle and molecule types, temperature and mean velocity, local stress tensors – in cuboidal subdivisions of the simulation volume for plotting and/or visualization.

The source code for this utility can read *HISTORY* files generated by the serial and parallel versions of DL_MESO_DPD: the utility can be used on different machines to the one used for DPD calculations as it includes an endianness check for reading data from the file. If `f90` is the command for the available Fortran 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 the number of divisions required in each dimension.

This utility can be run with the following command line arguments specifying the number of divisions in each dimension:

- `-nx` i

  Set number of system divisions in x-dimension to i
- `-ny` i

  Set number of system divisions in y-dimension to i
- `-nz` i

  Set number of system divisions in z-dimension to i
- `-av` i

  Only write time-averaged data (suppress files for each frame)
- `-sf` i

  Start collecting and writing data from frame i of *HISTORY* file (overriding default value of 1)
- `-sl` i

  Finish collecting and writing data at frame i of *HISTORY* file (overriding default value of last available frame)
- `-tf` i

  Set frequency of frames used from *HISTORY* file to i (overriding default value of 1)

If these values are not set at the command line, the user will be asked to enter these values.

Unless the `-av` option is selected, files named local_*.vtk are produced for all the specified time steps after equilibration containing the data for each cuboidal cell. The level of data available in the *HISTORY* files (i.e. *keytrj*: 0 = particle positions, 1 = particle positions and velocities, 2 = particle positions, velocities and forces) dictates which of the following properties are output:

- the number of unfrozen beads
- densities for each bead species
- volume fractions for bead species
- volume fractions for molecule types (including a 'type' for all unbonded beads)
- the mean velocity for all unfrozen beads
- overall temperature
- partial temperatures for each dimension (i.e. for dimension α, $T_\alpha = \frac{\sum_i m_i v_{i,\alpha}^2}{N}$)
- local pressure tensors (based on Method of Planes [138])

An additional file, averages.vtk, is also produced with time-averaged values for the species densities, velocities, overall and partial temperatures, and local pressure tensor in each cuboidal cell based on the starting/finishing frames and frequency selected.

The scalar properties (including compositions) may be considered to act across the entire volumes of the cells, while the velocities and pressure tensors are representative of the cell centres.

15.2.17 widom_insertion.F90

widom_insertion.F90 is a utility written in Fortran that can carry out Widom insertions to determine excess chemical potentials [149][150]. This utility reads in trajectories from *HISTORY* output data files generated by DL_MESO_DPD, along with relevant simulation and interaction information from *CONTROL* and *:field* input files, and randomly inserts a user-selected particle or molecule at different positions (and orientations) in the simulation box. The change in potential energy due to each insertion of particle or molecule i (ΔU_i) is measured and an ensemble average can be used to calculate the excess chemical potential, i.e.

$$\mu_i^{ex} = -k_B T \ln \left\langle \exp\left(-\frac{\Delta U_i}{k_B T}\right)\right\rangle$$

for a constant volume ensemble, or

$$\mu_i^{ex} = -k_B T \ln \left(\frac{\left\langle V \exp\left(-\frac{\Delta U_i}{k_B T}\right)\right\rangle}{\langle V \rangle}\right)$$

for simulations with varying volume (constant pressure, surface area or surface tension).

The source code for this utility can read *HISTORY* files generated by the serial and parallel versions of DL_MESO_DPD: endianness checks in the utility mean it should be possible to run the utility on a different machine to that used to generate the trajectory data. It can exploit OpenMP multithreading to speed up calculations: the source code includes directives to use different blocks of code depending on whether or not it is compiled with OpenMP.

If `f90` is the command for the available Fortran compiler, the executable `widom.exe` can be produced by typing

```
f90 -o widom.exe widom_insertion.F90
```

for the serial (single thread) version, or by typing

```
f90 -o widom.exe -openmp widom_insertion.F90
```

for the OpenMP multithreaded version, substituting `-openmp` with the required compiler flag for invoking OpenMP (e.g. `-fopenmp` for `gfortran`). The utility can be run either at the command line or via the GUI in **Process DPD Data**.

This utility can be run with the following command line arguments:

- `-p [SPECIES]`
  Insert a single particle of species `[SPECIES]` for each trial insertion
- `-m [MOLE]`
  Insert a molecule of type `[MOLE]` for each trial insertion
- `-rm`
  Use a randomly chosen molecule in each trajectory frame as a template for molecule trial insertion (overriding default of configuration given in *FIELD* file)
- `-n` i
  Set number of trial insertions per trajectory frame to i
- `-sf` i
  Start accumulating statistics on trial insertions from trajectory frame i (overriding default of 1)

- `-sl` i

  Stop accumulating statistics on trial insertions at trajectory frame i (overriding default of last available frame)
- `-r` i

  Set random number generator seed to i (only if RNDSEED file is unavailable)

The particle or molecule type and the number of trial insertions per frame are essential to carry out Widom insertions: if these are not included in the command line, the user will be asked to type in these data.

The utility will produce two files: a file called RNDSEED giving the final state of the random number generator (which can be used as the initial state for future calculations), and one called CHEMPOT_*, ending with the name of the bead species or molecule being inserted. The latter plottable file contains five columns with the time (in DPD units), the 'instantaneous' block-averaged excess chemical potential for the trajectory frame and its standard deviation, the time-averaged excess chemical potential and its standard deviation. The utility will also output the same information for each trajectory timestep to the screen or standard output.

APPENDIX

A

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. UKRI 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 UKRI STFC. UKRI STFC may refuse to grant the Licensee a commercial licence. If UKRI STFC agrees to grant a commercial licence, that licence will be on such terms and conditions as UKRI 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 UKRI 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 UKRI STFC gives no warranty and makes no representation in relation to the DL_MESO Software or any assistance or advice that UKRI STFC may give in connection with the DL_MESO Software. The Licensee will indemnify UKRI 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. UKRI STFC does not warrant that the DL_MESO Software will run without interruption or be error free, or be free from any Harmful Element. UKRI 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 UKRI STFC chooses to make available on the DL_MESO Website from time to time. If UKRI STFC does provide that sort of service, assistance or advice, subject to clause 3.7, UKRI STFC will not be liable for any loss or damage suffered by the Licensee as a result.

3.3. UKRI 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 UKRI 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. UKRI STFC will not be liable for the consequences of decisions taken by the Licensee or any other person on the basis of that output. UKRI 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, UKRI 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 UKRI STFC of the possibility of those losses arising or if they were or are within UKRI STFC's contemplation.

3.6. Subject to clause 3.7, the aggregate liability of UKRI 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 UKRI 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 UKRI 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 UKRI 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 UKRI 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 UKRI STFC's Daresbury Laboratory via the website http://www.ccp5.ac.uk/DL_MESO"

and cite the following reference:

> "M.A. Seaton, R.L. Anderson, S. Metz & W. Smith, *Mol. Sim.*, **39** (10), 796-821 (2013)".

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. UKRI 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 UKRI 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 UKRI 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 UKRI 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 UKRI 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 UKRI STFC may bring proceedings against the Licensee or for an injunction in any jurisdiction.

BIBLIOGRAPHY

[1] Y. Afshar, F. Schmid, A. Pishevar, and S. Worley. Exploiting seeding of random number generators for efficient domain decomposition parallelization of dissipative particle dynamics. *Computer Physics Communications*, 184(4):1119–1128, April 2013. doi:10.1016/j.cpc.2012.12.003.

[2] Hans C. Andersen. Molecular dynamics simulations at constant pressure and/or temperature. *Journal of Chemical Physics*, 72(4):2384–2393, 1980. doi:10.1063/1.439486.

[3] Hans C. Andersen. RATTLE: a "velocity" version of the SHAKE algorithm for molecular dynamics calculations. *Journal of Computational Physics*, 52(1):24–34, 1983. doi:10.1016/0021-9991(83)90014-1.

[4] Richard L. Anderson, David J. Bray, Annalaura Del Regno, Michael A. Seaton, Andrea S. Ferrante, and Patrick B. Warren. Micelle formation in alkyl sulfate surfactants using dissipative particle dynamics. *Journal of Chemical Theory and Computation*, 14(5):2633–2643, 2018. PMID: 29570296. doi:10.1021/acs.jctc.8b00075.

[5] Santosh Ansumali and Iliya V. Karlin. Kinetic boundary conditions in the lattice Boltzmann method. *Physical Review E*, 66:026311, August 2002. doi:10.1103/PhysRevE.66.026311.

[6] Abdel Monim Mohamed Ali Mohamed Hassan Artoli. *Mesoscopic computational haemodynamics*. PhD thesis, University of Amsterdam, October 2003.

[7] Pietro Asinari. Generalized local equilibrium in the cascaded lattice Boltzmann method. *Physical Review E*, 78:016701, July 2008. doi:10.1103/PhysRevE.78.016701.

[8] 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. doi:10.1063/1.448118.

[9] 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. doi:10.1103/PhysRevE.62.R7611.

[10] 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. doi:10.1103/PhysRev.94.511.

[11] Eugene C. Bingham. *Fluidity and plasticity*. McGraw-Hill, first edition, 1922.

[12] J. Boyd, J. Buick, and S. Green. A second-order accurate lattice Boltzmann non-Newtonian flow model. *Journal of Physics A: Mathematical and General*, 39(46):14241, 2006. doi:10.1088/0305-4470/39/46/001.

[13] A. J. Briant, P. Papatzacos, and J. M. Yeomans. Lattice Boltzmann simulations of contact line motion in a liquid-gas system. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 360(1792):485–495, 2002. doi:10.1098/rsta.2001.0943.

[14] I. J. Bush, I. T. Todorov, and W. Smith. A DAFT DL_POLY distributed memory adaptation of the smoothed particle mesh Ewald method. *Computer Physics Communications*, 175(5):323–329, September 2006. doi:10.1016/j.cpc.2006.05.001.

[15] R. Byron Bird and Pierre J. Carreau. A nonlinear viscoelastic model for polymer solutions and melts–I. *Chemical Engineering Science*, 23(5):427–434, 1968. doi:10.1016/0009-2509(68)87018-6.

[16] Norman F. Carnahan and Kenneth E. Starling. Intermolecular repulsions and the equation of state for fluids. *AIChE Journal*, 18(6):1184–1189, 1972. doi:10.1002/aic.690180615.

[17] Pierre J. Carreau, Ian F. MacDonald, and R. Byron Bird. A nonlinear viscoelastic model for polymer solutions and melts–II. *Chemical Engineering Science*, 23(8):901–911, 1968. doi:10.1016/0009-2509(68)80024-7.

[18] Mauricio Carrillo-Tripp, Humberto Saint-Martin, and Iván Ortega-Blake. A comparative study of the hydration of Na+ and K+ with refined polarizable model potentials. *Journal of Chemical Physics*, 118(15):7062–7073, 2003. doi:10.1063/1.1559673.

[19] N. Casson. Flow equation for pigment oil suspensions of the printing ink type. In C. C. Mill, editor, *Rheology of disperse systems*, 84–102. Pergamon Press, 1959.

[20] Zhenhua Chai, Baochang Shi, Zhaoli Guo, and Fumei Rong. Multiple-relaxation-time lattice Boltzmann model for generalized Newtonian fluid flows. *Journal of Non-Newtonian Fluid Mechanics*, 166(5):332–342, 2011. doi:10.1016/j.jnnfm.2011.01.002.

[21] Cheng Chang, Chih-Hao Liu, and Chao-An Lin. Boundary conditions for lattice Boltzmann simulations with complex geometry flows. *Computers & Mathematics with Applications*, 58(5):940–949, 2009. doi:10.1016/j.camwa.2009.02.016.

[22] 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. doi:10.1080/08927020701713894.

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

[24] Daniele Coslovich, Jean-Pierre Hansen, and Gerhard Kahl. Ultrasoft primitive model of polyionic solutions: Structure, aggregation, and dynamics. *Journal of Chemical Physics*, 134(24):244514, 2011. doi:10.1063/1.3602469.

[25] 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*, 51(4):3718–3728, April 1995. doi:10.1103/PhysRevE.51.3718.

[26] Burkhard Dünweg and Wolfgang Paul. Brownian dynamics simulations without Gaussian random numbers. *International Journal of Modern Physics C*, 2(3):817–827, 1991. doi:10.1142/S0129183191001037.

[27] Paul J. Dellar. Bulk and shear viscosities in lattice Boltzmann equations. *Physical Review E*, 64(3):031203, August 2001. doi:10.1103/PhysRevE.64.031203.

[28] Michael M. Dupin, Ian Halliday, and Chris M. Care. Simulation of a microfluidic flow-focusing device. *Physical Review E*, 73(5):055701, 2006. doi:10.1103/PhysRevE.73.055701.

[29] A. Dupuis and J. M. Yeomans. Modeling droplets on superhydrophobic surfaces: equilibrium states and transitions. *Langmuir*, 21(6):2624–2629, 2005. doi:10.1021/la047348i.

[30] Pep Español. Hydrodynamics from dissipative particle dynamics. *Physical Review E*, 52(2):1734–1742, August 1995. doi:10.1103/PhysRevE.52.1734.

[31] Ulrich Essmann, Lalith Perera, Max L. Berkowitz, Tom Darden, Hsing Lee, and Lee G. Pedersen. A smooth particle mesh Ewald method. *Journal of Chemical Physics*, 103(19):8577–8593, 1995. doi:10.1063/1.470117.

[32] P. P. Ewald. Die Berechnung optischer und elektrostatischer Gitterpotentiale. *Annalen der Physik*, 369(3):253–287, 1921. doi:10.1002/andp.19213690304.

[33] Linlin Fei, Kai H. Luo, and Qing Li. Three-dimensional cascaded lattice Boltzmann method: improved implementation and consistent forcing scheme. *Physical Review E*, 97:053309, May 2018. doi:10.1103/PhysRevE.97.053309.

[34] Linlin Fei and Kai Hong Luo. Consistent forcing scheme in the cascaded lattice Boltzmann method. *Physical Review E*, 96:053307, November 2017. doi:10.1103/PhysRevE.96.053307.

[35] Linlin Fei, Kai Hong Luo, Chuandong Lin, and Qing Li. Modeling incompressible thermal flows using a central-moments-based lattice Boltzmann method. *International Journal of Heat and Mass Transfer*, 120:624–634, 2018. doi:10.1016/j.ijheatmasstransfer.2017.12.052.

[36] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the Institute of Electrical and Electronics Engineers (IEEE)*, 93(2):216–231, January 2005. doi:10.1109/JPROC.2004.840301.

[37] A. A. Gavrilov, A. V. Chertovich, and E. Yu. Kramarenko. Dissipative particle dynamics for systems with high density of charges: implementation of electrostatic interactions. *Journal of Chemical Physics*, 145(17):174101, 2016. doi:10.1063/1.4966149.

[38] Martin Geier, Andreas Greiner, and Jan G. Korvink. Cascaded digital lattice Boltzmann automata for high Reynolds number flow. *Physical Review E*, 73:066705, June 2006. doi:10.1103/PhysRevE.73.066705.

[39] A. Ghoufi and P. Malfreyt. Mesoscale modeling of the water liquid-vapor interface: A surface tension calculation. *Physical Review E*, 83:051601, May 2011. doi:10.1103/PhysRevE.83.051601.

[40] 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. doi:10.1142/S0129183199000176.

[41] Irina Ginzburg, Frederik Verhaeghe, and Dominique d'Humières. Study of simple hydrodynamic solutions with the two-relaxation-times lattice Boltzmann scheme. *Communications in Computational Physics*, 3(3):519–581, March 2008. URL: http://www.global-sci.com/freedownload/v3_519.pdf.

[42] Irina Ginzburg, Frederik Verhaeghe, and Dominique d'Humières. Two-relaxation-time lattice Boltzmann scheme: About parametrization, velocity, pressure and mixed boundary conditions. *Communications in Computational Physics*, 3(2):427–478, February 2008. URL: http://www.global-sci.com/freedownload/v3_427.pdf.

[43] Shuai Gong and Ping Cheng. Numerical investigation of droplet motion and coalescence by an improved lattice Boltzmann model for phase transitions and multiphase flows. *Computers & Fluids*, 53(0):93–104, 2012. doi:10.1016/j.compfluid.2011.09.013.

[44] Minerva González-Melchor, Estela Mayoral, Mar\'ıa Eugenia Velázquez, and José Alejandre. Electrostatic interactions in dissipative particle dynamics using the Ewald sums. *Journal of Chemical Physics*, 125(22):224107, 2006. doi:10.1063/1.2400223.

[45] R. D. Groot. Electrostatic interactions in dissipative particle dynamics—simulation of polyelectrolytes and anionic surfactants. *Journal of Chemical Physics*, 118(24):11265–11277, 2003. doi:10.1063/1.1574800.

[46] 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. doi:10.1063/1.474784.

[47] 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. doi:10.1103/PhysRevA.43.4320.

[48] 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. doi:10.1002/fld.337.

[49] Zhaoli Guo, Chuguang Zheng, and Baochang Shi. Discrete lattice effects on the forcing term in the lattice Boltzmann method. *Physical Review E*, 65(4):046308, April 2002. doi:10.1103/PhysRevE.65.046308.

[50] I. Halliday, A. P. Hollis, and C. M. Care. Lattice Boltzmann algorithm for continuum multicomponent flow. *Physical Review E*, 76(2):026708, 2007. doi:10.1103/PhysRevE.76.026708.

[51] 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*, 73(5):056708, 2006. doi:10.1103/PhysRevE.73.056708.

[52] Cecil Hastings. *Approximations for digital computers*. Princeton University Press, Princeton, NJ, 1955.

[53] 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. doi:10.1023/B:JOSS.0000015179.12689.e4.

[54] Xinoyi He, Xiaowen Shan, and Gary D. Doolen. Discrete Boltzmann equation model for nonideal gases. *Physical Review E*, 57(1):R13–R16, January 1998. doi:10.1103/PhysRevE.57.R13.

[55] Winslow H. Herschel and Ronald Bulkley. Konsistenzmessungen von Gummi-Benzollösungen. *Kolloid-Zeitschrift*, 39(4):291–300, 8 1926. doi:10.1007/BF01432034.

[56] F. J. Higuera and J. Jiménez. Boltzmann approach to lattice gas simulations. *EPL (Europhysics Letters)*, 9(7):663–668, 1989. doi:10.1209/0295-5075/9/7/009.

[57] R. W. Hockney and J. W. Eastwood. *Computer simulation using particles*. McGraw-Hill International, 1981.

[58] D. J. Holdych, D. Rovas, J. G. Georgiadis, and R. O. Buckius. An improved hydrodynamics formulation for multiphase flow Lattice-Boltzmann models. *International Journal of Modern Physics C*, 9(8):1393–1404, 1998. doi:10.1142/S0129183198001266.

[59] Kainan Hu, Jianping Meng, Hongwu Zhang, Xiao-Jun Gu, David R. Emerson, and Yonghao Zhang. A comparative study of boundary conditions for lattice Boltzmann simulations of high Reynolds number flows. *Computers & Fluids*, 156:1–8, 2017. doi:10.1016/j.compfluid.2017.06.008.

[60] William Humphrey, Andrew Dalke, and Klaus Schulten. VMD: visual molecular dynamics. *Journal of Molecular Graphics*, 14(1):33–38, 1996. doi:10.1016/0263-7855(96)00018-5.

[61] M. K. Ikeda, P. R. Rao, and L. A. Schaefer. A thermal multicomponent lattice Boltzmann model. *Computers & Fluids*, 101:250–262, 2014. doi:10.1016/j.compfluid.2014.06.006.

[62] 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. doi:10.1006/jcph.2002.7051.

[63] Takaji Inamuro, Masato Yoshino, and Fumimaru Ogino. A non-slip boundary condition for lattice Boltzmann simulations. *Physics of Fluids*, 7(12):2928–2930, 1995. doi:10.1063/1.868766.

[64] Ask F. Jakobsen. Constant-pressure and constant-surface tension simulations in dissipative particle dynamics. *Journal of Chemical Physics*, 122(12):124901, 2005. doi:10.1063/1.1867374.

[65] Erik Johansson. Simulating fluid flow and heat transfer using dissipative particle dynamics. Project report, Department of Energy Sciences, Faculty of Engineering, Lund University, Department of Energy Sciences, Faculty of Engineering, Lund University, Box 118, 22100 Lund, Sweden, 2012. URL: http://www.ht.energy.lth.se/fileadmin/ht/Kurser/MVK160/2012/Erik_Johansson.pdf.

[66] 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. doi:10.1098/rspa.1924.0082.

[67] 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. doi:10.1039/A809824G.

[68] 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. doi:10.1209/0295-5075/21/3/018.

[69] A. L. Kupershtokh and D. A. Medvedev. Lattice Boltzmann equation method in electrohydrodynamic problems. *Journal of Electrostatics*, 64(7–9):581–585, 2006. doi:10.1016/j.elstat.2005.10.012.

[70] A. L. Kupershtokh, D. A. Medvedev, and D. I. Karpov. On equations of state in a lattice Boltzmann method. *Computers & Mathematics with Applications*, 58(5):965–974, 2009. Mesoscopic Methods in Engineering and Science. doi:10.1016/j.camwa.2009.02.024.

[71] A. Kuzmin, M. Januszewski, D. Eskin, F. Mostowfi, and J. J. Derksen. Three-dimensional binary-liquid lattice Boltzmann simulation of microchannels with rectangular cross sections. *Chemical Engineering Journal*, 178:306–316, 2011. doi:10.1016/j.cej.2011.10.010.

[72] Martin Lísal, John K. Brennan, and Josep Bonet Avalos. Dissipative particle dynamics at isothermal, isobaric, isoenergetic, and isoenthalpic conditions using Shardlow-like splitting algorithms. *Journal of Chemical Physics*, 135(20):204105, 2011. doi:10.1063/1.3660209.

[73] 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. doi:10.1103/PhysRevE.61.6546.

[74] Jonas Latt, Bastien Chopard, Orestis Malaspinas, Michel Deville, and Andreas Michler. Straight velocity boundaries in the lattice Boltzmann method. *Physical Review E*, 77(5):056703, May 2008. doi:10.1103/PhysRevE.77.056703.

[75] Taehun Lee and Ching-Long Lin. A stable discretization of the lattice Boltzmann equation for simulation of incompressible two-phase flows at high density ratio. *Journal of Computational Physics*, 206(1):16–47, June 2005. doi:10.1016/j.jcp.2004.12.001.

[76] A. W. Lees and S. F. Edwards. The computer study of transport processes under extreme conditions. *Journal of Physics C*, 5(15):1921–1928, 1972. doi:10.1088/0022-3719/5/15/006.

[77] Benedict Leimkuhler and Xiaocheng Shang. Pairwise adaptive thermostats for improved accuracy and stability in dissipative particle dynamics. *Journal of Computational Physics*, 324:174–193, November 2016. doi:10.1016/j.jcp.2016.07.034.

[78] M. Leslie and N. J. Gillan. The energy and elastic dipole tensor of defects in ionic crystals calculated by the supercell method. *Journal of Physics C*, 18(5):973, 1985. doi:10.1088/0022-3719/18/5/005.

[79] Q. Li, K. H. Luo, and X. J. Li. Forcing scheme in pseudopotential lattice Boltzmann model for multiphase flows. *Physical Review E*, 86:016709, July 2012. doi:10.1103/PhysRevE.86.016709.

[80] Qing Li, K. H. Luo, Q. J. Kang, and Q. Chen. Contact angles in the pseudopotential lattice Boltzmann modeling of wetting. *Physical Review E*, 90:053301, November 2014. doi:10.1103/PhysRevE.90.053301.

[81] S. V. Lishchuk, C. M. Care, and I. Halliday. Lattice Boltzmann algorithm for surface tension with greatly reduced microcurrents. *Physical Review E*, 67(3):036701, March 2003. doi:10.1103/PhysRevE.67.036701.

[82] Qin Lou, Zhaoli Guo, and Baochang Shi. Evaluation of outflow boundary conditions for two-phase lattice Boltzmann equation. *Physical Review E*, 87:063301, June 2013. doi:10.1103/PhysRevE.87.063301.

[83] C. P. Lowe. An alternative approach to dissipative particle dynamics. *EPL (Europhysics Letters)*, 47(2):145–151, July 1999. doi:10.1209/epl/i1999-00365-x.

[84] Daniel Lycett-Brown and Kai H. Luo. Multiphase cascaded lattice Boltzmann method. *Computers & Mathematics with Applications*, 67(2):350–362, 2014. doi:10.1016/j.camwa.2013.08.033.

[85] Daniel Lycett-Brown, Kai H. Luo, Ronghou Liu, and Pengmei Lv. Binary droplet collision simulations by a multiphase cascaded lattice Boltzmann method. *Physics of Fluids*, 26(2):023303, 2014. doi:10.1063/1.4866146.

[86] John F. Marko and Eric D. Siggia. Stretching DNA. *Macromolecules*, 28(26):8759–8770, 1995. doi:10.1021/ma00130a008.

[87] G. Marsaglia and T. A. Bray. A convenient method for generating normal variables. *SIAM Review*, 6(3):260–264, July 1964.

[88] George Marsaglia, Arif Zaman, and Wai Wan Tsang. Toward a universal random number generator. *Statistics and Probability Letters*, 9(1):35–39, January 1990. doi:10.1016/0167-7152(90)90092-L.

[89] 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. doi:10.1103/PhysRevE.56.1676.

[90] Glenn J. Martyna, Mark E. Tuckerman, Douglas J. Tobias, and Michael L. Klein. Explicit reversible integrators for extended systems dynamics. *Molecular Physics*, 87(5):1117–1157, 1996. doi:10.1080/00268979600100761.

[91] 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. doi:10.1103/PhysRevE.53.743.

[92] 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, January 1998. doi:10.1145/272991.272995.

[93] 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. doi:10.1016/j.cpc.2006.09.005.

[94] Philip M. Morse. Diatomic molecules according to the wave mechanics. II. Vibrational levels. *Physical Review*, 34(1):57–64, July 1929. doi:10.1103/PhysRev.34.57.

[95] 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. doi:10.1063/1.868767.

[96] Rafik Ouared and Bastien Chopard. Lattice Boltzmann simulations of blood flow: non-Newtonian rheology and clotting processes. *Journal of Statistical Physics*, 121:209–221, 2005. doi:10.1007/s10955-005-8415-x.

[97] I. Pagonabarraga and D. Frenkel. Dissipative particle dynamics for interacting systems. *Journal of Chemical Physics*, 115(11):5015–5026, 2001. doi:10.1063/1.1396848.

[98] Tasos C. Papanastasiou. Flows of materials with yield. *Journal of Rheology*, 31(5):385–404, 1987. doi:10.1122/1.549926.

[99] Ding-Yu Peng and Donald B. Robinson. A new two-constant equation of state. *Industrial & Engineering Chemistry Fundamentals*, 15(1):59–64, 1976. doi:10.1021/i160057a011.

[100] E. A. J. F. Peters. Elimination of time step effects in DPD. *EPL (Europhysics Letters)*, 66(3):311–317, May 2004. doi:10.1209/epl/i2004-10010-4.

[101] B. Piaud, M.J. Clifton, S. Blanco, and R. Fournier. Lattice Boltzmann method for colloidal dispersions with phase change. *Progress in Computational Fluid Dynamics*, 8(1–4):129–137, 2008. doi:10.1504/PCFD.2008.018094.

[102] C. M. Pooley and K. Furtado. Eliminating spurious velocities in the free-energy lattice Boltzmann method. *Physical Review E*, 77:046702, April 2008. doi:10.1103/PhysRevE.77.046702.

[103] C. M. Pooley, H. Kusumaatmaja, and J. M. Yeomans. Contact line dynamics in binary lattice Boltzmann simulations. *Physical Review E*, 78:056709, November 2008. doi:10.1103/PhysRevE.78.056709.

[104] 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. doi:10.1016/j.jcp.2006.10.023.

[105] 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. doi:10.1103/PhysRevLett.89.148302.

[106] Y. H. Qian, S. Succi, and S. A. Orszag. Recent advances in lattice Boltzmann computing. In Dietrich Stauffer, editor, *Annual Reviews of Computational Physics III*, chapter 6, pages 195–242. World Scientific, October 1995. doi:10.1142/9789812830647_0006.

[107] D. Quigley and M. I. J. Probert. Langevin dynamics in constant pressure extended systems. *Journal of Chemical Physics*, 120(24):11432–11441, 2004. doi:10.1063/1.1755657.

[108] P. Raiskinmäki, A Koponen, J. Merikoski, and J. Timonen. Spreading dynamics of three-dimensional droplets by the lattice-Boltzmann method. *Computational Materials Science*, 18(1):7–12, 2000. doi:10.1016/S0927-0256(99)00095-6.

[109] P. Raiskinmäki, A. Shakib-Manesh, A. Jäsberg, A. Koponen, J. Merikoski, and J. Timonen. Lattice-Boltzmann simulation of capillary rise dynamics. *Journal of Statistical Physics*, 107(1–2):143–158, 2002. doi:10.1023/A:1014506503793.

[110] Otto. Redlich and J. N. S. Kwong. On the thermodynamics of solutions. V. An equation of state. Fugacities of gaseous solutions. *Chemical Reviews*, 44(1):233–244, 1949. doi:10.1021/cr60137a013.

[111] M. Revenga, I. Zúñiga, and P. Español. Boundary conditions in dissipative particle dynamics. *Computer Physics Communications*, 121–122:309–311, 1999. doi:10.1016/S0010-4655(99)00341-0.

[112] Jean-Paul Ryckaert, Giovanni Ciccotti, and Herman J. C. Berendsen. Numerical integration of the Cartesian equations of motion of a system with constraints: molecular dynamics of n-alkanes. *Journal of Computational Physics*, 23(3):327–341, 1977. doi:https://doi.org/10.1016/0021-9991(77)90098-5.

[113] 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. doi:10.1122/1.550713.

[114] M. A. Seaton, I. Halliday, and A. J. Masters. Application of the multicomponent lattice Boltzmann simulation method to oil/water dispersions. *Journal of Physics A*, 44(10):105502, March 2011. doi:10.1088/1751-8113/44/10/105502.

[115] Takeshi Seta, Roberto Rojas, Kosuke Hayashi, and Akio Tomiyama. Implicit-correction-based immersed boundary-lattice Boltzmann method with two relaxation times. *Physical Review E*, 89:023307, February 2014. doi:10.1103/PhysRevE.89.023307.

[116] Xiaowen Shan. Analysis and reduction of the spurious current in a class of multiphase lattice Boltzmann models. *Physical Review E*, 73(4):047701, 2006. doi:10.1103/PhysRevE.73.047701.

[117] Xiaowen Shan. Pressure tensor calculation in a class of nonideal gas lattice Boltzmann models. *Physical Review E*, 77(6):066702, June 2008. doi:10.1103/PhysRevE.77.066702.

[118] Xiaowen Shan and Hudong Chen. Lattice Boltzmann model for simulating flows with multiple phases and components. *Physical Review E*, 47(3):1815–1819, March 1993. doi:10.1103/PhysRevE.47.1815.

[119] Xiaowen Shan and Hudong Chen. Simulation of nonideal gases and liquid-gas phase transitions by the lattice Boltzmann equation. *Physical Review E*, 49(4):2941–2948, April 1994. doi:10.1103/PhysRevE.49.2941.

[120] Tony Shardlow. Splitting for dissipative particle dynamics. *SIAM Journal on Scientific Computing*, 24(4):1267–1282, 2003. doi:10.1137/S1064827501392879.

[121] Julian C. Shillcock and Reinhard Lipowsky. Equilibrium structure and lateral stress distribution of amphiphilic bilayers from dissipative particle dynamics simulations. *Journal of Chemical Physics*, 117(10):5048–5061, 2002. doi:10.1063/1.1498463.

[122] Richard C. Singleton. An algorithm for computing the mixed radix fast Fourier transform. *IEEE Transactions on Audio and Electroacoustics*, 17(2):93–103, 1969. doi:10.1109/TAU.1969.1162042.

[123] W. Smith. Coping with the pressure! How to calculate the virial. *CCP5 Information Quarterly*, 26:43–51, September 1987. URL: https://www.ccp5.ac.uk/sites/www.ccp5.ac.uk/files/infoweb/CCP5_Newsletter_1987_09_26.pdf.

[124] W. Smith. Molecular dynamics on hypercube parallel computers. *Computer Physics Communications*, 62(2-3):229–248, March 1991. doi:10.1016/0010-4655(91)90097-5.

[125] W. Smith. A replicated data molecular dynamics strategy for the parallel Ewald sum. *Computer Physics Communications*, 67(3):392–406, January 1992. doi:10.1016/0010-4655(92)90048-4.

[126] W. Smith. Calculating the pressure. *CCP5 Information Quarterly*, 39:14–20, October 1993. URL: https://www.ccp5.ac.uk/sites/www.ccp5.ac.uk/files/infoweb/CCP5_Newsletter_1993_10_39.pdf.

[127] 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.

[128] Giorgio Soave. Equilibrium constants from a modified Redlich-Kwong equation of state. *Chemical Engineering Science*, 27(6):1197–1203, 1972. doi:10.1016/0009-2509(72)80096-4.

[129] 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. doi:10.1103/PhysRevE.82.066701.

[130] Timothy J. Spencer, Ian Halliday, and Chris M. Care. A local lattice Boltzmann method for multiple immiscible fluids and dense suspensions of drops. *Philosophical Transactions of the Royal Society of London A*, 369(1944):2255–2263, 2011. doi:10.1098/rsta.2011.0029.

[131] 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. doi:10.1063/1.1870892.

[132] Alexander Stukowski. Visualization and analysis of atomistic simulation data with OVITO–the Open Visualization Tool. *Modelling and Simulation in Materials Science and Engineering*, 18(1):015012, 2010. doi:10.1088/0965-0393/18/1/015012.

[133] Sauro Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Clarendon Press, Oxford, 2001.

[134] K. Suga, Y. Kuwata, K. Takashima, and R. Chikasue. A D3Q27 multiple-relaxation-time lattice Boltzmann method for turbulent flows. *Computers & Mathematics with Applications*, 69(6):518–529, 2015. doi:10.1016/j.camwa.2015.01.010.

[135] Michael R. Swift, E. Orlandini, W. R. Osborn, and J. M. Yeomans. Lattice Boltzmann simulations of liquid-gas and binary fluid systems. *Physical Review E*, 54(5):5041–5052, November 1996. doi:10.1103/PhysRevE.54.5041.

[136] 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. doi:10.1103/PhysRevLett.75.830.

[137] Ketzasmin A. Terrón-Mejía, Roberto López-Rendón, and Armando Gama Goicochea. Electrostatics in dissipative particle dynamics using Ewald sums with point charges. *Journal of Physics: Condensed Matter*, 28(42):425101, 2016. doi:10.1088/0953-8984/28/42/425101.

[138] B. D. Todd, Denis J. Evans, and Peter J. Daivis. Pressure tensor for inhomogeneous fluids. *Physical Review E*, 52:1627–1638, August 1995. doi:10.1103/PhysRevE.52.1627.

[139] 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.

[140] 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. doi:10.1063/1.1515774.

[141] Loup Verlet. Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Physical Review*, 159(1):98–103, July 1967. doi:10.1103/PhysRev.159.98.

[142] P. B. Warren. Vapor-liquid coexistence in many-body dissipative particle dynamics. *Physical Review E*, 68(6):066702, December 2003. doi:10.1103/PhysRevE.68.066702.

[143] P. B. Warren, P. Prinsen, and M. A. J. Michels. The physics of surfactant dissolution. *Philosophical Transactions of the Royal Society of London A*, 361(1805):665–676, 2003. doi:10.1098/rsta.2002.1166.

[144] Patrick B. Warren. No-go theorem in many-body dissipative particle dynamics. *Physical Review E*, 87:045303, April 2013. doi:10.1103/PhysRevE.87.045303.

[145] Patrick B. Warren and Andrey Vlasov. Screening properties of four mesoscale smoothed charge models, with application to dissipative particle dynamics. *Journal of Chemical Physics*, 140(8):084904, 2014. doi:http://dx.doi.org/10.1063/1.4866375.

[146] Patrick B. Warren, Andrey Vlasov, Lucian Anton, and Andrew J. Masters. Screening properties of Gaussian electrolyte models, with application to dissipative particle dynamics. *Journal of Chemical Physics*, 138(20):204907, 2013. doi:10.1063/1.4807057.

[147] 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. doi:10.1063/1.1674820.

[148] Dean R. Wheeler, Norman G. Fuller, and Richard L. Rowley. 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. doi:10.1080/002689797170608.

[149] B. Widom. Some topics in the theory of fluids. *Journal of Chemical Physics*, 39(11):2808–2812, 1963. doi:10.1063/1.1734110.

[150] B. Widom. Potential-distribution theory and the statistical mechanics of fluids. *Journal of Physical Chemistry*, 86(6):869–872, 1982. doi:10.1021/j100395a005.

[151] 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. doi:10.1063/1.1456031.

[152] Kenji Yasuda. *Investigation of the analogies between viscometric and linear viscoelastic properties of polystyrene fluids*. PhD thesis, Massachusetts Institute of Technology, 1979. URL: http://hdl.handle.net/1721.1/16043.

[153] In-Chul Yeh and Max L. Berkowitz. Ewald summation for systems with slab geometry. *Journal of Chemical Physics*, 111(7):3155–3162, 1999. doi:10.1063/1.479595.

[154] 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. doi:10.1002/fld.607.

[155] Peng Yuan and Laura Schaefer. Equations of state in a lattice Boltzmann model. *Physics of Fluids*, 18(4):042101, 2006. doi:10.1063/1.2187070.

[156] Raoyang Zhang and Hudong Chen. Lattice Boltzmann method for simulations of liquid-vapor thermal flows. *Physical Review E*, 67(6):066711, June 2003. doi:10.1103/PhysRevE.67.066711.

[157] You-Liang Zhu, Hong Liu, Zhan-Wei Li, Hu-Jun Qian, Giuseppe Milano, and Zhong-Yuan Lu. GALAMOST: GPU-accelerated large-scale molecular simulation toolkit. *Journal of Computational Chemistry*, 34(25):2197–2211, 2013. doi:10.1002/jcc.23365.

[158] 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. doi:10.1063/1.869307.

[159] 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 A*, 360(1792):437–451, 2002. doi:10.1098/rsta.2001.0955.