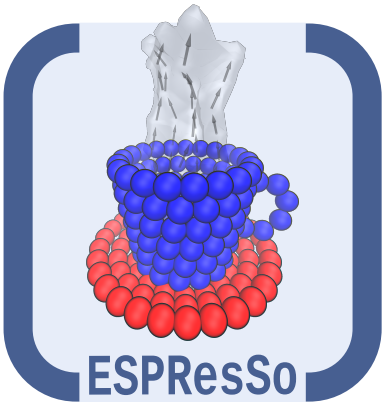



# ESPResSo User's Guide

for version

June 30, 2016

# Todo list


Integrator, once the interface for that is clear . . . . . 18
What is the syntax of variant 3? . . . . . 38
Better throw some out (*e.g.* switches)? . . . . . 100
Missing: lattice_switch, dpd_tgamma, n_rigidbonds . . . . . 100
Which commands can be used to set the *read-only* variables? . . . . . 100
Docs missing. . . . . 101
Docs missing. . . . . 101
Add a comment to the UG that the analysis routines in ESPResSo with Python currently require the user to 'from espressomd import analyze.' . . . . . 131
Document the usage! . . . . . 135
Document the usage! . . . . . 136
Document the usage and what it is! . . . . . 136
I think there is still a bug in there (Hanjo) . . . . . 141
Describe the different energies components returned by the different commands! . 142
Document arguments nb_inter, nb_intra, tot_nb_inter and tot_nb_intra . . . . . 142
Description of how electrostatic contribution to Pressure is calculated . . . . . 143
Check this! . . . . . 148
Missing descriptions of parameters of several observables . . . . . 154
any suggestion for a more suitable name? . . . . . 155
Formatted printing is not fully supported yet. . . . . 158
Does not work yet . . . . . 158
Processing data from Tcl input or from input files is not fully supported yet. . . . 160
Complex conjugate product must be defined. . . . . 161
Maybe not all parameters are printed. . . . . 162
I do not understand this. How does the error look? . . . . . 186
Missing commands: Probably all from `scripts/auxiliary.tcl`? . . . . . 187
Complete in broad strokes the applicability of the electrokinetics model. Also mention the difference in temperatures between EK and LB species. . . . . 204
At the moment this fails badly, if you try to parse incorrectly formatted files. This will be fixed in the future. . . . . 212
The list contains all features, but there are tons of docs missing! . . . . . 267
Docs missing . . . . . 268
How to use it? . . . . . 268
Documentation! . . . . . 269
BOND_ANGLEDIST and BOND_ENDANGLEDIST are completely undocumented. 271
Cleanup: References, mathematics . . . . . 281

# Contents


**1 Introduction** **9**
- 1.1 Guiding principles . . . 9
- 1.2 Available simulation methods . . . 10
- 1.3 Basic program structure . . . 12
- 1.4 On units . . . 12
- 1.5 Requirements . . . 14
- 1.6 Tcl: Syntax description . . . 14

**2 First steps** **15**
- 2.1 Quick installation . . . 15
- 2.2 Running ESPResSo . . . 16
- 2.3 Python: Basic concepts . . . 16
- 2.4 Tcl: Creating the first simulation script . . . 18
- 2.5 Tcl: `tutorial.tcl` . . . 24

**3 Getting, compiling and running ESPResSo** **25**
- 3.1 cmake . . . 25
- 3.2 Autotools . . . 26
- 3.3 `make`: Compiling, testing and installing ESPResSo . . . 29
- 3.4 Running ESPResSo . . . 30
- 3.5 `myconfig.hpp`: Activating and deactivating features . . . 30

**4 Setting up particles** **32**
- 4.1 `part`: Creating single particles . . . 32
- 4.2 Creating groups of particle . . . 39
- 4.3 `constraint`: Setting up constraints . . . 46
- 4.4 Virtual sites . . . 50
- 4.5 Grand canonical feature . . . 53

**5 Setting up interactions** **55**
- 5.1 Isotropic non-bonded interactions . . . 55
- 5.2 Anisotropic non-bonded interactions . . . 63
- 5.3 Bonded interactions . . . 65
- 5.4 Object-in-fluid interactions . . . 70
- 5.5 Bond-angle interactions . . . 74

5.6 Dihedral interactions . . . 76
5.7 Coulomb interaction . . . 77
5.8 Dipolar interaction . . . 92
5.9 Special interaction commands . . . 96

**6 Setting up the system 100**
6.1 `setmd`: Setting global variables. . . . 100
6.2 `thermostat`: Setting up the thermostat . . . 102
6.3 `nemd`: Setting up non-equilibrium MD . . . 109
6.4 `cellsystem`: Setting up the cell system . . . 110
6.5 CUDA . . . 111
6.6 Creating bonds when particles collide . . . 112
6.7 Catalytic Reactions . . . 114
6.8 Galilei Transform and Particle Velocity Manipulation . . . 116

**7 Running the simulation 118**
7.1 `integrate`: Running the simulation . . . 118
7.2 `time_integration`: Runtime of the integration loop . . . 119
7.3 `minimize_energy`: Run steepest descent minimization . . . 119
7.4 `tune_skin`: Tune the skin . . . 120
7.5 `change_volume`: Changing the box volume . . . 120
7.6 `rotate_system`: Rotating the system around its center of mass . . . 121
7.7 `lees_edwards_offset`: Applying shear between periodic images . . . 121
7.8 Stopping particles . . . 122
7.9 `velocities`: Setting the velocities . . . 122
7.10 Fixing the particle sorting . . . 123
7.11 Parallel tempering . . . 123
7.12 Metadynamics . . . 127
7.13 `integrate_sd`: Running a stokesian dynamics simulation . . . 129
7.14 Multi-timestepping . . . 130

**8 Analysis in Tcl 131**
8.1 Available observables . . . 131
8.2 Analyzing groups of particles (molecules) . . . 145
8.3 Storing configurations . . . 150
8.4 `uwerr`: Computing statistical errors in time series . . . 151

**9 Analysis in the core 153**
9.1 Observables . . . 153
9.2 Correlations . . . 160

**10 Input / Output 167**
10.1 No generic checkpointing! . . . 167
10.2 Writing H5MD-files . . . 168

10.3 Writing H5MD-files with Python . . . 170
10.4 Reading H5MD-files with Python . . . 172
10.5 `blockfile`: Using the structured file format . . . 173
10.6 Writing and reading binary files . . . 177
10.7 MPI-IO . . . 177
10.8 Writing VTF files . . . 178
10.9 `writevtk`: Particle Visualization in paraview . . . 181
10.10 Reading and Writing PDB/PSF files . . . 181
10.11 Online-visualisation with VMD . . . 183
10.12 Error handling . . . 185

**11 Auxilliary commands** **187**
11.1 Finding particles and bonds . . . 187
11.2 Additional Tcl math-functions . . . 188
11.3 Checking for features of ESPResSo . . . 193

**12 Lattice-Boltzmann** **194**
12.1 Setting up a LB fluid . . . 194
12.2 LB as a thermostat . . . 196
12.3 The Shan Chen bicomponent fluid . . . 197
12.4 SC as a thermostat . . . 198
12.5 SC component-dependent interactions between particles . . . 198
12.6 Reading and setting single lattice nodes . . . 199
12.7 Removing total fluid momentum . . . 200
12.8 Visualization . . . 200
12.9 Setting up boundary conditions . . . 200
12.10 Choosing between the GPU and CPU implementations . . . 201
12.11 Electrohydrodynamics . . . 202

**13 Electrokinetics** **203**
13.1 Electrokinetic Equations . . . 203
13.2 Setup . . . 205
13.3 Output . . . 207
13.4 Checkpointing . . . 208
13.5 Catalytic Reaction . . . 209

**14 Object-in-fluid** **213**
14.1 Membranes . . . 214
14.2 Parameters . . . 214
14.3 Geometry . . . 214
14.4 Available commands . . . 216

**15 Immersed Boundary Method for soft elastic objects** **222**

**16 External package: mbtools** **224**

16.1 Introduction . . . 224
16.2 Installing and getting started . . . 225
16.3 The `main.tcl` script . . . 226
16.4 Analysis . . . 228
16.5 System generation . . . 232
16.6 Utils . . . 239
16.7 mmsg . . . 245

**17 Under the hood** **248**
17.1 Internal particle organization . . . 248

**18 Getting involved** **250**
18.1 Community support and mailing lists . . . 250
18.2 Contributing your own code . . . 251
18.3 Developers' guide . . . 251
18.4 User's guide . . . 251

**A ESPResSo quick reference** **252**

**B Features** **267**
B.1 General features . . . 267
B.2 Interactions . . . 270
B.3 Debug messages . . . 271

**C Sample scripts** **274**

**D Maxwell Equations Molecular Dynamics (MEMD)** **275**
D.1 Equations of motion . . . 275
D.2 Discretization . . . 276
D.3 Initialization of the algorithm . . . 276
D.4 Time integrator . . . 277
D.5 Self–energy . . . 278
D.6 For which systems to use the algorithm . . . 279

**E The MMM family of algorithms** **281**
E.1 Introduction . . . 281
E.2 MMM2D . . . 283
E.3 MMM1D . . . 285
E.4 ELC . . . 286
E.5 Errors . . . 287

**F Bibliography** **289**

# 1. Introduction

ESPResSo is a simulation package designed to perform Molecular Dynamics (MD) and Monte Carlo (MC) simulations. It is meant to be a universal tool for simulations of a variety of soft matter systems. ESPResSo features a broad range of interaction potentials which opens up possibilities for performing simulations using models with different levels of coarse-graining. It also includes modern and efficient algorithms for treatment of electrostatics (P3M, MMM-type algorithms, Maggs algorithm, ...), hydrodynamic interactions (DPD, Lattice-Boltzmann), and magnetic interactions. It is designed to exploit the capabilities of parallel computational environments. The program is being continuously extended to keep the pace with current developments both in the algorithms and software.

The kernel of ESPResSo is written in C with computational efficiency in mind. Interaction between the user and the simulation engine is provided via a Tcl scripting interface. This enables setup of arbitrarily complex systems which users might want to simulate in future, as well as modifying simulation parameters during runtime.

## 1.1. Guiding principles

ESPResSo is a tool for performing computer simulation and this user guide describes how to use this tool. However, it should be borne in mind that being able to operate a tool is not sufficient to obtain physically meaningful results. It is always the responsibility of the user to understand the principles behind the model, simulation and analysis methods he is using. ESPResSo will *not* do that for you!

It is expected that the users of ESPResSo and readers of this user guide have a thorough understanding of simulation methods and algorithms they are planning to use. They should have passed a basic course on molecular simulations or read one of the renown textbooks, *e.g.* [26]. It is not necessary to understand everything that is contained in ESPResSo, but it is inevitable to understand all methods that you want to use. Using the program as a black box without proper understanding of the background will most probably result in wasted user and computer time with no useful output.

To enable future extensions, the functionality of the program is kept as general as possible. It is modularized, so that extensions to some parts of the program (*e.g.* implementing a new potential) can be done by modifying or adding only few files, leaving most of the code untouched.

To facilitate the understanding and the extensibility, much emphasis is put on readability of the code. Hard-coded assembler loops are generally avoided in hope that the overhead in computer time will be more than compensated for by saving much of the user time while trying to understand what the code is supposed to do.

Hand-in-hand with the extensibility and readability of the code comes the flexibility of the whole program. On the one hand, it is provided by the generalized functionality of its parts, avoiding highly specialized functions. An example can be the implementation of the Generic Lennard-Jones potential described in section 5.1.3 where the user can change all available parameters. Where possible, default values are avoided, providing the user with the possibility of choice. ESPResSo cannot be aware whether your particles are representing atoms or billiard balls, so it cannot check if the chosen parameters make sense and it is the user's responsibility to make sure they do.

On the other hand, flexibility of ESPResSo stems from the employment of a scripting language at the steering level. Apart from the ability to modify the simulation and system parameters at runtime, many simple tasks which are not computationally critical can be implemented at this level, without even touching the C-kernel. For example, simple problem-specific analysis routines can be implemented in this way and made interact with the simulation core. Another example of the program's flexibility is the possibility to integrate system setup, simulation and analysis in one single control script. ESPResSo provides commands to create particles and set up interactions between them. Capping of forces helps prevent system blow-up when initially some particles are placed on top of each other. Using the Tcl interface, one can simulate the randomly set-up system with capped forces, interactively check whether it is safe to remove the cap and switch on the full interactions and then perform the actual productive simulation.

## 1.2. Available simulation methods

ESPResSo provides a number of useful methods. The following table shows the various methods as well as their status. The table distinguishes between the state of the development of a certain feature and the state of its use. We distinguish between five levels:

**Core** means that the method is part of the core of ESPResSo, and that it is extensively developed and used by many people.

**Good** means that the method is developed and used by independent people from different groups.

**Group** means that the method is developed and used in one group.

**Single** means that the method is developed and used by one person only.

**None** means that the method is developed and used by nobody.

If you believe that the status of a certain method is wrong, please report so to the developers.

| Feature | Development Status | Usage Status |
|---|---|---|
| **Integrators, Thermostats, Barostats** | | |
| Velocity-Verlet Integrator | Core | Core |
| Langevin Thermostat | Core | Core |
| GHMC Thermostat | Single | Single |
| DPD Thermostat | None | Good |
| Isotropic NPT | None | Single |
| NEMD | None | Group |
| Quarternion Integrator | None | Good |
| **Interactions** | | |
| Short-range Interactions | Core | Core |
| Directional Lennard-Jones | Single | Single |
| Gay-Berne Interaction | None | Single |
| Constraints | Core | Core |
| Relative Virtual Sites | Good | Good |
| Center-of-mass Virtual Sites | None | Good |
| RATTLE Rigid Bonds | None | Group |
| **Coulomb Interaction** | | |
| P3M | Core | Core |
| P3M on GPU | Single | Single |
| Dipolar P3M | Group | Good |
| Ewald on GPU | Single | Single |
| MMM1D | Single | Good |
| MMM2D | Single | Good |
| MMM1D on GPU | Single | Single |
| ELC | Good | Good |
| MEMD | Single | Group |
| ICC* | Group | Group |
| **Hydrodynamic Interaction** | | |
| Lattice-Boltzmann | Core | Core |
| Lattice-Boltzmann on GPU | Group | Core |
| DPD | None | Good |
| Shan-Chen Multicomponent Fluid | Group | Group |
| Tunable Slip Boundary | Single | Single |
| Stokesian Dynamics | Single | Single |
| **Analysis** | | |
| uwerr | None | Good |
| **Input/Output** | | |
| Blockfiles | Core | Core |
| VTF output | Core | Core |
| VTK output | Group | Group |
| PDB output | Good | Good |
| Online visulation with VMD | Good | Good |

| Miscellaneous | | |
|---|---|---|
| Grand canonical feature | Single | Single |
| Metadynamics | Single | Single |
| Parallel Tempering | Single | Single |
| Electrokinetics | Group | Group |
| Object-in-fluid | Group | Group |
| Collision Detection | Group | Group |
| Catalytic Reactions | Single | Single |
| mbtools package | Group | Group |

## 1.3. Basic program structure

As already mentioned, ESPResSo consists of two components. The simulation engine is written in C and C++ for the sake of computational efficiency. The steering or control level is interfaced to the kernel via an interpreter of the Tcl and Python scripting languages. Please be aware that th TCL interface is going to be removed soon and new simulations should use Python as scripting language.

The kernel performs all computationally demanding tasks. Before all, integration of Newton's equations of motion, including calculation of energies and forces. It also takes care of internal organization of data, storing the data about particles, communication between different processors or cells of the cell-system. The kernel is modularized so that basic functions are accessed via a set of well-defined lean interfaces, hiding the details of the complex numerical algorithms.

The scripting interface (Python or Tcl) are used to setup the system (particles, boundary onditions, interactions, ...), control the simulation, run analysis, and store and load results. The user has at hand the full rl3sibili5y and functionality of the scripting language. For instance, it is possible to use the SciPy package for analysis and PyPlot for plotting. With a certain overhead in efficiency, it can also be used to reject/accept new configurations in combined MD/MC schemes. In principle, any parameter which is accessible from the scripting level can be changed at any moment of runtime. In this way methods like thermodynamic integration become readily accessible.

The focus of the user guide is documenting the scripting interfacce, its behaviour and use in the simulation. It only describes certain technical details of implementation which are necessary for understanding how the script interface works. Technical documentation of the code and program structure is contained in the Developers' guide (see section 18.3).

## 1.4. On units

What is probably one of the most confusing subjects for beginners of ESPResSo is, that ESPResSo does not predefine any units. While most MD programs specify a set of units, like, for example, that all lengths are measured in Ångström or nanometers, times are measured in nano- or picoseconds and energies are measured in kJ/mol, ESPResSo does not do so.

Instead, the length-, time- and energy scales can be freely chosen by the user. Once these three scales are fixed, all remaining units are derived from these three basic choices.

The probably most important choice is the length scale. A length of 1.0 can mean a nanometer, an Ångström, or a kilometer - depending on the physical system, that the user has in mind when he writes his ESPResSo-script. When creating particles that are intended to represent a specific type of atoms, one will probably use a length scale of Ångström. This would mean, that *e.g.* the parameter $\sigma$ of the Lennard-Jones interaction between two atoms would be set to twice the van-der-Waals radius of the atom in Ångström. Alternatively, one could set $\sigma$ to 2.0 and measure all lengths in multiples of the van-der-Waals radius. When simulation colloidal particles, which are usually of micrometer size, one will choose their diameter (or radius) as basic length scale, which is much larger than the Ångström scale used in atomistic simulations.

The second choice to be made is the energy scale. One can for example choose to set the Lennard-Jones parameter $\epsilon$ to the energy in kJ/mol. Then all energies will be measured in that unit. Alternatively, one can choose to set it to 1.0 and measure everything in multiples of the van-der-Waals binding energy of the respective particles.

The final choice is the time (or mass) scale. By default, ESPResSo uses a reduced mass of 1, so that the mass unit is simply the mass of all particles. Combined with the energy and length scale, this is sufficient to derive the resulting time scale:

$$[\text{time}] = [\text{length}]\sqrt{\frac{[\text{mass}]}{[\text{energy}]}}.$$

This means, that if you measure lengths in Ångström, energies in $k_BT$ at 300 K and masses in 39.95u, then your time scale is $\mathring{A}\sqrt{39.95u/k_BT} = 0.40\,\text{ps}$.

On the other hand, if you want a particular time scale, then the mass scale can be derived from the time, energy and length scales as

$$[\text{mass}] = [\text{energy}]\frac{[\text{time}]^2}{[\text{length}]^2}.$$

By activating the feature MASSES, you can specify particle masses in the chosen unit system.

A special note is due regarding the temperature, which is coupled to the energy scale by Boltzmann's constant. However, since ESPResSo does not enforce a particular unit system, we also don't know the numerical value of the Boltzmann constant in the current unit system. Therefore, when specifying the temperature of a thermostat, you actually do not define the temperature, but the value of the thermal energy $k_BT$ in the current unit system. For example, if you measure energy in units of kJ/mol and your real temperature should be 300 K, then you need to set the thermostat's effective temperature to $k_B 300\,K\text{mol/kJ} = 2.494$.

As long as one remains within the same unit system throughout the whole ESPResSo-script, there should be no problems.

## 1.5. Requirements

The following libraries and tools are required to be able to compile and use ESPResSo:

**Tcl/Tk** ESPResSo requires the Toolkit Command Language Tcl/Tk [1] in the version 8.3 or later. Some example scripts will only work with Tcl 8.4. You do not only need the interpreter, but also the header files and libraries. Depending on the operating system, these may come in separate development packages. If you want to use a graphical user interface (GUI) for your simulation scripts, you will also need Tk.

**FFTW** For some algorithms (*e.g.* P$^3$M), ESPResSo needs the FFTW library version 3 or later [2] for Fourier transforms. Again, the header files are required.

**MPI** Finally, if you want to use ESPResSo in parallel, you need a working MPI environment (that implements the MPI standard version 1.2).

## 1.6. Tcl: Syntax description

Throughout the user's guide, formal definitions of the syntax of several Tcl-commands can be found. The following conventions are used in these descriptions:

- Different *variants* of a command are labeled (1), (2), ...
- Keywords and literals of the command that have to be typed exactly as given are written in `typewriter` font.
- If the command has variable arguments, they are set in *italicfont*. The description following the syntax definition should contain a detailed explanation of the argument and its type.
- ( *alt1* | *alt2* ) specifies, that one of the alternatives *alt1* or *alt2* can be used.
- [*argument*] specifies, that the argument *argument* is optional, *i.e.* it can be omitted.
- When an optional argument or a whole command is marked by a superscript label ([1]), this denotes that the argument can only be used, when the corresponding feature (see appendix B on page 267) specified in "Required features" is activated.

*Example*

(1) `constraint wall normal` $n_x$ $n_y$ $n_z$ `dist` $d$ `type` *id*
(2) `constraint sphere center` $c_x$ $c_y$ $c_z$ `radius` *rad* `direction` *direction* `type` *id*
(3) `constraint rod center` $c_x$ $c_y$ `lambda` *lambda* [1]
(4) `constraint ext_magn_field` $f_x$ $f_y$ $f_z$ [2,3]
Required features: `CONSTRAINTS` [1]`ELECTROSTATICS` [2]`ROTATION` [3]`DIPOLES`

[1] `http://www.tcl.tk/`
[2] `http://www.fftw.org/`

# 2. First steps

## 2.1. Quick installation

If you have installed the requirements (see section 1.5 on the preceding page) in standard locations, to compile ESPResSo, it is usually enough to execute the following sequence of two steps in the directory where you have unpacked the sources:

```
./configure
make
```

This will compile ESPResSo in a freshly created object path named according to your CPU and operating system. As you have not yet specified a configuration, a standard version will be built with the most often used features. Usually you will want to build another version of ESPResSo with options better suited for your purpose.

In some cases, *e.g.* when ESPResSo needs to be compiled for several different platforms or when different versions with different sets of features are required, it might be useful to execute the commands not in the source directory itself, but to start `configure` from another directory (see section 3.2.1 on page 27). Furthermore, many features of ESPResSo can be selectively turned on or off in the local configuration header (see section 3.5 on page 30) before starting the compilation with `make`.

The shell script `configure` prepares the source code for compilation. It will determine how to use and where to find the different libraries and tools required by the compilation process, and it will test what compiler flags are to be used. The script will find out most of these things automatically. If something is missing, it will complain and give hints on how to solve the problem. The configuration process can be controlled with the help of a number of options that are explained in section 3.2.1 on page 27.

The command `make` will compile the source code. Depending on the options passed to the program, `make` can also be used for a number of other things:

- It can install and uninstall the program to some other directories. However, normally it is not necessary to actually *install* ESPResSo to run it.
- It can test ESPResSo for correctness.
- It can build the documentation.

The details of the usage of `make` are described in section 3.3 on page 29.

When these steps have successfully completed, ESPResSo can be started with the command (see section 3.4 on page 30)

`Espresso` *script*

where *script* is a Tcl script that tells ESPResSo what to do, and has to be written by the user. You can find some examples in the `samples` folder of the source code directory. If you want to run in parallel, you should have compiled ESPResSo to use MPI, and need to tell MPI to run ESPResSo in parallel. The actual invocation is implementation dependent, but in many cases, such as OpenMPI, you can use

`mpirun -n` *n_nodes* `Espresso` *script*

where *n_nodes* is the number of prcessors to be used.

## 2.2. Running ESPResSo

### 2.2.1. Python

ESPResSo is implemented as a Python module. This means that you need to write a python script for any task you want to perform with ESPResSo. In this chapter, the basic structure of the interface will be explained. For a practical introduction, see the tutorials, which are also part of the ESPResSo distribution. To use ESPResSo, you need to import the espressomd module in your Python script. To this end, the folder containing the python module needs to be in the Python search path. The module is located in the src/python folder under the build directory. A convenient way to run python with the correct path is to use the pypresso script located in the build directory.

```
pypresso simulation.py
```

### 2.2.2. Tcl (deprecated)

ESPResSo is implemented as an extension to the Tcl scripting language. This means that you need to write a script for any task you want to perform with ESPResSo. To learn about the Tcl script language and especially the ESPResSo extensions, this chapter offers two tutorial scripts. The first will guide you step-by-step through creating your first simulation script, while the second script is a well documented example simulation script. Since the latter is slightly more complex and uses more advanced features of ESPResSo, we recommend to work through both scripts in the presented order. If you want to learn about the Tcl language in greater detail, there is an excellent tutorial [1].

## 2.3. Python: Basic concepts

In this section, a brief overview is given over the most important components of the Python interface and their usage is illustrated by short examples. The interface is contained in the espressomd Python module, which needs to be imported, before anything ESPResSo related can be done.

```
import espressomd
```

[1] `http://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html`

Access to the simulation system is provided via the System class. As a first step, an instance of the class needs to be created

```
system=espressomd.System()
```

Note that only one instance of the System class can be created, due to limitations in the simulation core. Properties of the System class are used to access the parameters concerning the simulation system as a whole, *e.g.*, the box geometry and the time step

```
system.box_l = (10.0,10.0,15.0)
print system.time_step
```

The particles in the simulation are accessed via the ParticleList class. It is used to retrieve individual particles of the simulation as well as for adding particles. An instance of the class is provided as the part attribute of the System class. Individual particles can be retrieved by their numerical id by using angular brackets

```
p=system.part[0]
```

It is also possible to loop over all particles

```
for p in system.part:
    ...
```

Particles are added via the add method

```
p=system.part.add(id=1,pos=(3.0,0.5,1.0),q=1)
```

An individual particle is represented by an instance of ParticleHandle. The properties of the particle are implemented as Python properties:

```
p=system.part[0]
p.pos=(0,0,0)
print p.id,p.pos
system.part[0].q=-1
```

Properties of several particles can be accessed by using Python ranges

```
v=system.part[:].v
```

Interactions between particles fall in three categories:

- Non-bonded interactions are short-ranged interactions between *all* pairs of particles of specified types. An example is the Lennard-Jones interaction mimicking overlap repulsion and van der Wals attraction.

- Bonded interactions act only between two specific particles. An example is the harmonic bond between adjacent particles in a polymer chain.

- Long-range interactions act between all particles with specific properties in the entire system. An example is the coulomb interaction.

Non-bonded interactions are represented as subclasses of NonBondedInteraction, *e.g.* LennardJonesInteraction. Instances of these classes for a given pair of particle types are accessed via the non_bonded_inter attribute of the System class. Parameters are set as follows

```
system.non_bonded_inter[0,0].lennard_jones.set_params(epsilon=1,sigma=1,c
```

Bonded interactions are represented by subclasses of BondedInteraction. To set up a bonded interaction, first an instance of the appropriate class is created with the desired parameters. Then, the bonded interaction is registered with the simulation core. Finally, the bond can be added to particles using the add_bond()-method of ParticleHandle with the instance of the bond class and the id of the bond partner particle.

```
from interactions import HarmonicBond
harmonic=HarmonicBond(k=1,r_0=1)
system.bonded_inter.add(harmonic)
system.part[0].add_bond((harmonic,1))
system.part[1].add_bond((harmonic,2))
```

Long-range interactions are subclasses of Actor. They are used by first creating an instance of the desired actor and then adding it to the system. To activate the P3M electrostatics solver, execute

```
from electrostatics import P3M
p3m=P3M(accuracy=1E-3, bjerrum_length=1)
system.actors.add(p3m)
```

Integrator, once the interface for that is clear

## 2.4. Tcl: Creating the first simulation script

This section introduces some of the features of ESPResSo by constructing step by step a simulation script for a simple salt crystal. We cannot give a full Tcl tutorial here; however, most of the constructs should be self–explanatory. We also assume that the reader is familiar with the basic concepts of a MD simulation here. The code pieces can be copied step by step into a file, which then can be run using `Espresso` ***`file`*** from the ESPResSo source directory.

Our script starts with setting up the initial configuration. Most conveniently, one would like to specify the density and the number of particles of the system as parameters:

```
set n_part 200; set density 0.7
set box_l [expr pow($n_part/$density,1./3.)]
```

These variables do not change anything in the simulation engine, but are just standard Tcl variables; they are used to increase the readability and flexibility of the script. The box length is not a parameter of this simulation; it is calculated from the number of

particles and the system density. This allows to change the parameters later easily, *e.g.* to simulate a bigger system.

The parameters of the simulation engine are modified by the `setmd` command. For example

```
setmd box_l $box_l $box_l $box_l
setmd periodic 1 1 1
```

defines a cubic simulation box of size `box_l`, and periodic boundary conditions in all spatial dimensions. We now fill this simulation box with particles

```
set q 1; set type 0
for {set i 0} { $i < $n_part } {incr i} {
  set posx [expr $box_l*[t_random]]
  set posy [expr $box_l*[t_random]]
  set posz [expr $box_l*[t_random]]
  set q [expr -$q]; set type [expr 1-$type]
  part $i pos $posx $posy $posz q $q type $type
}
```

This loop adds `n_part` particles at random positions, one by one. In this construct, only two commands are not standard Tcl commands: the random number generator `t_random` and the `part` command, which is used to specify particle properties, here the position, the charge `q` and the type. In ESPResSo the particle type is just an integer number which allows to group particles; it does not imply any physical parameters. Here we use it to tag the charges: positive charges have type 0, negative charges have type 1.

Now we define the ensemble that we will be simulating. This is done using the `thermostat` command. We also set some integration scheme parameters:

```
setmd time_step 0.01; setmd skin 0.4
set temp 1; set gamma 1
thermostat langevin $temp $gamma
```

This switches on the Langevin thermostat for the NVT ensemble, with temperature `temp` and friction `gamma`. The skin depth `skin` is a parameter for the link–cell system which tunes its performance, but cannot be discussed here.

Before we can really start the simulation, we have to specify the interactions between our particles. We use a simple, purely repulsive Lennard-Jones interaction to model the hard core repulsion [28], and the charges interact via the Coulomb potential:

```
set sig 1.0; set cut   [expr 1.12246*$sig]
set eps 1.0; set shift [expr 0.25*$eps]
inter 0 0 lennard-jones $eps $sig $cut $shift 0
inter 1 0 lennard-jones $eps $sig $cut $shift 0
inter 1 1 lennard-jones $eps $sig $cut $shift 0
inter coulomb 10.0 p3m tunev2 accuracy 1e-3 mesh 32
```

The first three `inter` commands instruct ESPResSo to use the same purely repulsive Lennard–Jones potential for the interaction between all combinations of the two particle types 0 and 1; by using different parameters for different combinations, one could simulate differently sized particles. The last line sets the Bjerrum length to the value 10, and then instructs ESPResSo to use $P^3M$ for the Coulombic interaction and to try to find suitable parameters for an rms force error below $10^{-3}$, with a fixed mesh size of 32. The mesh is fixed here to speed up the tuning; for a real simulation, one will also tune this parameter.

If we want to calculate the temperature of our system from the kinetic energy, we need to know the number of the degrees of freedom of the particles. In ESPResSo these are usually 3 translational plus 3 rotational degrees of freedom (if the feature ROTATION is activated). You can get this number in the following way [2]:

```
if { [regexp "ROTATION" [code_info]] } {
  set deg_free 6
} else { set deg_free 3 }
```

Now we can integrate the system:

```
set integ_steps 200
for {set i 0} { $i < 20 } { incr i} {
  set temp [expr [analyze energy kinetic]/(($deg_free/2.0)*$n_part)]
  puts "t=[setmd time] E=[analyze energy total], T=$temp"
  integrate $integ_steps
}
```

This code block is the primary simulation loop and runs $20\times$`integ_steps` MD steps. Every `integ_steps` time steps, the potential, electrostatic and kinetic energies are printed out (the latter one as temperature). However, the simulation will crash: ESPResSo complains about particle coordinates being out of range. The reason for this is simple: Due to the initial random setup, the overlap energy is around a million kT, which we first have to remove from the system. In ESPResSo, this is can be accelerated by capping the forces, i. e. modifying the Lennard–Jones force such that it is constant below a certain distance. Before the integration loop, we therefore insert this equilibration loop:

```
for {set cap 20} {$cap < 200} {incr cap 20} {
  puts "t=[setmd time] E=[analyze energy total]"
  inter forcecap $cap; integrate $integ_steps
}
inter forcecap 0
```

This loop integrates the system with a force cap of initially 20 and finally 200. The last command switches the force cap off again. With this equilibration, the simulation script runs fine.

However, it takes some time to simulate the system, and one will probably like to

[2] There also exists a Tcl function `degrees_of_freedom` which does the same.

write out simulation data to configuration files, for later analysis. For this purpose ESPResSo has commands to write simulation data to a Tcl stream in an easily parsable form. We add the following lines at end of integration loop to write the configuration files "config_0" through "config_19":

```
set f [open "config_$i" "w"]
blockfile $f write tclvariable {box_l density}
blockfile $f write variable box_l
blockfile $f write particles {id pos type}
close $f
```

The created files "config_..." are human–readable and look like

```
{tclvariable
        {box_l 10}
        {density 0.7}
}
{variable  {box_l 10.0 10.0 10.0} }
{particles {id pos type}
        {0 3.51770181433 4.3208975936 5.30529948918 0}
        {1 3.93145531704 6.58506447035 6.95045147034 1}
        ...
}
```

As you can see, such a *blockfile* consists of several Tcl lists, which are called *blocks*, and can store any data available from the simulation. Reading a configuration is done by the following simple script:

```
set f [open $filename "r"]
while { [blockfile $f read auto] != "eof" } {}
close $f
```

The `blockfile read auto` commands will set the Tcl variables `box_l` and `density` to the values specified in the file when encountering the `tclvariable` block, and set the box dimensions for the simulation when encountering the `variable` block. The particle positions and types of all 216 particles are restored when the `particles` block is read. Note that it is important to have the box dimensions set before reading the particles, to avoid problems with the periodic boundary conditions.

With these configurations, we can now investigate the system. As an example, we will create a second script which calculates the averaged radial distribution functions $g_{++}(r)$ and $g_{+-}(r)$. The radial distribution function for a the current configuration can be obtained using the `analyze` command:

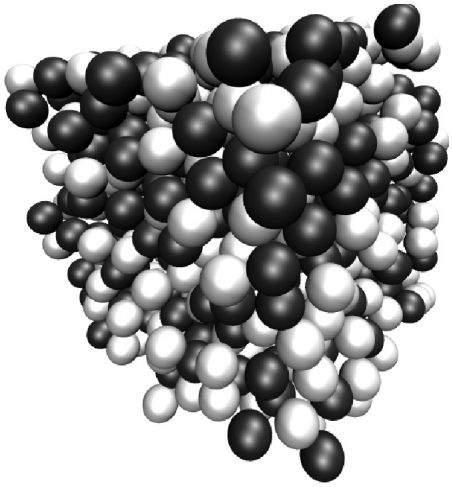


Figure 2.1.: VMD Snapshot of the salt system

```
set rdf [analyze rdf 0 1 0.9 [expr $box_l/2] 100]
set rlist ""
set rdflist ""
foreach value [lindex $rdf 1] {
  lappend rlist   [lindex $value 0]
  lappend rdflist [lindex $value 1]
}
```

The shown `analyze rdf` command returns the distribution function of particles of type 1 around particles of type 0 (i. e. of opposite charges) for radii between 0.9 and half the box length, subdivided into 100 bins. Changing the first two parameters to either "0 0" or "1 1" allows to determine the distribution for equal charges. The result is a list of $r$ and $g(r)$ pairs, which the following foreach loop divides up onto two lists `rlist` and `rdflist`.

To average over a set of configurations, we put the two last code snippets into a loop like this:

```
set cnt 0
for {set i 0} {$i < 100} {incr i} { lappend avg_rdf 0}
foreach filename $argv {
  set f [open $filename "r"]
  while { [blockfile $f read auto] != "eof" } {}
  close $f
  set rdf [analyze rdf 0 1 0.9 [expr $box_l/2] 100]
  set rlist ""
  set rdflist ""
  foreach value [lindex $rdf 1] {
     lappend rlist   [lindex $value 0]
     lappend rdflist [lindex $value 1] }
  set avg_rdf [vecadd $avg_rdf $rdflist]
  incr cnt
}
set avg_rdf [vecscale [expr 1.0/$cnt] $avg_rdf]
```

Initially, the sum of all $g(r)$, which is stored in `avg_rdf`, is set to 0. Then the loops over all configurations given by `argv`, calculates $g(r)$ for each configuration and adds up all the $g(r)$ in `avg_rdf`. Finally, this sum is normalized by dividing by the number of configurations. Note the "1.0/$cnt"; this is necessary, since "1/$cnt" is interpreted as an integer division, which results in 0 for cnt $> 1$. `argv` is a predefined variable: it contains all the command line parameters. Therefore this script should be called like

`Espresso` *script* [*config*`...` ]

The printing of the calculated radial distribution functions is simple. Add to the end of the previous snippet the following lines:

```
set plot [open "rdf.data" "w"]
puts $plot "\# r rdf(r)"
foreach r $rlist rdf $avg_rdf { puts $plot "$r $rdf" }
close $plot
```

This instructs the Tcl interpreter to write the `avg_rdf` to the file `rdf.data` in gnuplot–compatible format. Fig. 2.2 shows the resulting radial distribution functions, averaged over 100 configurations. In addition, the distribution for a neutral system is given, which can be obtained from our simulation script by simply removing the command `inter coulomb ...` and therefore not turning on $P^3M$.

The code example given before is still quite simple, and the reader is encouraged to try to extend the example a little bit, e. g. by using differently sized particle, or changing the interactions. If something does not work, ESPResSo will give comprehensive error messages, which should make it easy to identify mistakes. For real simulations, the simulation scripts can extend over thousands of lines of code and contain automated adaption of parameters or online analysis, up to automatic generation of data plots. Parameters can be changed arbitrarily during the simulation process, as needed for e. g. simulated annealing. The possibility to perform non–standard simulations without the

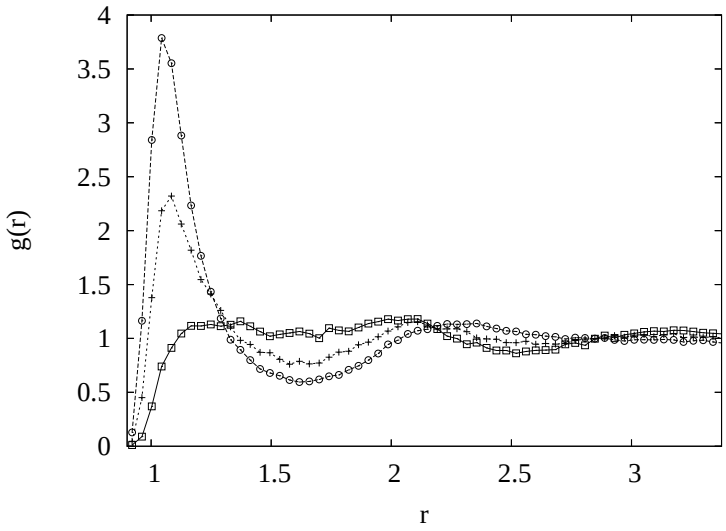



Figure 2.2.: Radial distribution functions $g_{++}(r)$ between equal charges (rectangles) and $g_{+-}(r)$ for opposite charges (circles). The plus symbols denote $g(r)$ for an uncharged system.

need of modifications to the simulation core was one of the main reasons why we decided to use a script language for controlling the simulation core.

## 2.5. Tcl: `tutorial.tcl`

In the directory `samples/` of the es sources, you will find a well documented simulation script `tutorial.tcl`, which takes you step by step through a slightly more complicated simulation of a polyelectrolyte system. The basic structure of the script is however the same as in the previous example and probably the same as the structure of most ESPResSo simulation scripts.

Initially, some parameters and global variables are set, the interactions are initialized, and particles are added. For this, the script makes use of the `polymer` command, which provides a faster way to set up chain molecules.

The actual simulation falls apart again into two loops, the warmup loop with increasing force capping, and the final simulation loop. Note that the electrostatic interaction is only activated after equilibrating the excluded volume interactions, which speeds up the warmup phase. However, depending on the problem, this splitted warmup may not be possible due to physical restrictions. ESPResSo cannot detect these mistakes and it is your responsibility to find simulation procedure suitable to your specific problem.

# 3. Getting, compiling and running ESPResSo

This chapter will describe how to get, compile and run the ESPResSo software.

ESPResSo releases are available as source code packages from the ESPResSo home page[1]. This is where new users should get the code. The code within release packages is tested and known to run on a number of platforms. Alternatively, people that want to use the newest features of ESPResSo or that want to start contributing to the software can instead obtain the current development code via the version control system software git[2] from ESPResSo's project page at Github [3]. This code might be not as well tested and documented as the release code; it is recommended to use this code only if you have already gained some experience in using ESPResSo.

Unlike most other software, no binary distributions of ESPResSo are available, and the software is usually not installed globally for all users. Instead, users of ESPResSo should compile the software themselves. The reason for this is that it is possible to activate and deactivate various features before compiling the code. Some of these features are not compatible with each other, and some of the features have a profound impact on the performance of the code. Therefore it is not possible to build a single binary that can satisfy all needs. F or performance reasons a user should always activate only those features that are actually needed. This means, however, that learning how to compile ESPResSo is a necessary evil. The build system of ESPResSo uses either the GNU autotools or cmake to compile software easily on a wide range of platforms.

## 3.1. cmake

In order to build ESPResSo the first step is to create a build directory in which cmake can be executed. In cmake, the *source directory* (that contains all the source files) is completely separated from the *build directory* (where the files created by the build process are put). cmake is designed to not be executed in the source directory. Cmake will determine how to use and where to find the compiler, as well as the different libraries and tools required by the compilation process. By having multiple build directories you can build several variants of ESPResSo, each variant having different activated features, and for as many platforms as you want.

[1] `http://espressomd.org`
[2] `http://git.org`
[3] `https://github.com/espressomd/espresso`

**Example** When the source directory is `srcdir` (*i.e.* the files where unpacked to this directory), then the user can create a build directory `build` below that path by calling the `mkdir srcdir/build`. In the build direcotry cmake is to be executed, followed by a call of make. None of the files in the source directory is ever modified when by the build process.

```
cd build
cmake srcdir
make
Espresso
```

Afterwards Espresso can be run via calling Espresso from the command line.

### 3.1.1. Options and Variables

The behaviour of `cmake` can be controlled by the means of options and variables in the CMakeLists.txt file. Also options are defined there. The following options are available:

WITH_PYTHON: Build python interface

WITH_TCL: Build tcl interface

WITH_CUDA: Build with GPU support

WITH_TESTS: Enable tests

WITH_SCAFACOS: Build with Scafacos support

WITH_VALGRIND_INSTRUMENTATION: Build with valgrind instrumentation markers

When the value in the CMakeLists.txt file is set to ON the corresponding option is created if the value of the opition is set to OFF the corresponding option is not created. These options can also be modified by calling cmake with the command line argument -D:

```
cmake -D WITH_TCL=OFF srcdir
```

In the rare event when working with cmake and you want to have a totally clean build (for example because you switched the compiler), remove the build directory and create a new one.

## 3.2. Autotools

**Warning:** Deprecated

### 3.2.1. Running `configure`

The first step of building ESPResSo is to run the shell script `configure` which is to be found in the top level source directory. The script collects all the information required by the compilation process. It will determine how to use and where to find the compiler, as well as the different libraries and tools required by the compilation process, and it will test what compiler flags are to be used. The script will find out about most of these things automatically. If something is missing, it will complain and give hints how to solve the problem. The generic syntax of calling the `configure` script is:

`configure [`*options* `...] [`*variable*`=`*value* `...]`

If you are using the development source code from the git repository, before you can call `configure`, it is necessary to have the GNU autotools (autoconf and automake) installed. Then you can call the script `bootstrap.sh` from the top level source directory, which will generate the `configure` script.

#### Source and build directories

Usually, when a program is compiled, the resulting binary files are put into the same directory as the sources of the program. In ESPResSo's build system, the *source directory* that contains all the source files can be completely separated from the *build directory*, where the files created by the build process are put. The location of the build directory is the current working directory at the time when `configure` is called. In this way, you can build several variants of ESPResSo, each variant having different activated features, and for as many platforms as you want. All further commands concerning compiling and running ESPResSo have to be called from the build directory. None of the files in the source directory is ever modified when by the build process.

**Example** When the source directory is `$srcdir` (*i.e.* the files where unpacked to this directory), then the build directory can be set to `$builddir` by calling the `configure`-script from there:

```
cd $builddir
$srcdir/configure
make
Espresso
```

#### Options and Variables

The behaviour of `configure` can be controlled by the means of command line options and variables. In the following, only important command line options and variables ESPResSo will be explained. For a complete list of options, variables and explanations thereof, call

```
configure --help
```

`--with-mpi=( yes | no | guess )/ --without-mpi` By default, `configure` will automatically determine whether an MPI compiler is available. If it is, it will use it. If you specify `--without-mpi` or `--with-mpi=no`, then MPI will not be used, even if it is available.

`--with-efence / --without-efence` Whether or not to use the "electric fence" memory debugging library. [4] Efence is not used by default.

`--with-tcl=TCL` By default, `configure` will automatically determine which version of Tcl is used. If the wrong version is chosen automatically, you can specify the name of the library with this option, *e.g.* `tcl8.4`.

`--with-tk=TK / --without-tk` By default, the GUI toolkit Tk is not used by ESPResSo. This option can be used to activate Tk and to specify which Tk version to use, *e.g.* `tk8.4`. If you only specify `--with-tk` and do not give a version number, `configure` will try to automatically deduce the right version.

`--with-fftw / --without-fftw` This can be used to specify whether the FFTW library is to be used, and which version. By default, version 3 will be used if it is found, otherwise version 2 is used. Note that quite a number of central features of ESPResSo require FFTW.

`--with-cuda=path / --without-cuda` This switch enables CUDA support. `path` should be the path to the CUDA directory, which can be omitted if it is the NVIDIA default path, *i.e.* `/usr/local/cuda`. The variable `NVCCFLAGS` can be used to define compiler flags for the NVIDIA CUDA-compiler `nvcc`. For example, `NVCCFLAGS = "-gencode arch=compute_20,code=sm_20"` will compile code only for Fermi cards. Default is to compile for compute model 2.0, i.e. everything with a Fermi chip or newer. Note that we require at least compute model 1.1, that is G90. However, to use G90 (e. g. Tesla C1060), you need to manually specificy compute model 1.1.

`--with-arpack=path / --without-arpack` This switch enables ARPACK support. `path` should be the path to the ARPACK directory. ARPACK is currently only needed for Stokesian Dynamics support.

`LDFLAGS=`*`linker-flags`* This variable can be used to change the flags that the linker will get when linking the ESPResSo binaries. This variable can be used to modify the path where the compiler finds library files when they are installed in non-standard places, *e.g.* `LDFLAGS="-L/home/juser/lib"`.

`CPPFLAGS=`*`preprocessor-flags`* This variable can be used to change the flags that the preprocessor will see. This variable can be used to modify the path wherer the compiler finds include files when they are installed in non-standard places, *e.g.* `CPPFLAGS="-I/home/juser/include"`.

[4] `http://freshmeat.net/projects/efence/`

`CXXFLAGS=`*`C++-compiler flags`* This variable can be used to modify the compilation flags, *e.g.* to change the optimization level for debugging (`CXXFLAGS="-g -O0"`).

## 3.3. `make`: Compiling, testing and installing ESPResSo

The command `make` is mainly used to compile the ESPResSo source code, but it can do a number of other things. The generic syntax of the `make` command is:

`make [`*options*`] [`*target*`...] [`*variable*`=`*value*`]`

When no target is given, the target `all` is used. The following targets are available:

`all` Compiles the complete ESPResSo source code. The variable `myconf` can be used to specify the name of the configuration header to be used.

`check` Runs the testsuite. By default, all available tests will be run on 1, 2, 3, 4, 6, or 8 processors. Which tests are run can be controlled by means of the variable `tests`, which processor numbers are to be used can be controlled via the variable `processors`. Note that depending on your MPI installation, MPI jobs can only be run in the queueing system, so that ESPResSo will not run from the command line. In that case, you may not be able to run the testsuite, or you have to directly submit the testsuite script `testsuite/test.sh` to the queueing system.
**Example:** `make check tests="madelung.tcl" processors="1 2"`
will run the test `madlung.tcl` on one and two processors.

`clean` Deletes all files that were created during the compilation.

`mostlyclean` Deletes most files that were created during the compilation. Will keep for example the built doxygen documentation and the ESPResSo binary.

`dist` Creates a `.tar.gz`-file of the ESPResSo sources. This will include all source files as they currently are in the source directory, *i.e.* it will include local changes. This is useful to give your version of ESPResSo to other people. The variable `extra` can be used to specify additional files and directories that are to be included in the archive file.
**Example:** `make dist extra="myconfig.hpp internal"`
will create the archive file and include the file `myconfig.hpp` and the directory `internal` with all files and subdirectories.

`install` Install ESPResSo. The variables `prefix` and `exec-prefix` can be used to specify the installation directories, otherwise the defaults defined by the `configure` script are used. `prefix` sets the prefix where all ESPResSo files are to be installed, `exec-prefix` sets the prefix where the executable files are to be installed and is required only when there is an architecture-specific directory.
**Example:** `make install prefix=/usr/local`
will install all files below `/usr/local`.

`ug` Creates the User guide in the `doc/ug` subdirectory (only when using the development sources).

`dg` Creates the Developers' guide in the `doc/dg` subdirectory (only when using the development sources).

`doxygen` Creates the Doxygen code documentation in the `doc/doxygen` subdirectory.

`tutorials` Creates the ESPResSo tutorials in the `doc/tutorials` subdirectory.

`doc` Creates all documentation in the `doc` subdirectory (only when using the development sources).

A number of options are available when calling `make`. The most interesting option is probably `-j` *`num_jobs`*, which can be used for parallel compilation on computers that have more than one CPU or core. *num_jobs* specifies the maximal number of jobs that will be run. Setting *num_jobs* to the number of available processors speeds up the compilation process significantly.

## 3.4. Running ESPResSo

When ESPResSo is found in your path, it can be run via

`Espresso [`*tcl_script* `[`*args*`]]`

When ESPResSo is called without any arguments, it is started in the interactive mode, where new commands can be entered on the command line. When the name of a *tcl_script* is given, the script is executed. Any further arguments are passed to the script.

If you want to run ESPResSo in parallel using MPI, the actual invocation depends on your MPI implementation. In many cases, *e.g.* OpenMPI, the command will be

`mpiexec -n` *n_nodes* `Espresso [`*tcl_script* `[`*args*`]]`

where *n_nodes* denotes the number of MPI nodes to be used. However, note that depending on your MPI installation, MPI jobs can only be run in a queueing system, so that ESPResSo will not run from the command line. Also, older installations sometimes require "-np" instead of "-n" or "mpirun" instead of "mpiexec".

## 3.5. `myconfig.hpp`: Activating and deactivating features

ESPResSo has a large number of features that can be compiled into the binary. However, it is not recommended to actually compile in all possible features, as this will slow down ESPResSo significantly. Instead, compile in only the features that are actually required. A strong gain in speed can be achieved, by disabling all non-bonded interactions except for a single one, e.g. `LENNARD_JONES`. For the developers, it is also possible to turn on or off a number of debugging messages. The features and debug messages can be controlled

via a configuration header file that contains C-preprocessor declarations. Appendix B on page 267 lists and describes all available features. The file `myconfig-sample.hpp` that configure will generate in the build directory contains a list of all possible features that can be copied into your own configuration file. When no configuration header is provided by the user, a default header, found in `src/core/myconfig-default.hpp`, will be used that turns on the default features.

When you distinguish between the build and the source directory, the configuration header can be put in either of these. Note, however, that when a configuration header is found in both directories, the one in the build directory will be used.

By default, the configuration header is called `myconfig.hpp`. The name of the configuration header can be changed either when the `configure`-script is called via the variable `MYCONFIG` (see section 3.2.1 on page 27), or when `make` is called with the setting `myconfig=`*myconfig_header* (see section 3.3 on page 29).

The configuration header can be used to compile different binary versions of ESPResSo with a different set of features from the same source directory. Suppose that you have a source directory `$srcdir` and two build directories `$builddir1` and `$builddir2` that contain different configuration headers:

- `$builddir1/myconfig.hpp`:

  ```
  #define ELECTROSTATICS
  #define LENNARD-JONES
  ```

- `$builddir2/myconfig.hpp`:

  ```
  #define LJCOS
  ```

Then you can simply compile two different versions of ESPResSo via

```
cd $builddir1
$srcdir/configure
make

cd $builddir2
$srcdir/configure
make
```

# 4. Setting up particles

## 4.1. `part`: Creating single particles

### 4.1.1. Defining particle properties

*Python Syntax* (1)

```
espressomd.System().part[⟨pid⟩]
  .type=⟨int⟩
  .pos=⟨array of 3 floats⟩
  .v=⟨array of 3 floats⟩
  .f=⟨array of 3 floats⟩
  .bonds=⟨bond⟩
  .mass=⟨float⟩
  .omega_lab=⟨array of 3 floats⟩
  .rinertia=⟨array of 3 floats⟩
  .omega_body=⟨array of 3 floats⟩
  .torque_lab=⟨array of 3 floats⟩
  .quat=⟨array of 4 floats⟩
  .q=⟨float⟩
  .virtual=⟨int ≥ 0⟩
  .vs_relative=⟨tuple of int and float⟩
  .vs_auto_relate_to=⟨int⟩
  .dip=⟨array of 3 floats⟩
  .dipm=⟨float⟩
  .fix=⟨array of 3 ints⟩
  .ext_force=⟨array of 3 floats⟩
  .ext_torque=⟨array of 3 floats⟩
  .exclude=⟨list of ints⟩
  .temp=⟨float⟩
  .gamma=⟨float⟩
  .gamma_rot=⟨float⟩
  .rotation=⟨bool⟩
  .swimming=⟨dictionary {'f_swim':⟨float⟩, 'v_swim':⟨float⟩,
'mode':⟨"pusher"|"puller"⟩, 'dipole_length':⟨float⟩,
'rotational_friction':⟨float⟩}⟩ delete()
```

*Python Syntax* (2)

```
espressomd.System().part.clear()
```

*Python Syntax* (3)

```
| len(Espressomd.System().part)
```

*TCL Syntax*

`part` *pid* [`pos` *x y z*] [`type` *typeid*] [`v` *vx vy vz*] [`f` *fx fy fz*]
[`bond` *bondid pid2 ...*] [`q` *charge*] [1] [`quat` *q1 q2 q3 q4*] [2]
[`omega_body/lab` *x y z*] [2] [`torque_body/lab` *x y z*] [2]
[`rinertia` *x y z*] [2] [[`un`]`fix` *x y z*] [3] [`ext_force` *x y z*] [3]
[`ext_torque` *x y z*] [2,3] [`exclude` *pid2...*] [4] [`exclude delete` *pid2...*] [4]
[`mass` *mass*] [5] [`dipm` *moment*] [6] [`dip` *dx dy dz*] [6]
[`virtual` *v*] [7,8] [`vs_relative` *pid* `distance`] [8]
[`vs_auto_relate_to` *pid*] [8] [`temp` *T*] [9] [`gamma` *g*] [9]
[`gamma_rot` *grot*] [2,9] [`rotation` *rot*] [10] [`solvation` *lA kA lB kB*] [11]
[`swimming` ( ( `v_swim` *v_swim* | `f_swim` *f_swim* ) | `off` )] [12]
[`swimming` ( ( `v_swim` *v_swim* | `f_swim` *f_swim* ) ( `pusher` | `puller` )
`dipole_length` *dipole_length* `rotational_friction` *rotational_friction* |
`off` )] [12,13]

Required features: [1] `ELECTROSTATICS` [2] `ROTATION` [3] `EXTERNAL_FORCES` [4] `EXCLUSION` [5] `MASS` [6] `DIPOLES` [7] `VIRTUAL_SITES_COM` [8] `VIRTUAL_SITES_RELATIVE` [9] `LANGEVIN_PER_PARTICLE` [10] `ROTATION_PER_PARTICLE` [11] `SHANCHEN` [12] `ENGINE` [13] `LB` or `LB_GPU`

*Description*

This command modifies particle data, namely position, type (monomer, ion, ...), charge, velocity, force and bonds. Multiple properties can be changed at once. If you add a new particle the position has to be set first because of the spatial decomposition.

*Arguments*

- *pid*
- [`pos` *x y z*] Sets the position of this particle to $(x, y, z)$.
- [`type` *typeid*] Restrictions: *typeid* $\geq 0$.
  The *typeid* is used in the `inter` command (see section 5 on page 55) to define the parameters of the non bonded interactions between different kinds of particles.
- [`v` *vx vy vz*] Sets the velocity of this particle to $(vx, vy, vz)$. The velocity remains variable and will be changed during integration.
- [`f` *fx fy fz*] Set the force acting on this particle to $(fx, fy, fz)$. The force remains variable and will be changed during integration. However, whereas the velocity is modified with respect to the velocity you set upon integration, the force it recomputed during the integration step and any force set in this way is lost during the integration step.
- [`bond` *bondid pid2...*] Restrictions: *bondid* $\geq 0$; *pid2* must be an existing particle. The *bondid* is used for the inter command to define bonded interactions.

- `bond delete` Will delete all bonds attached to this particle.
- [`q` *charge*] Sets the charge of this particle to $q$.
- [`quat` *q1 q2 q3 q4*] Sets the quaternion representation of the rotational position of this particle.
- [`omega_body`] $x$ $y$ $z$ ( [ ])`omega_body` $x$ $y$ $z$ The command [omega_body] sets the angular momentum of this particle in the particle's co-rotating frame (or body frame) and the command [omega_lab] sets it for the particle in the fixed frame (or laboratory frame). If you set the angular momentum of the particle in the lab frame, the orientation of the particle ([quat]) must be set before invoking [omega_lab], otherwise the conversion from lab to body frame will not be handled properly.
- [`torque_body/lab` $x$ $y$ $z$] The command [torque_body] sets the torque of this particle in the particle's co-rotating frame (or body frame) and the command [torque_lab] sets it for the particle in the fixed frame (or laboratory frame). If you set the torque of the particle in the lab frame, the orientation of the particle ([quat]) must be set before invoking [torque_lab], otherwise the conversion from lab to body frame will not be handled properly.
- [`rinertia` $x$ $y$ $z$] Sets the diagonal elements of this particles rotational inertia tensor. These correspond with the inertial moments along the coordinate axes in the particle's co-rotating coordinate system. When the particle's quaternions are set to 1 0 0 0, the co-rotating and the fixed (lab) frame are co-aligned.
- [`fix` $x$ $y$ $z$] Fixes the particle in space. By supplying a set of 3 integers as arguments it is possible to fix motion in $x$, $y$, or $z$ coordinates independently. For example *fix* 0 0 1 will fix motion only in z. Note that *fix* without arguments is equivalent to *fix* 1 1 1.
- [`unfix`] Release any external influence from the particle.
- [`ext_force` $x$ $y$ $z$] An additional external force is applied to the particle.
- [`ext_torque` $x$ $y$ $z$] An additional external torque is applied to the particle. This torque is specified in the laboratory frame!
- [`exclude` *pid2*`...+`] Restrictions: *pid2* must be an existing particle. Between the current particle an the exclusion partner(s), no nonbonded interactions are calculated. Note that unlike bonds, exclusions are stored with both partners. Therefore this command adds the defined exclusions to both partners.
- [`exclude delete` *pid2*`...`] Searches for the given exclusion and deletes it. Again deletes the exclusion with both partners.
- [`mass` *mass*] Sets the mass of this particle to *mass*. If not set, all particles have a mass of 1 in reduced units.

- [`dipm` *moment*] Sets the dipol moment of this particle to *moment*.
- [`dip` *dx dy dz*] Sets the orientation of the dipole axis to $(dx, dy, dz)$.
- [`virtual` *v*] Declares the particles as virtual (1) or non-virtual (0, default). Please read chapter 4.4 before using virtual sites.
- [`vs_auto_relate_to` *pid*] Automatically relates a virtual site to a non-virtual particle for the "relative" implementation of virtual sites. *pid* is the id of the particle to which the virtual site should be related.
- [`vs_relative` *pid distance*] Allows for manual access to the attributes of virtual sites in the "relative" implementation. *pid* denotes the id of the particle to which this virtual site is related and *distance* the distance between non-virtual and virtual particle.
- [`temp` *T*] If used in combination with the Langevin thermostat (as documented in section 6.2), sets the temperature *T* individually for the particle with id *pid*. This allows to simulate systems containing particles of different temperatures. Caution: this has no influence on any other thermostat then the Langevin thermostat.
- [`gamma` *g*] If used in combination with the Langevin thermostat (as documented in section 6.2), sets the frictional coefficient *T* individually for the particle with id *pid*. This allows to simulate systems containing particles with different diffusion constants. Caution: this has no influence on any other thermostat then the Langevin thermostat.
- [`rotation` *rot*] Specifies whether a particle's rotational degrees of freedom along the different axes in the particle's body-fixed frame are integrated or not. If set to zero, rotation is disabled entirely, and the content of the torque and omega variables are meaningless. Rotation of a particular axis is turned on by adding up the corresponding flags: x=2, y=4, z=8. To enable rotation around all axis, use a value of 2+4+8=14. This is also the default.
- [`solvation` *lA kA lB kB*] Sets the four solvation coupling constants for the two components of a Shan-Chen fluid, as documented in Section 12.4.
- [`swimming` ( ( `v_swim` *v_swim* | `f_swim` *f_swim* ) | `off` )] Enables the particle to be self-propelled in the direction determined by its quaternion. For setting the quaternion of the particle see `quat`. The self-propulsion speed will relax to a constant velocity, that is specified by `v_swim`. Alternatively it is possible to achieve a constant velocity by imposing a constant force term `f_swim` that is balanced by friction of a (Langevin) thermostat. The way the velocity of the particle decays to the constant terminal velocity in either of these methods is completely determined by the friction coefficient. You may only set one of the possibilities `v_swim` *or* `f_swim` as you cannot relax to constant force *and* constant velocity at the same time. The option `off` (re)sets *v_swim* and *f_swim* both to 0.0 and thus disables swimming. This option applies to all non-lattice-Boltzmann thermostats.

Note that there is no real difference between `v_swim` and `f_swim`, since the latter may aways be chosen such that the same terminal velocity is achieved for a given friction coefficient.

- [`swimming` ( ( `v_swim` *v_swim* | `f_swim` *f_swim* ) ( `pusher` | `puller` ) `dipole_length` *dipole_length* `rotational_friction` *rotational_friction* | `off` )]

  For an explanation of the parameters `v_swim`, `f_swim` and `off` see the previous item. In lattice-Boltzmann self-propulsion is less trivial than for normal MD, because the self-propulsion is achieved by a force-free mechanism, which has strong implications for the far-field hydrodynamic flow field induced by the self-propelled particle. In ESPResSo only the dipolar component of the flow field of an active particle is taken into account. This flow field can be generated by a *pushing* or a *pulling* mechanism, leading to change in the sign of the dipolar flow field with respect to the direction of motion. You can specify the nature of the particle's flow field by using the keywords `pusher` or `puller`. You will also need to specify a *dipole_length* which determines the distance of the source of propulsion from the particle's center. Note that you should not put this distance to zero; ESPResSo (currently) does not support mathematical dipole flow fields. The key `rotational_friction` can be used to set the friction that causes the orientation of the particle to change in shear flow. The torque on the particle is determined by taking the cross product of the difference between the fluid velocity at the center of the particle and at the source point and the vector connecting the center and source.

  You may ask: "Why are there two methods *v_swim* and *f_swim* for the self-propulsion using the lattice-Bolzmann algorithm?" The answer is straightforward. When a particle is accelerating, it has a monopolar flow-field contribution which vanishes when it reaches its terminal velocity (for which there will only be a dipolar flow field). The major difference between the above two methods is that with *v_swim* the flow field *only* has a monopolar moment and *only* while the particle is accelerating. As soon as the particle reaches a constant speed (given by *v_swim*) this monopolar moment is gone and the flow field is zero! In contrast, *f_swim* always, i.e., while accelerating *and* while swimming at constant force possesses a dipolar flow field.

  Please note that even though swimming is interoperable with the CPU version of LB it is only supported on *one* Open MPI node, i.e. `n_nodes` = 1.

**Warning:** The options [omega], [torque], and [tbf] are deprecated and will be removed in some future version.

### 4.1.2. Getting particle properties

*TCL Syntax*

(1) `part` *pid* `print` [( `id` | `pos` | `type` | `folded_position` | `type` | `q` | `v` | `f` | `torque_body` | `torque_lab` | `body_frame_velocity` | `fix` | `ext_force` | `ext_torque` | `bond` | `exclusions connections` [*range*] | `swimming` )]...

(2) `part`

*Description*

Variant (1) will return a list of the specified properties of particle *pid*, or all properties, if no keyword is specified. Variant (2) will return a list of all properties of all particles.

Note that there is a difference between the `*_body` and `*_lab`. The first prints the variable in the co-rotating frame, whereas the second gives the variable in the stationary frame, the body and laboratory frames, respectively. One would typically want to output the variable in the laboratory frame, since it is the frame of interest. However for some tests involving reading and writing the variable it may be desireable to know it in the body frame as well. Be careful with reading and writing, if you write in the lab frame, then read in the lab frame. If you are setting the variable in the lab frame, the orientation of the particle's `quat` must be set before, otherwise the conversion from lab to body frame will not be handled properly. Also be careful about the order in which you write and read in data from a blockfile, for instance if you output the variable in both frames!

The `body_frame_velocity` command is a print-only command that gives the velocity in the body frame, which can be useful for determining the translational diffusion tensor of an anisotropic particle via the velocity auto-correlation (Green-Kubo) method.

*Example*

```
part 40 print id pos q bonds
```

will return a list like

```
40 8.849 1.8172 1.4677 1.0 {}
```

This routine is primarily intended for effective use in Tcl scripts.

When the keyword `connection` is specified, it returns the connectivity of the particle up to *range* (defaults to 1). For particle 5 in a linear chain the result up to *range* = 3 would look like:

```
{ { 4 } { 6 } } { { 4 3 } { 6 7 } } { {4 3 2 } { 6 7 8 } }
```

The function is useful when you want to create bonded interactions to all other particles a certain particle is connected to. Note that this output can not be used as input to the part command. Check results if you use them in ring structures.

If none of the options is specified, it returns all properties of the particle, if it exists, in the form

```
0 pos 2.1 6.4 3.1 type 0 q -1.0 v 0.0 0.0 0.0 f 0.0 0.0 0.0
bonds { {0 480} {0 368} ... }
```

which may be used as an input to this function later on. The first integer is the particle number.

Variant (2) returns the properties of all stored particles in a tcl-list with the same format as specified above:

```
{0 pos 2.1 6.4 3.1 type 0 q -1.0 v 0.0 0.0 0.0 f 0.0 0.0 0.0
 bonds{{0 480}{0 368}...}}
{1 pos 1.0 2.0 3.0 type 0 q 1.0 v 0.0 0.0 0.0 f 0.0 0.0 0.0
 bonds{{0 340}{0 83}...}}
{2...{{...}...}}
{3...{{...}...}}
...
```

When using `pos`, the particle position returned is **unfolded**, for convenience in diffusion calculations etc. Note that therefore blockfiles will contain imaged positions, but un-imaged velocities, which should not be interpreted together. However, that is fine for restoring the simulation, since the particled data is loaded the same way.

### 4.1.3. Deleting particles

What is the syntax of variant 3?

*TCL Syntax*

(1) `part` *pid* `delete`
(2) `part deleteall`

*Description*

In variant (1), the particle *pid* is deleted and all bonds referencing it. Variant (2) will delete all particles currently present in the simulation. Variant (3) will delete all currently defined exclusions.

### 4.1.4. Exclusions

*TCL Syntax*

(1) `part auto_exclusions` [*range*]
(2) `part delete_exclusions`

Required features: `EXCLUSIONS`

*Description*

Variant (1) will create exclusions for all particles pairs connected by not more than *range* bonds (*range* defaults to 2). This is typically used in atomistic simulations, where nearest and next nearest neighbour interactions along the chain have to be omitted since they are included in the bonding potentials. For example, if the system contains particles $0 \ldots 100$, where particle $n$ is bonded to particle $n-1$ for $1 \leq n \leq 100$, then it will result in the exclusions:

- particle 1 does not interact with particles 2 and 3

- particle 2 does not interact with particles 1, 3 and 4
- particle 3 does not interact with particles 1, 2, 4 and 5
- ...

Variant (2) deletes all exclusions currently present in the system.

## 4.2. Creating groups of particle

### 4.2.1. `polymer`: Setting up polymer chains

*TCL Syntax*

`polymer` *num_polymers monomers_per_chain bond_length*
[`start` *pid*] [`pos` *x y z*] [`mode` ( `RW` | `SAW` | `PSAW` ) [*shield* [$try_{\max}$]]]
[`charge` *q*] [1] [`distance` $d_{\text{charged}}$] [1] [`types` $typeid_{\text{neutral}}$ [$typeid_{\text{charged}}$]]
[`bond` *bondid*] [`angle` $\phi$ [$\theta$ [*x y z*]]] [`constraints`] [2]

Required features: [1] `ELECTROSTATICS` [2] `CONSTRAINTS`

*Description*

*Python Syntax* (4)

```
espressomd.polymer(
  N_P = ⟨int⟩,
  MPC = ⟨int⟩,
  bond_length = ⟨float⟩,
  start_id = ⟨int⟩,
  start_pos = ⟨ndarray⟩,
  mode = 0 or 1 or 2,
  shield = ⟨float⟩,
  max_tries = ⟨int⟩,
  val_poly = ⟨float⟩,
  charge_distance = ⟨int⟩,
  type_poly_neutral = ⟨int⟩,
  type_poly_charged = ⟨int⟩,
  bond_id = ⟨int⟩,
  angle = ⟨float⟩,
  angle2 = ⟨float⟩,
  pos2 = ⟨ndarray⟩,
  constraint = 0 or 1)
```

This command will create *num_polymers* N_P polymer or polyelectrolyte chains with *monomers_per_chain* MPC monomers per chain. The length of the bond between two adjacent monomers will be set up to be *bond_length*.

*Arguments*

- *num_polymers* N_P Sets the number of polymer chains.

- *monomers_per_chain* MPC Sets the number of monomers per chain.

- *bond_length* Sets the initial distance between two adjacent monomers. The distance during the course of the simulation depends on the applied potentials. For fixed bond length please refer to the Rattle Shake algorithm[2]. The algorithm is based on Verlet algorithm and satisfy internal constraints for molecular models with internal constrains, using Lagrange multipliers.

- [`start` *pid*] start_id Sets the particle number of the start monomer to be used with the `part` command. This defaults to 0.

- [`pos` *x* *y* *z*] start_pos Sets the position of the first monomer in the chain to *x*, *y*, *z* (defaults to a randomly chosen value)

- [`mode` ( `RW` | `PSAW` | `SAW` ) [*shield* [$try_{\max}$]]] Selects the setup mode:

  `RW` **(Random walk)** mode = 1 The monomers are randomly placed by a random walk with a steps size of $bond_length$.

  `PSAW` **(Pruned self-avoiding walk)** mode = 2 The position of a monomer is randomly chosen in a distance of *bond_length* to the previous monomer. If the position is closer to another particle than *shield*, the attempt is repeated up to $try_{\max}$ times. Note, that this is not a real self-avoiding random walk, as the particle distribution is not the same. If you want a real self-avoiding walk, use the `SAW` mode. However, `PSAW` is several orders of magnitude faster than `SAW`, especially for long chains.

  `SAW` **(Self-avoiding random walk)** mode = 0 The positions of the monomers are chosen as in the plain random walk. However, if this results in a chain that has a monomer that is closer to another particle than *shield*, a new attempt of setting up the whole chain is done, up to $try_{\max}$ times.

  The default for the mode is `RW`, the default for the *shield* is 1.0, and the default for $try_{\max}$ is 30000, which is usually enough for `PSAW`. Depending on the length of the chain, for the `SAW` mode, $try_{\max}$ has to be increased by several orders of magnitude.

- [`charge` *valency*] Sets the valency of the charged monomers. If the valency of the charged polymers *valency* is smaller than $10^{-10}$, the charge is assumed to be zero, and the types are set to $typeid_{\text{charged}} = typeid_{\text{neutral}}$. If charge is not set, it defaults to 0.0.

- [`distance` $d_{\text{charged}}$] charge_distance Sets the stride between the indices of two charged monomers. This defaults defaults to 1, meaning that all monomers in the chain are charged.

- [`types` $typeid_{\text{neutral}}$ `type_poly_neutral type_poly_charged` $typeid_{\text{charged}}$] Sets the type ids of the neutral and charged monomer types to be used with the `part` command. If only $typeid_{\text{neutral}}$ is defined, $typeid_{\text{charged}}$ defaults to 1. If the option is omitted, both monomer types default to 0.

- [`bond` *bondid*] bond_id Sets the type number of the bonded interaction to be set up between the monomers. This defaults to 0. Any bonded interaction, no matter how many bonding-partners needed, is stored with the second particle in this bond. See chapter 5.3.
- [`angle` $\phi$ `[`$\theta$ `[`$x$ $y$ $z$`]]`] angle angle2 pos2 Allows for setting up helices or planar polymers: $\phi$ and *theta* are the angles between adjacent bonds. $x$, $y$ and $z$ set the position of the second monomer of the first chain.
- [`constraints`] If this option is specified, the particle setup-up tries to obey previously defined constraints (see section 4.3 on page 46).

### 4.2.2. `counterions`: Setting up counterions

**Warning:** This feature will not be ported to the python interface.

*TCL Syntax*

`counterions` $N$ [`start` *pid*] [`mode` ( `SAW` | `RW` ) [*shield* [$try_{\max}$ ]]] [`charge` *val*] [1] [`type` *typeid*]

Required features: [1] `ELECTROSTATICS`

*Description*

This command will create $N$ counterions in the simulation box.

*Arguments*

- [`start` *pid*] Sets the particle id of the first counterion. It defaults to the current number of particles, *i.e.* counterions are placed after all previously defined particles.
- [`mode` ( `SAW` | `RW` ) [*shield* [$try_{\max}$ ]]] Specifies the setup method to place the counterions. It defaults to `SAW`. See the `polymer` command for a detailed description.
- [`charge` *val*] Specifies the charge of the counterions. If not set, it defaults to $-1.0$.
- [`type` *typeid*] Specifies the particle type of the counterions. It defaults to 2.

### 4.2.3. `salt`: Setting up salt ions

**Warning:** This feature will not be ported to the python interface.

*TCL Syntax*

`salt` $N_+$ $N_-$ [`start` *pid*] [`mode` ( `SAW` | `RW` ) [*shield* [$try_{\max}$]]] [`charges` $val_+$ [$val_-$]] [1] [`types` $typeid_+$ [$typeid_-$]] [`rad` $r$]

Required features: [1] `ELECTROSTATICS`

*Description*

Create $N_+$ positively and $N_-$ negatively charged salt ions of charge $val_+$ and $val_-$ within the simulation box.

*Arguments*

- [`start` *pid*] Sets the particle id of the first (positively charged) salt ion. It defaults to the current number of particles.
- [`mode` ( `SAW` | `RW` ) [*shield* [$try_{\max}$ ]]] Specifies the setup method to place the counterions. It defaults to `SAW`. See the `polymer` command for a detailed description.
- [`charge` $val_+$ [$val_-$]] Sets the charge of the positive salt ions to $val_+$ and the one of the negatively charged salt ions to $val_-$. If not set, the values default to 1.0 and $-1.0$, respectively.
- [`type` $typeid_+$ [$typeid_-$]] Specifies the particle type of the salt ions. It defaults to 3 respectively 4.
- [`rad` $r$] The salt ions are only placed in a sphere with radius $r$ around the origin.

### 4.2.4. `diamond`: Setting up diamond polymer networks

*TCL Syntax*

`diamond` *a bond_length monomers_per_chain* [`counterions` $N_{\mathrm{CI}}$] [`charges` $val_{\mathrm{node}}$ $val_{\mathrm{monomer}}$ $val_{\mathrm{CI}}$] [1] [`distance` $d_{\mathrm{charged}}$] [1] [`nonet`]

Required features: [1] `ELECTROSTATICS`

*Description*

```
from espressomd import diamond
```

*Python Syntax* (5)

```
diamond.Diamond(
  a=⟨float⟩,
  bond_length=⟨float⟩,
  MPC=⟨int⟩,
  N_CI=⟨int⟩,
  val_nodes=⟨float⟩,
  val_cM=⟨float⟩,
  val_CI=⟨float⟩,
  cM_dist=⟨int⟩,
  nonet)
```

Creates a diamond-shaped polymer network with 8 tetra-functional nodes connected by $2*8$ polymer chains of length *monomers_per_chain* (MPC) in a unit cell of length $a$. Chain monomers are placed at a mutual distance *bond_length* along the vector connecting network nodes. The polymer is created starting from particle ID 0. Nodes are assigned

type 0, monomers (both charged and uncharged) are type 1 and counterions type 2. For inter-particle bonds interaction 0 is taken which must be a two-particle bond.

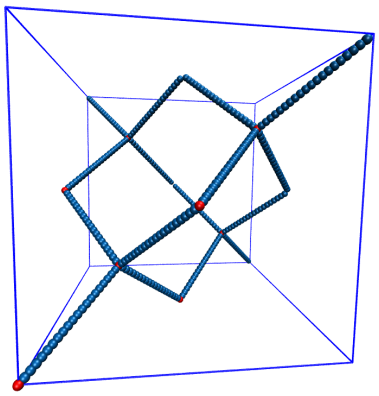


Figure 4.1.: Diamond-like polymer network with *monomers_per_chain*=15.

*Arguments*

- *a* Determines the size of the of the unit cell.
- *bond_length* Specifies the bond length of the polymer chains connecting the 8 tetra-functional nodes.
- *monomers_per_chain* (MPC) Sets the number of chain monomers between the functional nodes.
- [`counterions` $N_{\mathrm{CI}}$] Adds $N_{\mathrm{CI}}$ counterions to the system.
- [`charges` $val_{\mathrm{node}}$ $val_{\mathrm{monomer}}$ $val_{\mathrm{CI}}$] Sets the charge of the nodes to $val_{\mathrm{node}}$, the charge of the connecting monomers to $val_{\mathrm{monomer}}$ (val_cM), and the charge of the counterions to $val_{\mathrm{CI}}$.
- [`distance` $d_{\mathrm{charged}}$] (cM_dist) Specifies the distance between charged monomers along the interconnecting chains. If $d_{\mathrm{charged}} > 1$ the remaining chain monomers are uncharged.
- [`nonet`] Do not create bonds between the chains.

### 4.2.5. `icosaeder`: Setting up an icosaeder

*TCL Syntax*

`icosaeder` *a monomers_per_chain* [`counterions` $N_{\mathrm{CI}}$]
[`charges` $val_{\mathrm{monomers}}$ $val_{\mathrm{CI}}$] [1] [`distance` $d_{\mathrm{charged}}$] [1]

Required features: [1] `ELECTROSTATICS`

*Description*
Creates a modified icosaeder to model a fullerene (or soccer ball). The edges are modeled by polymer chains connected at the corners of the icosaeder. For inter-particle bonds interaction 0 is taken which must be a two-particle bond. Two particle types are used for the pentagons and the interconnecting links. For an example, see figure 4.2.

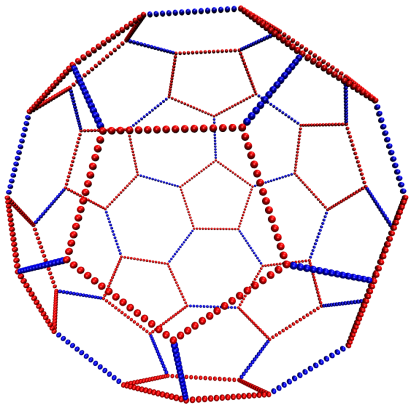


Figure 4.2.: Icosaeder with *monomers_per_chain*=15.

*Arguments*

- $a$ Length of the links. Defines the size of the icosaeder.
- *monomers_per_chain* Specifies the number of chain monomers along one edge.
- [`counterions` $N_{\mathrm{CI}}$] Specifies the number of counterions to be placed into the system.
- [`charges` $val_{\mathrm{monomers}}$ $val_{\mathrm{CI}}$] Set the charges of the monomers to $val_{\mathrm{monomers}}$ and the charges of the counterions to $val_{\mathrm{CI}}$.
- [`distance` $d_{\mathrm{charged}}$] Specifies the distance between two charged monomer along the edge. If $d_{\mathrm{charged}} > 1$ the remaining monomers are uncharged.

### 4.2.6. `crosslink`: Cross-linking polymers

*TCL Syntax*

```
crosslink num_polymer monomers_per_chain [start pid] [catch r_catch]
      [distLink link_dist] [distChain chain_dist] [FENE bondid]
      [trials try_max]
```

*Description*
Attempts to end-crosslink the current configuration of *num_polymer* equally long polymers with *monomers_per_chain* monomers each, returning how many ends are success-

fully connected.

*Arguments*

- [`start` *pid*] *pid* specifies the first monomer of the chains to be linked. It has to be specified if the polymers do not start at id 0.
- [`catch` $r_catch$] Set the radius around each monomer which is searched for possible new monomers to connect to. $r_{\text{catch}}$ defaults to 1.9.
- [`distLink` *link_dist*] The minimal distance of two interconnecting links. It defaults to 2.
- [`distChain` *chain_dist*] The minimal distance for an interconnection along the same chain. It defaults to 0. If set to *monomers_per_chain*, no interchain connections are created.
- [`FENE` *bondid*] Sets the bond type for the connections to *bondid*.
- [`trials` $try_{\max}$] If not specified, $try_{\max}$ defaults to 30000.

### 4.2.7. `copy_particles`: copying a set of particles

*TCL Syntax*

`|copy_particles` [`set` *id1 id2 ...*| `range` *from* `to ...`] [`shift` *s_x s_y s_z*]

*Description*

Copy a group of particles including their bonds. Positions can be [shift]ed by an offset $(s_x, s_y, s_z)$, otherwise the copied set is at exactly the same position as the original set. The particles can be given as a combination of [list]s or [range]s. The new particles obtain in any case consecutive identities after the largest current identity. The mapping of the particles is returned as a list of old-new pairs, which can be conveniently read into an array:

```
array set newidentities [copy_particles ...]
puts "particle 42 is now at position $newidentities(42)"
```

Bonds within the defined particle set are copied with translated identities, but not bonds with particles outside the list. That is, if the particle set corresponds to a molecule, intramolecular bonds are preserved, but not intermolecular ones.

Examples of use:

```
copy_particles set {1 2 3 4} shift 0.0 0.0 0.0
copy_particles set {1 2} set {3 4}
copy_particles range 1 4
```

All these examples do the same—making exact copies of particles 1 through 4.

Please note that `copy_particles` only works if only fundamental particle properties are used, such as `pos`, `type`, etc. as the `copy_particles` procedure only possesses parsers for these. Other properties, such as `quatu` cannot be parsed and will thus lead to an error.

## 4.3. `constraint`: Setting up constraints

*TCL Syntax*

(1) `constraint wall normal` $n_x$ $n_y$ $n_z$ `dist` $d$ `type` *id* [`penetrable` *flag*] [`reflecting` *flag*] [`only_positive` *flag*] [`tunable_slip` *flag*]
(2) `constraint sphere center` $c_x$ $c_y$ $c_z$ `radius` *rad* `direction` *direction* `type` *id* [`penetrable` *flag*] [`reflecting` *flag*]
(3) `constraint cylinder center` $c_x$ $c_y$ $c_z$ `axis` $n_x$ $n_y$ $n_z$ `radius` *rad* `length` *length* `direction` *direction* `type` *id* [`penetrable` *flag*] [`reflecting` *flag*]
(4) `constraint rhomboid corner` $p_x$ $p_y$ $p_z$ `a` $a_x$ $a_y$ $a_z$ `b` $b_x$ $b_y$ $b_z$ `c` $c_x$ $c_y$ $c_z$ `direction` *direction* `type` *id* [`penetrable` *flag*] [`reflecting` *flag*]
(5) `constraint maze nsphere` $n$ `dim` $d$ `sphrad` $r_s$ `cylrad` $r_c$ `type` *id* [`penetrable` *flag*]
(6) `constraint pore center` $c_x$ $c_y$ $c_z$ `axis` $n_x$ $n_y$ $n_z$ `radius` *rad* [`outer_radius` $rad_S$] [`smoothing_radius` $rad_S$] `length` *length* `type` *id*
(7) `constraint stomatocyte center` $x$ $y$ $z$ `orientation` *ox oy oz* `outer_radius` *Ro* `inner_radius` *Ri* `layer_width` $w$ `direction` *direction* `type` *id* [`penetrable` *flag*] [`reflecting` *flag*]
(8) `constraint slitpore pore_mouth` $z$ `channel_width` $c$ `pore_width` $w$ `pore_length` $l$ `upper_smoothing_radius` *us* `lower_smoothing_radius` *ls*
(9) `constraint rod center` $c_x$ $c_y$ `lambda` *lambda* [1]
(10) `constraint plate height` $h$ `sigma` *sigma* [1]
(11) `constraint ext_magn_field` $f_x$ $f_y$ $f_z$ [2,3]
(12) `constraint plane cell` $x$ $y$ $z$ `type` *id*
(13) `constraint mindist_position` $x$ $y$ $z$
(14) `constraint hollow_cone center` $x$ $y$ $z$ `orientation` *ox oy oz* `outer_radius` *Ro* `inner_radius` *Ri* `width` $w$ `opening_angle` *alpha* `direction` *direction* `type` *id* [`penetrable` *flag*] [`reflecting` *flag*]
(15) `constraint spherocylinder center` $c_x$ $c_y$ $c_z$ `axis` $n_x$ $n_y$ $n_z$ `radius` *rad* `length` *length* `direction` *direction* `type` *id* [`penetrable` *flag*] [`reflecting` *flag*]
(16) `constraint mindist_position` $x$ $y$ $z$

Required features: `CONSTRAINTS` [1]`ELECTROSTATICS` [2]`ROTATION` [3]`DIPOLES`

*Description*

The `constraint` command offers a variety of surfaces that can be defined to interact with desired particles. Variants (1) to (7) create interactions via a non-bonded interaction potential, where the distance between the two particles is replaced by the distance of the center of the particle to the surface. The constraints are identified like a particle via its type for the non-bonded interaction. After a type is defined for each constraint one has to define the interaction of all different particle types with the constraint using the `inter` command. In variants (1) to (7), constraints are able to be penetrated if *flag* is set to 1. Otherwise, when the penetrable option is ignored or *flag* is set to 0, the constraint cannot be violated, i.e. no particle can go through the constraint surface. In variants (1) to (4) and (7) it is also possible to specify a flag indicating if the constraints should be reflecting. The flags can equal 1 or 2. The flag 1 corresponds to a reflection process where the normal component of the velocity is reflected and the tangential component remains unchanged. If the flag is 2, also the tangential component is turned around, so that a bounce back motion is performed. The second variant is useful for boundaries of DPD. The reflection property is only activated if an interaction is defined between a particular particle and the constraint! This will usually be a lennard-jones interaction with $\epsilon = 0$, but finite interaction range.

In variant (1) if the `only_positive` flag is set to 1, interactions are only calculated if the particle is on the side of the wall in which the normal vector is pointing. This has only an effect for penetrable walls. If the `tunable_slip` flag is set to 1, then slip boundary interactions apply that are essential for microchannel flows like the Plane Poiseuille or Plane Couette Flow. You also need to use the tunable_slip interaction (see 5.9.1) for this too work.

Variants (9) and (10) create interactions based on electrostatic interactions. The corresponding force acts in direction of the normal vector of the surface and applies to all charged particles.

Variant (11) does not define a surface but is based on magnetic dipolar interaction with an external magnetic field. It applies to all particles with a dipole moment.

The resulting surface in variant (1) is a plane defined by the normal vector $n_x\ n_y\ n_z$ and the distance $d$ from the origin (in the direction of the normal vector). The force acts in direction of the normal. Note that the $d$ describes the distance from the origin in units of the normal vector so that the product of $d$ and $n$ is a point on the surface. Therefore negative distances are quite common!

The resulting surface in variant (2) is a sphere with center $c_x\ c_y\ c_z$ and radius *rad*. The *direction* determines the force direction, -1 or [inside] for inward and +1 or [outside] for outward.

The resulting surface in variant (3) is a cylinder with center $c_x\ c_y\ c_z$ and radius *rad*. The *length* parameter is **half** of the cylinder length. The *axis* is a vector along the cylinder axis, which is normalized in the program. The *direction* is defined the same way as for the spherical constraint.

The resulting surface in variant (4) is a rhomboid, defined by one corner located at $p_x\ p_y\ p_z$ and three adjacent edges, defined by the three vectors connecting the corner p with it's three neighboring corners, a ($a_x\ a_y\ a_z$), b ($b_x\ b_y\ b_z$) and c ($c_x\ c_y\ c_z$).

The resulting surface in variant (5) is $n$ spheres of radius $r_s$ along each dimension, connected by cylinders of radius $r_c$. The spheres have simple cubic symmetry. The spheres are distributed evenly by dividing the $box_l$ by $n$. Dimension of the maze can be controlled by $d$: 0 for one dimensional, 1 for two dimensional and 2 for three dimensional maze.

Variant (6) sets up a cylindrical pore similar to variant (3) with a center $c_x$ $c_y$ $c_z$ and radius *rad*. The *length* parameter is **half** of the cylinder length. The *axis* is a vector along the cylinder axis, which is normalized in the program. Optionally the outer radius of the pore can be specified. By default this is (numerical) infinity and thus results in an infinite wall with one pore. The argument `radius` *rad* can be replaced by the argument `radii` *rad1 rad2* to obtain a pore with a conical shape and corresponding opening radii. The first radius is in the direction opposite to the axis vector. The same applies for `outer_radius` *orad* which can be replaced with `outer_radii` *orad1 orad2*. Per default sharp edges are replaced by circles of unit radius. The radius of this smoothing can be set with the optional keyword `smoothing_radius`.

Variant (7) creates a stomatocyte shaped boundary. This command should be used with care. The position can be any point in the simulation box, and the orientation of the (cylindrically symmetric) stomatocyte is given by a vector, which points in the direction of the symmetry axis, it does not need to be normalized. The parameters: outer_radius *Ro*, inner_radius *Ri*, and layer_width $w$, specify the shape of the stomatocyte. Here inappropriate choices of these parameters can yield undersired results. The width is used as a scaling parameter. That is, a stomatocyte given by $Ro{:}Ri{:}w = 7{:}3{:}1$ is half the size of the stomatocyte given by 7:3:2. Not all choices of the parameters give reasonable values for the shape of the stomatocyte, but the combination 7:3:1 is a good point to start from when trying to modify the shape.

In variant (8), a slit-shaped pore in a T-orientation to a flat channel is created. The geometry is depicted in Fig. 4.3. It translationally invariant in y direction. The pore (lower vertical part) extends in z-direction, and the channel (upper horizontal part). The pore mouth is defined as the z-coordinate, where the lower plane of the channel and the slit pore intersect. It is always centered in the x-direction. A corresponding `dielectric` command decorates the surface with surface charges that can be calculated with the ICC$\star$ algorithm.

Variant (8) specifies an electrostatic interaction between the charged particles in the system to an infinitely long rod with a line charge of *lambda* which is alinge along the z-axis and centered at $c_x$ and $c_y$.

Variant (9) specifies the electrostatic interactinos between the charged particles in the system and an inifinitely large plate in the x-y-plane at height $h$. The plate carries a charge density of *sigma*.

Variant (10) specifies the dipolar coupling of particles with a dipolar moment to an external field $f_x$ $f_y$ $f_z$.

Variant (11) creates an infinite plane at a fixed position. For non-initializing a direction of the constraint values of the positions have to be negative. For the tunable-slip boundary interactions you have to set *two* constraints.

Variant (14) creates a hollow-cone shaped boundary. The position can be any point

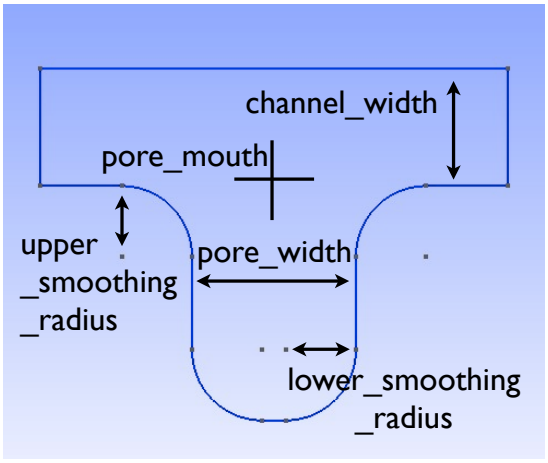



Figure 4.3.: The slitpore created by the `constraint slitpore`.

in the simulation box, and the orientation of the (cylindrically symmetric) cone is given by a vector, which points in the direction of the symmetry axis, it does not need to be normalized. The parameters: outer_radius $Ro$, inner_radius $Ri$, width $w$, and opening_-angle *alpha*, specify the shape of the object. The inner radius gives the narrow end opening size, the outer radius the length of the shaft, and the width the layer width, i.e., the thickness of the cone. The opening_angle (between 0 and $\pi/2$) specifies the angle of the cone.

Variant (15) creates a spherocylinder, that is, a cylinder capped by a hemisphere on either side. The parameter length *length* specifies the length of the shaft, excluding the two hemispherical caps.

Variant (16) calculates the smallest distance to all non-penetrable constraints, that can be repulsive (wall, cylinder, sphere, rhomboid, maze, pore, slitpore). Negative distances mean that the position is "within" the area that particles should not access. Helpful to find initial configurations.)

*Example*

To create an infinite plane in $z$-direction at $z = 20.0$ of type id 1, use:

```
constraint plane cell -10 -10 20 type 1
```

### 4.3.1. Deleting a constraint

*TCL Syntax*

`constraint delete` [*num*]

*Description*

This command will delete constraints. If *num* is specified only this constraint will deleted, otherwise all constraints will be removed from the system.

### 4.3.2. Getting the force on a constraint

*TCL Syntax*

`constraint force` $n$

*Description*
Returns the force acting on the $n$th constraint. Note, however, that this are only forces due to interactions with particles, not with other constraints. Also, these forces still do not mean that the constraints move, they are just the negative of the sum of forces acting on all particles due to this constraint. Similarly, the total energy does not containt constraint-constraint contributions.

### 4.3.3. Getting the currently defined constraints

*TCL Syntax*

`constraint` [$num$]

*Description*
Prints out all constraint information. If $num$ is specified only this constraint is displayed, otherwise all constraints will be printed.

### 4.3.4. `harmonic_well`: Creating a harmonic trap

*TCL Syntax*

`harmonic_well` { $x$ $y$ $z$ } $k$

Required features: `CUDA`

*Description*
Calculates a spring force for all particles, where the equilibrium position of the spring is at $xyz$ and it's force constant is $k$. A more flexible trap can be constructed with constraints, but this one runs on the GPU.

## 4.4. Virtual sites

Virtual sites are particles, the positions and velocities of which are not obtained by integrating an equation of motion. Rather, their coordinates are obtained from the position (and orientation) of one or more other particles. In this way, rigid arrangements of particles can be constructed and a particle can be placed in the center of mass of a set of other particles. Virtual sites can interact with other particles in the system by means of interactions. Forces are added to them according to their respective particle type. Before the next integration step, the forces accumulated on a virtual site are distributed back to those particles, from which the virtual site was derived.

There are two distinct types of virtual sites, described in the following.

### 4.4.1. Virtual sites in the center of mass of a molecule

To activate this implementation, enable the feature `VIRTUAL_SITES_COM` (sec. 3.5). Virtual sites are then placed in the center of mass of a set of particles (as defined below). Their velocity will also be that of the center of mass. Forces accumulating on the virtual sites are distributed back to the particles which form the molecule. To place a virtual site at the center of a molecule, perform the following steps in that order

1. Create a particle of the desired type for each molecule. It should be placed at least roughly in the center of the molecule to make sure, it's on the same node as the other particles forming the molecule, in a simulation with more than one cpu.

2. Make it a virtual site using

   | `part` *pid* `virtual 1`

3. Declare the list of molecules and the particles they consist of:

   | `analyze set` {*moltype listofparticleids* `...`} `...`
   The lists of particles in a molecule comprise the non-virtual particles as well as the virtual site. The id of this molecule is its index in this list. For example,

   ```
   analyze set {0 1 2 3 4} {0 5 6 7 8} {1 9 10 11}
   ```

   declares three molecules, of which the first two consist of three particles and a virtual site each (particles 1–4 and 5–8, respectively). The third molecule has type 1 and consists of two particles and a virtual site. The virtual sites were determined before by setting the *virtual* flag. You can choose freely one out of each molecule, for example particles 1, 5, and 9.

4. Assign to all particles that belong to the same molecule the molecule's id

   | `part` *pid* `mol` *molid*
   The molid is the index of the particle in the above list, so you would assign *molid* 0 to particles 1-4, *molid* 1 to particles 5-8 and *molid* 2 to particles 9-11. Alternatively, you can call

   | `analyze set topo_part_sync`
   to set the *molid*s from the molecule declarations.

5. Update the position of all virtual particles (optional)

   | `integrate 0`

Please note that the use of virtual sites requires that the particles are numbered consecutively. I.e., the particle ids should go from zero to $N - 1$, where $N$ is the number of particles.

The type of the molecule you can choose freely, it is only used in certain analysis functions, namely `energy_kinetic_mol`, `pressure_mol` and `dipmom_mol`, which compute kinetic energy, pressure and dipole moment per molecule type, respectively.

### 4.4.2. Rigid arrangements of particles

The "relative" implementation of virtual sites allows for the simulation of rigid arrangements of particles. It can be used, *e.g.*, for extended dipoles and raspberry-particles, but also for more complex configurations. Position and velocity of a virtual site are obtained from the position and orientation of exactly one non-virtual particle, which has to be placed in the center of mass of the rigid body. Several virtual sites can be related to one and the same non-virtual particle. The position of the virtual site is given by

$$\vec{x_v} = \vec{x_n} + O_n(O_v\vec{E}_z)d, \tag{4.1}$$

where $\vec{x_n}$ is the position of the non-virtual particle, $O_n$ is the orientation of the non-virtual particle, $O_v$ denotes the orientation of the vector $\vec{x_v} - \vec{x_n}$ with respect to the non-virtual particle's body fixed frame and $d$ the distance between virtual and non-virtual particle. In words: The virtual site is placed at a fixed distance from the non-virtual particle. When the non-virtual particle rotates, the virtual sites rotates on an orbit around the non-virtual particle's center.

To use this implementation of virtual sites, activate the feature `VIRTUAL_SITES_RELATIVE` (see sec. 3.5). To set up a virtual site,

1. Place the particle to which the virtual site should be related. It needs to be in the center of mass of the rigid arrangement of particles you create. Let its particle id be n.

2. Place a particle at the desired relative position, make it virtual and relate it to the first particle

   ```
   |part v pos pos virtual 1 vs_auto_relate n
   ```

3. Repeat the previous step with more virtual sites, if desired.

4. To update the positions of all virtual sites, call

   ```
   |integrate 0
   ```

Please note:

- The relative position of the virtual site is defined by its distance from the non-virtual particle, the id of the non-virtual particle and a quaternion which defines the vector from non-virtual particle to virtual site in the non-virtual particle's body-fixed frame. This information is saved in the virtual site's vs_relative-attribute. Take care, not to overwrite these after using vs_auto_relate.

- Virtual sites can not be placed relative to other virtual sites, as the order in which the positions of virtual sites are updated is not guaranteed. Always relate a virtual site to a non-virtual particle placed in the center of mass of the rigid arrangement of particles.

- Don't forget to declare the particle virtual in addition to calling vs_auto_relate

- In case you know the correct quaternions, you can also setup a virtual site using

  `|part` *v* `virtual 1 vs_relative` *n d q*

  where n is the id of the non-virtual particle, d is its distance from the virtual site, and q are the quaternions.

- In a simulation on more than one CPU, the effective cell size needs to be larger than the largest distance between a non-virtual particle and its associated virtual sites. To this aim, you need to set the global variable *min_global_cut* to this largest distance. ESPResSo issues a warning when creating a virtual site with `vs_auto_relate_to` and the cutoff is insufficient.

- If the virtual sites represent actual particles carrying a mass, the inertia tensor of the non-virtual particle in the center of mass needs to be adapted.

- The presence of rigid bodies constructed by means of virtual sites adds a contribution to the pressure and stress tensor.

- The use of virtual sites requires that the particles are numbered consecutively, *i.e.*, the particle ids should go from zero to $N-1$, where $N$ is the number of particles.

### 4.4.3. Additional features

The behaviour of virtual sites can be fine-tuned with the following switches in `myconfig.hpp` (sec. 3.5)

- `VIRTUAL_SITES_NO_VELOCITY` specifies that the velocity of virtual sites is not computed

- `VIRTUAL_SITES_THERMOSTAT` specifies that the Langevin thermostat should also act on virtual sites

- `THERMOSTAT_IGNORE_NON_VIRTUAL` specifies that the thermostat does not act on non-virtual particles

## 4.5. Grand canonical feature

For using ESPResSo conveniently for simulations in the grand canonical ensemble, or other purposes, when particles of certain types are created and deleted frequently. Particle ids can be stored in lists for each individual type and so random ids of particles of a certain type can be drawn.

*TCL Syntax*

`|part gc (` *type* `| ( ( find | delete | status | number )` *type* `) )`

*Description*

```
from espressomd import grand_canonical
grand_canonical.setup([_type])
grand_canonical.delete_particles(_type)
grand_canonical.find_particle(_type)
grand_canonical.number_of_particles(_type)
```

If you want ESPResSo to keep track of particle ids of a certain type you have to initialize the method by calling

*TCL Syntax*

`|part gc` *type*

*Description*

```
grand_canonical.setup([_type])
```

After that ESPResSo will keep track of particle ids of that type. When using the keyword `find` and a particle type, the command will return a randomly chosen particle id, for a particle of the given type. The keyword `status` will return a list with all particles with the given type, similarly giving `number` as argument will return the number of particles which share the given type.

# 5. Setting up interactions

In ESPResSo, interactions are set up and investigated by the `inter` command. There are mainly two types of interactions: non-bonded and bonded interactions. Non-bonded interactions only depend on the *type* of the two involved particles. This also applies to the electrostatic interaction; however, due to its long-ranged nature, it requires special care and ESPResSo handles it separately with a number of state-of-the-art algorithms. The particle type and the charge are both defined using the `part` command.

A bonded interaction defines an interaction between a number of specific particles; it only applies to the set of particles for which it has been explicitly set. A bonded interaction between a set of particles has to be specified explicitly by the `part bond` command, while the `inter` command is used to define the interaction parameters.

*TCL Syntax*

`inter`

*Description*

Without any arguments, `inter` returns a list of all defined interactions as a Tcl-list. The format of each entry corresponds to the syntax for defining the interaction as described below. Typically, this list looks like

```
{0 0 lennard-jones 1.0 2.0 1.1225 0.0 0.0} {0 FENE 7.0 2.0}
```

## 5.1. Isotropic non-bonded interactions

*TCL Syntax*

`inter` *type1 type2* [*interaction*] [*parameters*]

*Description*

This command defines an interaction of type *interaction* between all particles of type *type1* and *type2*. The possible interaction types and their parameters are listed below. If the interaction is omitted, the command returns the currently defined interaction between the two types using the syntax to define the interaction, *e.g.*

```
0 0 lennard-jones 1.0 2.0 1.1225 0.0 0.0
```

For many non-bonded interactions, it is possible to artificially cap the forces, which often allows to equilibrate the system much faster. See the subsection 5.9.5 for details.

### 5.1.1. Tabulated interaction

*TCL Syntax*

`inter` *type1* *type2* `tabulated` *filename*

Required features: `TABULATED`

*Description*

This defines an interaction between particles of the types *type1* and *type2* according to an arbitrary tabulated pair potential. *filename* specifies a file which contains the tabulated forces and energies as a function of the separation distance. The tabulated potential allows capping the force using `inter forcecap`, see section 5.9.5.

At present the required file format is simply an ordered list separated by whitespace. The data reader first looks for a `#` character and begins reading from that point in the file. Anything before the `#` will be ignored.

The first three parameters after the `#` specify the number of data points $N_{\text{points}}$ and the minimal and maximal tabulated separation distances $r_{\text{min}}$ and $r_{\text{max}}$. The number of data points obviously should be an integer, the two other can be arbitrary positive doubles. Take care when choosing the number of points, since a copy of each lookup table is kept on each node and must be referenced very frequently. The maximal tabulated separation distance also acts as the effective cutoff value for the potential.

The remaining data in the file should consist of n data triples $r$, $F(r)$ and $V(r)$. $r$ gives the particle separation, $V(r)$ specifies the interaction potential, and $F(r) = -V'(r)/r$ the force (note the factor $1/r$!). The values of $r$ are assumed to be equally distributed between $r_{\text{min}}$ and $r_{\text{max}}$ with a fixed distance of $(r_{\text{max}} - r_{\text{min}})/(N_{\text{points}} - 1)$; the distance values $r$ in the file are ignored and only included for human readability.

### 5.1.2. Lennard-Jones interaction

*TCL Syntax*

`inter` *type1* *type2* `lennard-jones` $\epsilon$ $\sigma$ $r_{\text{cut}}$ [( $c_{\text{shift}}$|`auto` ) [$r_{\text{off}}$ [$r_{\text{cap}}$ [ $r_{\text{min}}$]]]]

Required features: `LENNARD_JONES`

*Description*

This command defines the traditional (12-6)-Lennard-Jones interaction between particles of the types *type1* and *type2*. The potential is defined by

$$V_{\text{LJ}}(r) = \begin{cases} 4\epsilon((\frac{\sigma}{r-r_{\text{off}}})^{12} - (\frac{\sigma}{r-r_{\text{off}}})^{6} + c_{\text{shift}}) & \text{, if } r_{\text{min}} + r_{\text{off}} < r < r_{\text{cut}} + r_{\text{off}} \\ 0 & \text{, otherwise} \end{cases}. \quad (5.1)$$

The traditional Lennard–Jones potential is the "work–horse" potential of particle–particle interactions in coarse–grained simulations. It is a simple model of the van–der–Waals interaction, and is attractive at large distance, but strongly repulsive at short distances. $r_{\text{off}} + \sigma$ corresponds to the sum of the radii of the interaction particles; at this radius, $V_{\text{LJ}}(r) = 4\epsilon c_{\text{shift}}$. The minimum of the potential is at $r = r_{\text{off}} + 2^{\frac{1}{6}}\sigma$. At this

value of $r$, $V_{\mathrm{LJ}}(r) = -\epsilon + 4\epsilon c_{\mathrm{shift}}$. The attractive part starts beyond this value of $r$. $r_{\mathrm{cut}}$ determines the radius where the potential is cut off.

If $c_{\mathrm{shift}}$ is not set or it is set to the string `auto`, the shift will be automatically computed such that the potential is continuous at the cutoff radius. If $r_{\mathrm{off}}$ is not set, it is set to 0.

The total force on a particle can be capped by using the command `inter forcecap`, see section 5.9.5, or on an individual level using the $r_{\mathrm{cap}}$ variable. When $r_{\mathrm{cap}}$ is set *and* `inter forcecap individual` has been issued before, the maximal force that is generated by this potential is the force at $r_{\mathrm{cap}}$. By default, force capping is off, *i.e.* the cap radius is set to 0.

An optional additional parameter can be used to restrict the interaction from a *minimal* distance $r_{\mathrm{min}}$. This is an optional parameter, set to 0 by default.

A special case of the Lennard–Jones potential is the Weeks–Chandler–Andersen (WCA) potential, which one obtains by putting the cutoff into the minimum, *i.e.* choosing $r_{\mathrm{cut}} = 2^{\frac{1}{6}}\sigma$. The WCA potential is purely repulsive, and is often used to mimick hard sphere repulsion.

When coupling particles to a Shan-Chen fluid, if the `affinity` interaction is set, the Lennard-Jones potential is multiplied by the function

$$A(r) = \begin{cases} \frac{(1-\alpha_1)}{2}[1+\tanh(2\phi)] + \frac{(1-\alpha_2)}{2}[1+\tanh(-2\phi)] & \text{, if } r > r_{\mathrm{cut}} + 2^{\frac{1}{6}}\sigma \\ 1 & \text{, otherwise} \end{cases}, \quad (5.2)$$

where $\alpha_i$ is the affinity to the $i$-th fluid component (see 5.2.3), and the order parameter $\phi$ is calculated from the fluid component local density as $\phi = \frac{\rho_1 - \rho_2}{\rho_1 + \rho_2}$. For example, if the affinities are chosen so that the first component is a good solvent ($\alpha_1 = 1$) and the second one is a bad solvent ($\alpha_2 = 0$), then, if the two particles are both in a region rich in the first component, then $\phi \simeq 1$, and $A(r) \simeq 0$ for $r > r_{\mathrm{cut}} + 2^{\frac{1}{6}}\sigma$. Therefore, the interaction potential will be very close to the WCA one. Conversely, if both particles are in a region rich in the second component, then $\phi \simeq -1$, and $A(r) \simeq 1$, so that the potential will be very close to the full LJ one. If the cutoff has been set large enough, the particle will experience the attractive part of the potential, mimiking the effective attraction induced by the bad solvent.

### 5.1.3. Generic Lennard-Jones interaction

*TCL Syntax*

`inter` *type1 type2* `lj-gen` $\epsilon$ $\sigma$ $r_{\mathrm{cut}}$ $c_{\mathrm{shift}}$ $r_{\mathrm{off}}$ $e_1$ $e_2$ $b_1$ $b_2$ [( $r_{\mathrm{cap}}$|`auto` ) $\lambda$ $\delta$]

Required features: `LENNARD_JONES_GENERIC`

*Description*
This command defines a generalized version of the Lennard-Jones interaction (see section 5.1.2) between particles of the types *type1* and *type2*. The potential is defined by

$$V_{\mathrm{LJ}}(r) = \begin{cases} \epsilon(b_1(\frac{\sigma}{r-r_{\mathrm{off}}})^{e_1} - b_2(\frac{\sigma}{r-r_{\mathrm{off}}})^{e_2} + c_{\mathrm{shift}}) & ,\text{if } r_{\mathrm{min}} + r_{\mathrm{off}} < r < r_{\mathrm{cut}} + r_{\mathrm{off}} \\ 0 & ,\text{otherwise} \end{cases}. \tag{5.3}$$

Note that the prefactor 4 of the standard LJ potential is missing, so the normal LJ potential is recovered for $b_1 = b_2 = 4$, $e_1 = 12$ and $e_2 = 6$.

The total force on a particle can be capped by using the command `inter forcecap`, see section 5.9.5, or on an individual level using the $r_{\mathrm{cap}}$ variable. When $r_{\mathrm{cap}}$ is set *and* `inter forcecap individual` has been issued before, the maximal force that is generated by this potential is the force at $r_{\mathrm{cap}}$. By default, force capping is off, *i.e.* the cap radius is set to 0.

The optional `LJGEN_SOFTCORE` feature activates a softcore version of the potential, where the following transformations apply: $\epsilon \to \lambda\epsilon$ and $r - r_{\mathrm{off}} \to \sqrt{(r - r_{\mathrm{off}})^2 - (1-\lambda)\delta\sigma^2}$. $\lambda$ allows to tune the strength of the interaction, while $\delta$ varies how smoothly the potential goes to zero as $\lambda \to 0$. Such a feature allows one to perform alchemical transformations, where a group of atoms can be slowly turned on/off during a simulation.

### 5.1.4. Lennard-Jones cosine interaction

*TCL Syntax*

(1) `inter` *type1* *type2* `lj-cos` $\epsilon$ $\sigma$ $r_{\mathrm{cut}}$ $r_{\mathrm{off}}$
(2) `inter` *type1* *type2* `lj-cos2` $\epsilon$ $\sigma$ $r_{\mathrm{off}}$ $\omega$
Required features: (1) `LJCOS` (2) `LJCOS2`

*Description*
specifies a Lennard-Jones interaction with cosine tail [57] between particles of the types *type1* and *type2*. The first variant behaves as follows: Until the minimum of the Lennard-Jones potential at $r_{\mathrm{min}} = r_{\mathrm{off}} + 2^{\frac{1}{6}}\sigma$, it behaves identical to the unshifted Lennard-Jones potential ($c_{\mathrm{shift}} = 0$). Between $r_{\mathrm{min}}$ and $r_{\mathrm{cut}}$, a cosine is used to smoothly connect the potential to 0, *i.e.*

$$V(r) = \frac{1}{2}\epsilon\left(cos\left[\alpha(r - r_{\mathrm{off}})^2 + \beta\right] - 1\right), \tag{5.4}$$

where $\alpha = \pi\left[(r_{\mathrm{cut}} - r_{\mathrm{off}})^2 - (r_{\mathrm{min}} - r_{\mathrm{off}})^2\right]^{-1}$ and $\beta = \pi - (r_{\mathrm{min}} - r_{\mathrm{off}})^2\,\alpha$.

In the second variant, the cutoff radius is $r_{\mathrm{cut}} = r_{\mathrm{min}} + \omega$, where $r_{\mathrm{min}} = r_{\mathrm{off}} + 2^{\frac{1}{6}}\sigma$ as in the first variant. The potential between $r_{\mathrm{min}}$ and $r_{\mathrm{cut}}$ is given by

$$V(r) = \epsilon\cos^2\left[\frac{\pi}{2\omega}(r - r_{\mathrm{min}})\right]. \tag{5.5}$$

For $r < r_{\mathrm{min}}$, $V(r)$ is implemented as normal Lennard-Jones potential, see equation 5.1 with $c_{\mathrm{shift}} = 0$.

Only the second variant allows capping the force using `inter forcecap`, see section 5.9.5.

### 5.1.5. Smooth step interaction

*TCL Syntax*

> `inter` *type1 type2* `smooth-step` $\sigma_1$ $n$ $\epsilon$ $k_0$ $\sigma_2$ $r_{\text{cut}}$
>
> Required features: `SMOOTH_STEP`

*Description*

This defines a smooth step interaction between particles of the types *type1* and *type2*, for which the potential is

$$V(r) = (\sigma_1/d)^n + \epsilon/(1 + \exp[2k_0(r - \sigma_2)]) \tag{5.6}$$

for $r < r_{cut}$, and $V(r) = 0$ elsewhere. With $n$ around 10, the first term creates a short range repulsion similar to the Lennard-Jones potential, while the second term provides a much softer repulsion. This potential therefore introduces two length scales, the range of the first term, $\sigma_1$, and the range of the second one, $\sigma_2$, where in general $\sigma_1 < \sigma_2$.

### 5.1.6. BMHTF potential

*TCL Syntax*

> `inter` *type1 type2* `bmhtf-nacl` $A$ $B$ $C$ $D$ $\sigma$ $r_{\text{cut}}$
>
> Required features: `BMHTF_NACL`

*Description*

This defines an interaction with the *short-ranged part* of the Born-Meyer-Huggins-Tosi-Fumi potential between particles of the types *type1* and *type2*, which is often used to simulate NaCl crystals. The potential is defined by:

$$V(r) = A\exp[B(\sigma - r)] - Cr^{-6} - Dr^{-8} + \epsilon_{\text{shift}}, \tag{5.7}$$

where $\epsilon_{\text{shift}}$ is chosen such that $V(r_{\text{cut}}) = 0$. For $r \geq r_{\text{cut}}$, the $V(r) = 0$.

For NaCl, the parameters should be chosen as follows:

| types | $A$ (kJ/mol) | $B$ (Å$^{-1}$) | $C$ (Å$^6$kJ/mol) | $D$ Å$^8$kJ/mol | $\sigma$ (Å) |
|---|---|---|---|---|---|
| Na-Na | 25.4435 | 3.1546 | 101.1719 | 48.1771 | 2.34 |
| Na-Cl | 20.3548 | 3.1546 | 674.4793 | 837.0770 | 2.755 |
| Cl-Cl | 15.2661 | 3.1546 | 6985.6786 | 14031.5785 | 3.170 |

The cutoff can be chosen relatively freely because the potential decays fast; a value around 10 seems reasonable.

In addition to this short ranged interaction, one needs to add a Coulombic, long–ranged part. If one uses elementary charges, *i.e.* a charge of $q = +1$ for the Na–particles, and $q = -1$ for the Cl–particles, the corresponding prefactor of the Coulomb interaction is $\approx 1389.3549 \mathring{A}\, kJ/mol$.

### 5.1.7. Morse interaction

*TCL Syntax*

> `inter` *type1* *type2* `morse` $\epsilon$ $\alpha$ $r_{\min}$ $r_{\text{cut}}$
>
> Required features: `MORSE`

*Description*

This defines an interaction using the Morse potential between particles of the types *type1* and *type2*. It serves similar purposes as the Lennard-Jones potential, but has a deeper minimum, around which it is harmonic. This models the potential energy in a diatomic molecule. This potential allows capping the force using `inter forcecap`, see section 5.9.5.

For $r < r_{\text{cut}}$, this potential is given by

$$V(r) = \epsilon \left(\exp\left[-2\alpha\left(r - r_{\min}\right)\right] - 2\exp\left[-\alpha\left(r - r_{\min}\right)\right]\right) - \epsilon_{\text{shift}}, \tag{5.8}$$

where $\epsilon_{\text{shift}}$ is again chosen such that $V(r_{\text{cut}}) = 0$. For $r \geq r_{\text{cut}}$, the $V(r) = 0$.

### 5.1.8. Buckingham interaction

*TCL Syntax*

> `inter` *type1* *type2* `buckingham` $A$ $B$ $C$ $D$ $r_{\text{cut}}$ $r_{\text{discont}}$ $\epsilon_{\text{shift}}$
>
> Required features: `BUCKINGHAM`

*Description*

This defines a Buckingham interaction between particles of the types *type1* and *type2*, for which the potential is given by

$$V(r) = A\exp(-Br) - Cr^{-6} - Dr^{-4} + \epsilon_{\text{shift}} \tag{5.9}$$

for $r_{\text{discont}} < r < r_{\text{cut}}$. Below $r_{\text{discont}}$, the potential is linearly continued towards $r = 0$, similarly to force capping, see below. Above $r = r_{\text{cut}}$, the potential is 0. This potential allows capping the force using `inter forcecap`, see section 5.9.5.

### 5.1.9. Soft-sphere interaction

*TCL Syntax*

> `inter` *type1* *type2* `soft-sphere` $a$ $n$ $r_{\text{cut}}$ $r_{\text{offset}}$
>
> Required features: `SOFT_SPHERE`

*Description*

This defines a soft sphere interaction between particles of the types *type1* and *type2*, which is defined by a single power law:

$$V(r) = a\left(r - r_{\mathit{offset}}\right)^{-n} \tag{5.10}$$

for $r < r_{\text{cut}}$, and $V(r) = 0$ above. There is no shift implemented currently, which means that the potential is discontinuous at $r = r_{\text{cut}}$. Therefore energy calculations should be used with great caution.

### 5.1.10. Membrane-collision interaction

*TCL Syntax*

> `inter` *type1* *type2* `membrane` $a$ $n$ $d_{\text{cut}}$ $d_{\text{offset}}$
>
> Required features: `MEMBRANE_COLLISION`

*Description*

This defines a membrane collision interaction between particles of the types *type1* and *type2*, where particle of *type1* belongs to one OIF or OIF-like object and particle of *type2* belongs to another such object.

It is very similar to soft-sphere interaction, but it takes into account the local outward normal vectors on the surfaces of the two objects to determine the direction for repulsion of objects (i.e. determine whether the two membranes are intersected). It is inversely proportional to the distance of nodes of membranes that are not crossed and saturating with growing distance of nodes of crossed membranes.

In order to work with the OIF objects, both of them need to be created using templates with keyword *normal*, because this implicitly sets up the bonded out-direction interaction, which computes the outward normal vector.

The membrane-collision interaction for non-intersected membranes is then defined by:

$$V(d) = a\frac{1}{1+e^{n\left(d-d_{offset}\right)}}, \tag{5.11}$$

for $d < d_{\text{cut}}$ and $V(d) = 0$ above. For intersected membranes, it is defined as $V(-d)$. There is no shift implemented currently, which means that the potential is discontinuous at $d = d_{\text{cut}}$. Therefore energy calculations should be used with great caution.

### 5.1.11. Hat interaction

*TCL Syntax*

> `inter` *type1* *type2* `hat` $F_{\text{max}}$ $r_c$
>
> Required features: `HAT`

*Description*

This defines a simple force ramp between particles of the types *type1* and *type2*. The maximal force $F_{\max}$ acts at zero distance and zero force is applied at distances $r_c$ and bigger. For distances smaller than $r_c$, the force is given by

$$F(r) = F_{\max} \cdot \left(1 - \frac{r}{r_c}\right), \tag{5.12}$$

for distances exceeding $r_c$, the force is zero.

The potential energy is given by

$$V(r) = F_{\max} \cdot (r - r_c) \cdot \left(\frac{r + r_c}{2r_c} - 1\right), \tag{5.13}$$

which is zero for distances bigger than $r_c$ and continuous at distance $r_c$.

This is the standard conservative DPD potential and can be used in combination with `inter DPD` 5.9.2. The potential is also useful for live demonstrations, where a big time step may be employed to obtain quick results on a weak machine, for which the physics do not need to be entirely correct.

### 5.1.12. Hertzian interaction

*TCL Syntax*

`inter` *type1* *type2* `hertzian` $\sigma$ $\epsilon$

Required features: `HERTZIAN`

*Description*

This defines an interaction according to the Hertzian potential between particles of the types *type1* and *type2*. The Hertzian potential is defined by

$$V(r) = \begin{cases} \epsilon \left(1 - \frac{r}{\sigma}\right)^{5/2} & r < \sigma \\ 0 & r \geq \sigma. \end{cases} \tag{5.14}$$

The potential has no singularity and is defined everywhere; the potential has nondifferentiable maximum at $r = 0$, where the force is undefined.

### 5.1.13. Gaussian

*TCL Syntax*

`inter` *type1* *type2* `gaussian` $\sigma$ $\epsilon$ $r_{\text{cut}}$

Required features: `GAUSSIAN`

*Description*

This defines an interaction according to the Gaussian potential between particles of the typers *type1* and *type2*. The Gaussian potential is defined by

$$V(r) = \begin{cases} \epsilon\, e^{-\frac{1}{2}\left(\frac{r}{\sigma}\right)^2} & r < r_{\text{cut}} \\ 0 & r \geq r_{\text{cut}} \end{cases} \tag{5.15}$$

The Gaussian potential is smooth except at the cutoff, and has a finite overlap energy of $\epsilon$. It can be used to model *e.g.* overlapping polymer coils.

Currently, there is no shift implemented, which means that the potential is discontinuous at $r = r_{\text{cut}}$. Therefore use caution when performing energy calculations. However, you can often choose the cutoff such that the energy difference at the cutoff is less than a desired accuracy, since the potential decays very rapidly.

## 5.2. Anisotropic non-bonded interactions

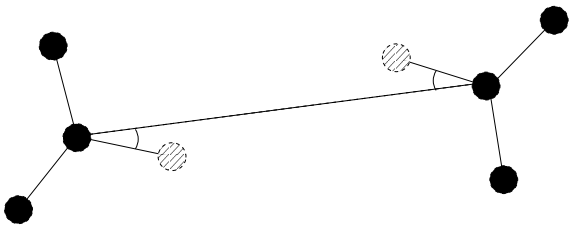
### 5.2.1. Directional Lennard-Jones interaction

*TCL Syntax*

`inter` *type1 type2* `lj-angle` $\epsilon$ $\sigma$ $r_{\text{cut}}$ $b1_a$ $b1_b$ $b2_a$ $b2_b$ $[r_{\text{cap}}$ $z_0$ $\delta z$ $\kappa$ $\epsilon']$

Required features: `LJ_ANGLE`

*Description*



Specifies a 12-10 Lennard-Jones interaction with angular dependence between particles of the types *type1* and *type2*. These two particles need two bonded partners oriented in a symmetric way. They define an orientation for the central particle. The purpose of using bonded partners is to avoid dealing with torques, therefore the interaction does *not* need the ROTATION feature. The angular part of the potential minimizes the system when the two central beads are oriented along the vector formed by these two particles. The shaded beads on the image are virtual particles that are formed from the orientation of the bonded partners, connected to the central beads. They are used to define angles. The potential is of the form

$$U(r_{ik}, \theta_{jik}, \theta_{ikn}) = \epsilon \left[ 5 \left( \frac{\sigma}{r} \right)^{12} - 6 \left( \frac{\sigma}{r} \right)^{10} \right] \cos^2 \theta_{jik} \cos^2 \theta_{ikn}, \tag{5.16}$$

where $r_{ik}$ is the distance between the two central beads, and each angle defines the orientation between the direction of a central bead (determined from the two bonded partners) and the vector $\mathbf{r_{ik}}$. Note that the potential is turned off if one of the angle is more than $\pi/2$. This way we don't end up creating a minimum for an anti-parallel configuration.

Unfortunately, the bonded partners are not sought dynamically. One has to keep track of the relative positions of the particle IDs. This can be done by setting the parameters $b1_a$, $b1_b$, $b2_a$, and $b2_b$. Say the first bead *type1* has particle ID $n$, then one should set the simulation such as its two bonded partners have particle IDs $n + b1_a$ and $n + b1_b$, respectively. On a linear chain, for example, one would typically have $b1_a = 1$ and $b1_b = -1$ such that the central bead and its two bonded partners have position IDs $n$, $n + 1$, and $n - 1$, respectively. This is surely not optimized, but once the simulation is set correctly the algorithm is very fast.

The force can be capped using `inter forcecap`. It might turn out to be useful in some cases to keep this capping during the whole simulation. This is due to the very sharp angular dependence for small distance, compared to $\sigma$. Two beads might come

very close to each other while having unfavorable angles such that the interaction is turned off. Then a change in the angle might suddenly turn on the interaction and the system will blow up (the potential is so steep that one would need extremely small time steps to deal with it, which is not very clever for such rare events).

For instance, when modeling hydrogen bonds (N-H...O=C), one can avoid simulating hydrogens and oxygens by using this potential. This comes down to implementing a HBond potential between N and C atoms.

The optional parameter $r_{\text{cap}}$ is the usual cap radius. The four other optional parameters ($z_0$, $\delta z$, $\kappa$, $\epsilon'$) describe a different interaction strength $\epsilon'$ for a subset of the simulation box. The box is divided through the $z$ plane in two different regions: region 1 which creates an interaction with strength $\epsilon$, region 2 with interaction strength $\epsilon'$. The 2nd region is defined by its $z$-midplane $z_0$, its total thickness $\delta z$, and the interface width $\kappa$. Therefore, the interaction strength is $\epsilon$ everywhere except for the region of the box $z_0 - \delta z/2 < z < z_0 + \delta z/2$. The interface width smoothly interpolates between the two regions to avoid discontinuities. As an example, one can think of modeling hydrogen bonds in two different environments: water, where the interaction is rather weak, and in a lipid bilayer, where it is comparatively stronger.

### 5.2.2. Gay-Berne interaction

*TCL Syntax*

`inter` *type1* *type2* `gay-berne` $\epsilon_0$ $\sigma_0$ $r_{\text{cutoff}}$ *k1* *k2* $\mu$ $\nu$

Required features: `ROTATION` `GAY_BERNE`

*Description*

This defines a Gay-Berne potential for prolate and oblate particles between particles of the types *type1* and *type2*. The Gay-Berne potential is an anisotropic version of the classic Lennard-Jones potential, with orientational dependence of the range $\sigma_0$ and the well-depth $\epsilon_0$.

Assume two particles with orientations given by the unit vectors $\hat{\mathbf{u}}_i$ and $\hat{\mathbf{u}}_j$ and intermolecular vector $\mathbf{r} = r\hat{\mathbf{r}}$. If $r < r_{cut}$, then the interaction between these two particles is given by

$$V(\mathbf{r}_{ij}, \hat{\mathbf{u}}_i, \hat{\mathbf{u}}_j) = 4\epsilon(\hat{\mathbf{r}}_{ij}, \hat{\mathbf{u}}_i, \hat{\mathbf{u}}_j)\left(\tilde{r}_{ij}^{-12} - \tilde{r}_{ij}^{-6}\right), \tag{5.17}$$

otherwise $V(r) = 0$. The reduced radius is

$$\tilde{r} = \frac{r - \sigma(\hat{\mathbf{r}}, \hat{\mathbf{u}}_i, \hat{\mathbf{u}}_j) + \sigma_0}{\sigma_0}, \tag{5.18}$$

$$\sigma(\hat{\mathbf{r}}, \hat{\mathbf{u}}_i, \hat{\mathbf{u}}_j) = \sigma_0 \left\{1 - \frac{1}{2}\chi\left[\frac{(\hat{\mathbf{r}}\cdot\hat{\mathbf{u}}_i + \hat{\mathbf{r}}\cdot\hat{\mathbf{u}}_j)^2}{1+\chi\hat{\mathbf{u}}_i\cdot\hat{\mathbf{u}}_j} + \frac{(\hat{\mathbf{r}}\cdot\hat{\mathbf{u}}_i - \hat{\mathbf{r}}\cdot\hat{\mathbf{u}}_j)^2}{1-\chi\hat{\mathbf{u}}_i\cdot\hat{\mathbf{u}}_j}\right]\right\}^{-\frac{1}{2}} \tag{5.19}$$

and

$$\epsilon(\hat{\mathbf{r}}, \hat{\mathbf{u}}_i, \hat{\mathbf{u}}_j) = \epsilon_0 \left(1 - \chi^2 (\hat{\mathbf{u}}_i \cdot \hat{\mathbf{u}}_j)\right)^{-\frac{\nu}{2}} \left[1 - \frac{\chi'}{2} \left( \frac{(\hat{\mathbf{r}} \cdot \hat{\mathbf{u}}_i + \hat{\mathbf{r}} \cdot \hat{\mathbf{u}}_j)^2}{1 + \chi' \hat{\mathbf{u}}_i \cdot \hat{\mathbf{u}}_j} + \frac{(\hat{\mathbf{r}} \cdot \hat{\mathbf{u}}_i - \hat{\mathbf{r}} \cdot \hat{\mathbf{u}}_j)^2}{1 - \chi' \hat{\mathbf{u}}_i \cdot \hat{\mathbf{u}}_j} \right) \right]^{\mu}. \quad (5.20)$$

The parameters $\chi = \left(k_1^2 - 1\right) / \left(k_1^2 + 1\right)$ and $\chi' = \left(k_2^{1/\mu} - 1\right) / \left(k_2^{1/\mu} + 1\right)$ are responsible for the degree of anisotropy of the molecular properties. $k_1$ is the molecular elongation, and $k_2$ is the ratio of the potential well depths for the side-by-side and end-to-end configurations. The exponents $\mu$ and $\nu$ are adjustable parameters of the potential. Several Gay-Berne parametrizations exist, the original one being $k_1 = 3$, $k_2 = 5$, $\mu = 2$ and $\nu = 1$.

### 5.2.3. Affinity interaction

*TCL Syntax*

`inter` *type1* *type2* `affinity` $\alpha_1$ $\alpha_2$

Required features: `SHANCHEN`

*Description*

Instead of defining a new interaction, this command acts as a modifier for existing interactions, so that the conditions of good/bad solvent associated to the two components of a Shan-Chen fluid. The two types must match those of the interaction that one wants to modify, and the two affinity values $\alpha_1$ and $\alpha_2$ are values between 0 and 1. A value of 1 (of 0) indicates that the component acts as a good (bad) solvent. The specific functional form depends on the interaction type and is listed in the interaction section. So far, only the standard Lennard-Jones interaction is modified by the `affinity` interaction.

## 5.3. Bonded interactions

*TCL Syntax*

`inter` *bondid* [*interaction*] [*parameters*]

*Description*

Bonded interactions are identified by their *bonded interaction type identificator* *bondid*, which is a non-negative integer. The `inter` *bondid* command is used to specify the type and parameters of a bonded interaction, which applies to all particles connected explicitly by this bond using the `part` command (see section 4.1 on page 32). Therefore, defining a bond between two particles always involves two steps: defining the interaction and applying it. Assuming that two particles with ids 42 and 43 already exist, one can create *e.g.* a FENE-bond between them using

```
inter 1 fene 10.0 2.0
part 42 bond 1 43
```

If a FENE-bond with the same interaction parameters is required between several particles (*e.g.* in a simple chain molecule), one can use the same type *id*:

```
inter 1 fene 10.0 2.0
part 42 bond 1 43; part 43 bond 1 44
```

Bonds can have more than just two bond partners. For the `inter` command that does not play a role as it only specifies the parameters, only when applying the bond using the `bond` particle, the number of involved particles plays a role. The number of involved particles and their order, if important, is nevertheless specified here for completeness.

In the python interface you don't have to worry about bond ids. A minimalistic example how to set up bonded harmonic interactions between two particles (particle ids 0 and 1) follows:

*Python Syntax* (6)

```
espressomd.interactions.HarmonicBond(
  r_0 = ⟨float⟩,
  k = ⟨float⟩,
  r_cut = ⟨float⟩)
```

### 5.3.1. FENE bond

*Python Syntax* (7)

```
espressomd.interaction.FeneBond(
  k = ⟨float⟩,
  d_r_max = ⟨float⟩)
```

*TCL Syntax*

`inter` *bondid* `fene` $K$ $\Delta r_{\max}$ $[r_0]$

*Description*

*Python Syntax* (8)

```
espressomd.System().bondedInter.add(FeneBond).setParams(
  k = ⟨float⟩,
  drmax = ⟨float⟩,
  r_0 = ⟨float=0.⟩)
```

This creates a bond type with identificator *bondid* with a FENE (finite extension nonlinear expander) interaction. This is a rubber-band-like, symmetric interaction between two particles with prefactor $K$, maximal stretching $\Delta r_{\max}$ and equilibrium bond length $r_0$. The bond potential diverges at a particle distance $r = r_0 - \Delta r_{\max}$ and $r = r_0 + \Delta r_{\max}$. It is given by

$$V(r) = -\frac{1}{2} K \Delta r_{\max}{}^2 \ln \left[ 1 - \left( \frac{r - r_0}{\Delta r_{\max}} \right)^2 \right]. \tag{5.21}$$

### 5.3.2. Harmonic bond

*Python Syntax* (9)

```
espressomd.interactions.HarmonicBond(
  k = ⟨float⟩,
  r_0 = ⟨float⟩,
  r_cut = ⟨float⟩)
```

*TCL Syntax*

`inter` *bondid* `harmonic` $K$ $R$ $[r_{\mathrm{cut}}]$

*Description*

This creates a bond type with identificator *bondid* with a classical harmonic potential. It is a symmetric interaction between two particles. The potential is minimal at particle distance $r = R$, and the prefactor is $K$. It is given by

$$V(r) = \frac{1}{2} K \left(r - R\right)^2 \tag{5.22}$$

The third, optional parameter $r_{\mathrm{cut}}$ defines a cutoff radius. Whenever a harmonic bond gets longer than $r_{\mathrm{cut}}$, the bond will be reported as broken, and a background error will be raised.

### 5.3.3. Harmonic Dumbbell Bond

*Python Syntax* (10)

```
espressomd.interactions.HarmonicDumbbellBond(
  k1 = ⟨float⟩,
  k2 = ⟨float⟩,
  r_0 = ⟨float⟩,
  r_cut = ⟨float⟩)
```

*TCL Syntax*

`inter` *bondid* `harmonic_dumbbell` $k_1$ $k_2$ $r$ $[r_{\mathrm{cut}}]$

Required features: `ROTATION`

*Description*

This bond is similar to the normal harmonic bond in such a way that it sets up a harmonic potential, i.e. a spring, between the two particles. Additionally the `quat` of the first particle in the bond will be aligned along the distance vector between both particles. This alignment can be controlled by the second harmonic constant $k_2$. Keep in mind that the `quat` will perform an oscillation around the distance vector and some kind of friction needs to be present for the `quat` to relax.

The role of the parameters $k_1$, $r$, and $r_{\mathrm{cut}}$ is exactly the same as for the harmonic bond.

### 5.3.4. Quartic bond

*TCL Syntax*

| `inter` *bondid* `quartic` $K_0$ $K_1$ $R$ $[r_{\text{cut}}]$

*Description*
This creates a bond type with identificator *bondid* with a quartic potential. The potential is minimal at particle distance $r = R$. It is given by

$$V(r) = \frac{1}{2}K_0\,(r-R)^2 + \frac{1}{4}K_1\,(r-R)^4 \tag{5.23}$$

The fourth, optional, parameter $r_{\text{cut}}$ defines a cutoff radius. Whenever a quartic bond gets longer than $r_{\text{cut}}$, the bond will be reported as broken, and a background error will be raised.

### 5.3.5. Bonded coulomb

*TCL Syntax*

| `inter` *bondid* `bonded_coulomb` $\alpha$

*Description*
This creates a bond type with identificator *bondid* with a coulomb pair potential. It is given by

$$V(r) = \frac{\alpha q_1 q_2}{r}, \tag{5.24}$$

where $q_1$ and $q_2$ are the charges of the bound particles. There is no cutoff, the bejerrum length of other coulomb interactions is not taken into account.

### 5.3.6. Subtracted Lennard-Jones bond

*TCL Syntax*

| `inter` *bondid* `subt_lj` *reserved* $R$

*Description*
This creates a "bond" type with identificator *bondid*, which acts between two particles and actually subtracts the Lennard-Jones interaction between the involved particles. The first parameter, *reserved* is a dummy just kept for compatibility reasons. The second parameter, $R$, is used as a check: if any bond length in the system exceeds this value, the program terminates. When using this interaction, it is worthwhile to consider capping the Lennard-Jones potential appropriately so that round-off errors can be avoided.

This interaction is useful when using other bond potentials which already include the short–ranged repulsion. This often the case for force fields or in general tabulated potentials.

### 5.3.7. Rigid bonds

*Python Syntax* (11)

```
espressomd.interactions.RigidBond(
  r = ⟨float⟩,
  ptol = ⟨float⟩,
  vtol = ⟨float⟩)
```

*TCL Syntax*

`inter` *bondid* `rigid_bond` *constrained_bond_distance* *positional_tolerance* *velocity_tolerance*

Required features: `BOND_CONSTRAINT`

*Description*

To simulate rigid bonds, ESPResSo uses the Rattle Shake algorithm which satisfies internal constraints for molecular models with internal constraints, using Lagrange multipliers.[2] In the python implementation the constrained bond distance is named r, the positional tolerance is named ptol and the velocity tolerance is named vtol.

### 5.3.8. Tabulated bond interactions

*TCL Syntax*

(1) `inter` *bondid* `tabulated bond` *filename*
(2) `inter` *bondid* `tabulated angle` *filename*
(3) `inter` *bondid* `tabulated dihedral` *filename*

*Description*

This creates a bond type with identificator *bondid* with a two-body bond length (variant (1)), three-body angle (variant (2)) or four-body dihedral (variant (3)) tabulated potential. The tabulated forces and energies have to be provided in a file *filename*, which is formatted identically as the files for non-bonded tabulated potentials (see section 5.1.1).

The potential is calculated as follows:

- Variant (1) is a two body interaction depending on the distance of two particles. The force acts in the direction of the connecting vector between the particles. The bond breaks above the tabulated range, but for distances smaller than the tabulated range, a linear extrapolation based on the first two tabulated force values is used.

- Variant (2) is a three-body angle interaction similar to the `angle` potential (see section 5.5). It is assumed that the potential is tabulated for all angles between 0 and $\pi$, where 0 corresponds to a stretched polymer, and just as for the tabulated pair potential, the forces are scaled with the inverse length of the connecting vectors. The force on particles $p_1$ and $p_3$ (in the notation of section 5.5) acts perpendicular to the connecting vector between the particle and the center particle

$p_2$ in the plane defined by the three particles. The force on the center particle $p_2$ balances the other two forces.

- Variant (3) tabulates a torsional dihedral angle potential (see section 5.6). It is assumed that the potential is tabulated for all angles between 0 and $2\pi$. *This potential is not tested yet! Use on own risk, and please report your findings and eventually necessary fixes.*

### 5.3.9. Virtual bonds

*Python Syntax* (12)

```
espressomd.interactions.Virtual()
```

*TCL Syntax*

```
inter bondid virtual_bond
```

*Description*
This creates a virtual bond type with identificator *bondid*, *i.e.* a pair bond without associated potential or force. It can used to specify topologies and for some analysis that rely on bonds, or *e.g.* for bonds that should be displayed in VMD.

## 5.4. Object-in-fluid interactions

Please cite [14] (BibTeX-key `cimrak` in file `doc/ug/citations.bib`) when using the interactions in this section in order to simulate extended objects embedded in a LB fluid. For more details also see the documentation at cell-in-fluid.fri.uniza.sk/oif-documentation

The following interactions are implemented in order to mimic the mechanics of elastic or rigid objects immersed in the LB fluid flow. Their mathematical formulations were inspired by [23]. Details on how the bonds can be used for modeling objects are described in section 14.

### 5.4.1. OIF local forces

*TCL Syntax*

`inter` *bondid* `oif_local_force` $L^0_{AB}$ $k_s$ $k_{slin}$ $\phi$ $k_b$ $A_1$ $A_2$ $k_{al}$

Required features: `OIF_LOCAL_FORCES`

*Description*
This type of interaction is available for closed 3D immersed objects as well as for 2D sheet flowing in the 3D flow.

This interaction comprises three different concepts. The local elasticity of biological membranes can be captured by three different elastic moduli. Stretching of the membrane, bending of the membrane and local preservation of the surface area. Parameters $L^0_{AB}$, $k_s$, $k_{slin}$ define the stretching, parameters $\phi$, $k_b$ define the bending, and $A_1$, $A_2$, $k_{al}$ define the preservation of local area. They can be used all together, or, by setting any of $k_s, k_{slin}, k_b, k_{al}$ to zero, the corresponding modulus can be turned off.

### Stretching

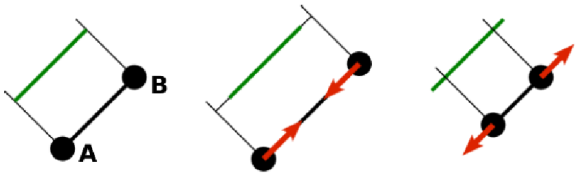
For each edge of the mesh, $L_{AB}$ is the current distance between point A and point B. $L^0_{AB}$ is the distance between these points in the relaxed state, that is if the current edge has the length exactly $L^0_{AB}$, then no forces are added. $\Delta L_{AB}$ is the deviation from the relaxed state, that is $\Delta L_{AB} = L_{AB} - L^0_{AB}$. The stretching force between A and B is calculated using

$$F_s(A, B) = (k_s \kappa(\lambda_{AB}) + k_{slin}) \Delta L_{AB} n_{AB}. \tag{5.25}$$

Here, $n_{AB}$ is the unit vector pointing from $A$ to $B$, $k_s$ is the constant for nonlinear stretching, $k_{slin}$ is the constant for linear stretching, $\lambda_{AB} = L_{AB}/L^0_{AB}$, and $\kappa$ is a nonlinear function that resembles neo-Hookean behaviour

$$\kappa(\lambda_{AB}) = \frac{\lambda_{AB}^{0.5} + \lambda_{AB}^{-2.5}}{\lambda_{AB} + \lambda_{AB}^{-3}}. \tag{5.26}$$

Typicaly, one wants either nonlinear or linear behaviour and therefore one of $k_s, k_{slin}$ is zero. But the interaction will work with both constants non-zero.



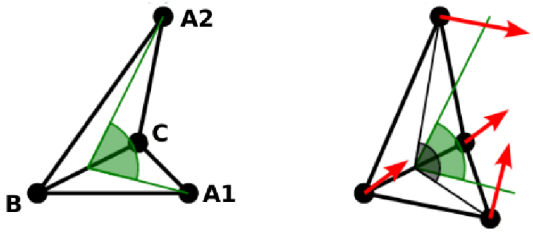

### Bending

The tendency of an elastic object to maintain the resting shape is achieved by prescribing the preferred angles between the neighbouring triangles of the mesh.

Denote the angle between two triangles in the resting shape by $\theta^0$. For closed immersed objects, one always has to set the inner angle. The deviation of this angle $\Delta\theta = \theta - \theta^0$ defines two bending forces for two triangles $A_1BC$ and $A_2BC$

$$F_{bi}(A_iBC) = k_b \frac{\Delta\theta}{\theta^0} n_{A_iBC} \tag{5.27}$$

Here, $n_{A_iBC}$ is the unit normal vector to the triangle $A_iBC$. The force $F_{bi}(A_iBC)$ is assigned to the vertex not belonging to the common edge. The opposite force divided by two is assigned to the two vertices lying on the common edge. This procedure is done twice, for $i = 1$ and for $i = 2$.




### Local area conservation

This interaction conserves the area of the triangles in the triangulation.

The deviation of the triangle surface $S_{ABC}$ is computed from the triangle surface in the resting shape $\Delta S_{ABC} = S_{ABC} - S^0_{ABC}$. The area constraint assigns the following shrinking/expanding force to every vertex

$$F_{al}(A) = -k_{al}\frac{\Delta S_{ABC}}{\sqrt{S_{ABC}}}w_A \tag{5.28}$$

where $k_{al}$ is the area constraint coefficient, and $w_A$ is the unit vector pointing from the centroid of triangle $ABC$ to the vertex $A$. Similarly the analogical forces are assigned to $B$ and $C$.

OIF local force is asymmetric. After creating the interaction

```
inter 33 oif_local_force 1.0 0.5 0.0 1.7 0.6 0.2 0.3 1.1
```

it is important how the bond is created. Particles need to be mentioned in the correct order. Command

```
part 0 bond 33 1 2 3
```

creates a bond related to the triangles 012 and 123. The particle 0 corresponds to point A1, particle 1 to C, particle 2 to B and particle 3 to A2. There are two rules that need to be fulfilled:

- there has to be an edge between particles 1 and 2
- orientation of the triangle 012, that is the normal vector defined as a vector product $01 \times 02$, must point to the inside of the immersed object.

Then the stretching force is applied to particles 1 and 2, with the relaxed length being 1.0. The bending force is applied to preserve the angle between triangles 012 and 123 with relaxed angle 1.7 and finally, local area force is applied to both triangles 012 and 123 with relaxed area of triangle 012 being 0.2 and relaxed area of triangle 123 being 0.3.

Notice that also concave objects can be defined. If $\theta_0$ is larger than $\pi$, then the inner angle is concave.

### 5.4.2. OIF global forces

*TCL Syntax*

`inter` *bondid* `oif_global_force` $S^0$ $k_{ag}$ $V^0$ $k_v$

Required features: `OIF_GLOBAL_FORCES`

*Description*

This type of interaction is available solely for closed 3D immersed objects.

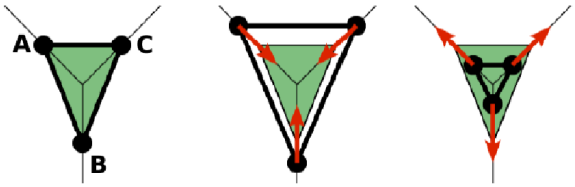
This interaction comprises two concepts: preservation of global surface and of volume of the object. Parameters $S^0, k_{ag}$ define preservation of the surface and parameters $V^0, k_v$ define volume preservation. They can be used together, or, by setting any $k_{ag}$ or $k_v$ to zero, the corresponding modulus can be turned off.

#### Global area conservation

Denote by $S$ the current surface of the immersed object, by $S_0$ the surface in the relaxed state and define $\Delta S = S - S_0$. The global area conservation force is defined as

$$F_{ag}(A) = -k_{ag}\frac{\Delta S}{S}w_A \tag{5.29}$$

Here, the above mentioned force divided by 3 is added to all three particles.




#### Volume conservation

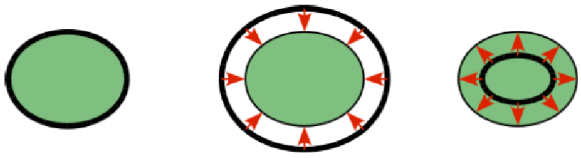
The deviation of the objects volume $V$ is computed from the volume in the resting shape $\Delta V = V - V^0$. For each triangle the following force is computed

$$F_v(ABC) = -k_v\frac{\Delta V}{V^0}S_{ABC}\ n_{ABC} \tag{5.30}$$

where $S_{ABC}$ is the area of triangle $ABC$, $n_{ABC}$ is the normal unit vector of plane $ABC$, and $k_v$ is the volume constraint coefficient. The volume of one immersed object is computed from

$$V = \sum_{ABC} S_{ABC}\ n_{ABC} \cdot h_{ABC} \tag{5.31}$$

where the sum is computed over all triangles of the mesh and $h_{ABC}$ is the normal vector from the centroid of triangle $ABC$ to any plane which does not cross the cell. The force $F_v(ABC)$ is equally distributed to all three vertices $A, B, C$.



This interaction is symmetric. After the definition of the interaction by

```
inter 22 oif_global_force 65.3 3.0 57.0 2.0
```

the order of vertices is crucial. By the following command the bonds are defined

```
part 0 bond 22 1 2
```

Triangle 012 must have correct orientation, that is the normal vector defined by a vector product $01 \times 02$. The orientation must point inside the immersed object.

### 5.4.3. Out direction

*TCL Syntax*

`inter` *bondid* `oif_out_direction`

Required features: `MEMBRANE_COLLISION`

*Description*

This type of interaction is primarily for closed 3D immersed objects to compute the input for membrane collision. After creating the interaction

```
inter 66 oif_out_direction
```

it is important how the bond is created. Particles need to be mentioned in the correct order. Command

```
part 0 bond 66 1 2 3
```

calculates the outward normal vector of triangle defined by particles 1, 2, 3 (these should be selected in such a way that particle 0 lies approximately at its centroid - for OIF objects, this is automatically handled by oif_create_template command, see Section 14.4.3). In order for the direction to be outward with respect to the underlying object, the triangle 123 needs to be properly oriented (as explained in the section on volume in oif_global_forces interaction).

## 5.5. Bond-angle interactions

*Python Syntax* (13)

```
espressomd.interactions.Angle_Cosine(
  K = ⟨float⟩,
  phi_0 = ⟨float⟩)
```

*TCL Syntax*

(1) `inter` *bondid* `angle_harmonic` $K$ $[\phi_0]$
(2) `inter` *bondid* `angle_cosine` $K$ $[\phi_0]$
(3) `inter` *bondid* `angle_cossquare` $K$ $[\phi_0]$

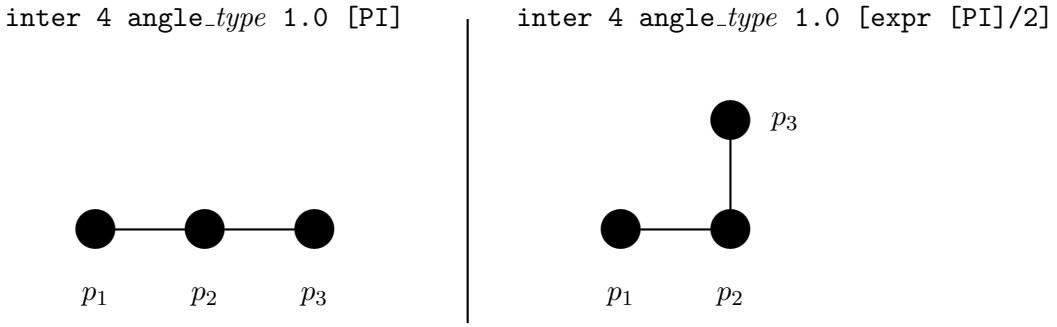
Required features: `BOND_ANGLE`

*Description*

This creates a bond type with identificator *bondid* with an angle dependent potential. This potential is defined between three particles. The particle for which the bond is created, is the central particle, and the angle $\phi$ between the vectors from this particle to the two others determines the interaction. $K$ is the bending constant, and the optional parameter $\phi_0$ is the equilibirum bond angle in radian ranging from 0 to $\pi$. If this parameter is not given, it defaults to $\phi_0 = \pi$, which corresponds to a stretched configuration. For example, for a bond defined by

```
part $p_2 bond 4 $p_1 $p_3
```

the minimal energy configurations are the following:




For the potential acting between the three particles three variants are possible

- Harmonic bond angle potential (1):
  A classical harmonic potential,

$$V(\phi) = \frac{K}{2}\left(\phi - \phi_0\right)^2 . \tag{5.32}$$

  Unlike the two following variants, this potential has a kink at $\phi = \phi_0 + \pi$ and accordingly a discontinuity in the force, and should therefore be used with caution.

- Cosine bond angle potential (2):

$$V(\alpha) = K\left[1 - \cos(\phi - \phi 0)\right] \tag{5.33}$$

  Around $\phi_0$, this potenial is close to a harmonic one (both are $1/2(\phi - \phi_0)^2$ in leading order), but it is periodic and smooth for all angles $\phi$.

- Cosine square bond angle potential (3):

$$V(\alpha) = \frac{K}{2}\left[\cos(\phi) - \cos(\phi_0)\right]^2 \tag{5.34}$$

This form is used for example in the GROMOS96 force field. The potential is $1/8(\phi - \phi_0)^4$ around $\phi_0$, and therefore much flatter than the two potentials before.

## 5.6. Dihedral interactions

*TCL Syntax*

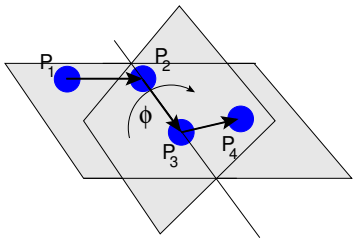
`|inter` *bondid* `dihedral` $n$ $K$ $p$

*Description*

This creates a bond type with identificator *bondid* with a dihedral potential, *i.e.* a four-body-potential. In the following, let the particle for which the bond is created be particle $p_2$, and the other bond partners $p_1$, $p_3$, $p_4$, in this order, *i.e.* `part` $p_2$ `bond bondid` $p_1$ $p_3$ $p_4$. Then, the dihedral potential is given by

$$V(\phi) = K\left[1 - \cos(n\phi - p)\right], \tag{5.35}$$

where $n$ is the multiplicity of the potential (number of minimas) and can take any integer value (typically from 1 to 6), $p$ is a phase parameter and $K$ is the bending constant of the potential. $\phi$ is the dihedral angle between the particles defined by the particle quadrupel $p_1$, $p_2$, $p_3$ and $p_4$, *i.e.* the angle between the planes defined by the particle triples $p_1$, $p_2$ and $p_3$ and $p_2$, $p_3$ and $p_4$:




Together with appropriate Lennard-Jones interactions, this potential can mimic a large number of atomic torsion potentials.

If you enable the feature OLD_DIHEDRAL, then the old, less general form of the potential is used:

$$V(\phi) = K\left[1 + p\,\cos(n\phi)\right], \tag{5.36}$$

where $p$ is rather a phase factor and can only take values $p = \pm 1$.

## 5.7. Coulomb interaction

*TCL Syntax*

(1) `inter coulomb 0.0`
(2) `inter coulomb`
(3) `inter coulomb` *parameters*

*Description*

These commands allow to set up the calculation of the Coulomb interaction. The Coulomb (or electrostatic) interaction is defined as follows. For a pair of particles at distance $r$ with charges $q_1$ and $q_2$, the interaction is given by

$$U^C(r) = l_B k_B T \frac{q_1 q_2}{r}. \tag{5.37}$$

where $l_B = e_o^2/(4\pi\epsilon k_B T)$ denotes the Bjerrum length, which measures the strength of the electrostatic interaction. As a special case, when the internal variable *temperature* is set to zero, the value of bjerrum length you enter is treated as $l_B k_B T$ rather than $l_B$. This occurs when the thermostat is switched off and ESPResSo performs an NVE integration (see also Section 6.2).

Computing electrostatic interactions is computationally very expensive. ESPResSo features some state-of-the-art algorithms to deal with these interactions as efficiently as possible, but almost all of them require some knowledge to use them properly. Uneducated use can result in completely unphysical simulations.

Variant (1) disables Coulomb interactions. Variant (2) returns the current parameters of the coulomb interaction as a Tcl-list using the same syntax as used to setup the method, *e.g.*

```
{coulomb 1.0 p3m 7.75 8 5 0.1138 0.0}
{coulomb epsilon 0.1 n_interpol 32768 mesh_off 0.5 0.5 0.5}
```

Variant (3) is the generic syntax to set up a specific method or its parameters, the details of which are described in the following subsections. Note that using the electrostatic interaction also requires assigning charges to the particles. This is done using the `part` command to set the charge `q`, *e.g.*

```
inter coulomb 1.0 p3m tune accuracy 1e-4
part 0 q 1.0; part 1 q -1.0
```

### 5.7.1. Coulomb P3M

*Python Syntax* (14)

```
electrostatics.P3M(
  bjerrum_length = ⟨float⟩,
  accuracy = ⟨float⟩)
electrostatics.P3M(
  bjerrum_length = ⟨float⟩,
  r_cut = ⟨float⟩,
  mesh = ⟨array of 3 ints⟩,
  cao = ⟨int⟩,
  alpha = ⟨float⟩,
  alpha = ⟨float⟩,
  epsilon = metallic | ⟨float⟩,
  inter = ⟨int⟩,
  mesh_off = ⟨array of 3 floats⟩)
```

Required features: `ELECTROSTATICS`

*Python Syntax* (15)

```
electrostatics.P3M_GPU(
  bjerrum_length = ⟨float⟩,
  accuracy = ⟨float⟩)
electrostatics.P3M_GPU(
  bjerrum_length = ⟨float⟩ r_cut = ⟨float⟩,
  mesh = ⟨array of 3 ints⟩,
  cao = ⟨int⟩,
  alpha = ⟨float⟩,
  alpha = ⟨float⟩,
  epsilon = metallic | ⟨float⟩,
  inter = ⟨int⟩,
  mesh_off = ⟨array of 3 floats⟩)
```

Required features: `ELECTROSTATICS` `CUDA`

*TCL Syntax*

`inter coulomb` $l_B$ `p3m [gpu]` $r_{\text{cut}}$ ( *mesh* | {$mesh_x$ $mesh_y$ $mesh_z$} ) *cao* *alpha*

Required features: `ELECTROSTATICS`

*Description*

For this feature to work, you need to have the `fftw3` library installed on your system. In ESPResSo, you can check if it is compiled in by checking for the feature `FFTW`.

This command activates the P3M method to compute the electrostatic interactions between charged particles. P3M requires full periodicity (1 1 1). The different parameters are described in more detail in [19].

[*gpu*] The optional flag gpu causes the far field portion of p3m to be calculated on the GPU. It should be noted that this does not always provide significant increase in performance. Furthermore it computes the far field interactions with only single precision which limits the maximum precision. Furthermore the algorithm does not work in combination with certain other methods implemented in ESPResSo and only for the case of cubic boxes.

$r_{\text{cut}}$ The real space cutoff as a positive floating point number.

*mesh* The number of mesh points, as a single positive integer.

$mesh_{x,y,z}$ The number of mesh points in x, y and z direction. This is relevant for noncubic boxes.

*cao* The *charge-assignment order*, an integer between 0 and 7.

*alpha* The Ewald parameter as a positive floating point number.

Make sure that you know the relevance of the P3M parameters before using P3M! If you are not sure, read the following references [24, 30, 36, 18, 19, 20, 17, 12].

### Tuning Coulomb P3M

*TCL Syntax*

`inter coulomb` $l_B$ `p3m [gpu] ( tune | tunev2 ) accuracy` *accuracy* [`r_cut` $r_{\text{cut}}$] [`mesh` *mesh*] [`cao` *cao*] [`alpha` $\alpha$]

Required features: `ELECTROSTATICS`

*Description*

It is not easy to calculate the various parameters of the P3M method such that the method provides the desired accuracy at maximum speed. To simplify this, ESPResSo provides a function to automatically tune the algorithm. Note that for this function to work properly, your system should already contain an initial configuration of charges and the correct initial box size. Also note that both provided tuning algorithms work very well on homogenous charge distributions, but might not achieve the requested precision for highly inhomogenous or symmetric systems. For example, because of the nature of the P3M algorithm, systems are problematic where most charges are placed in one plane, one small region, or on a regular grid.

The function employs the analytical expression of the error estimate for the P3M method [30] and its real space error [36] to obtain sets of parameters that yield the desired accuracy, then it measures how long it takes to compute the coulomb interaction using these parameter sets and chooses the set with the shortest run time.

The function will only automatically tune those parameters that are not set to a predetermined value using the optional parameters of the tuning command.

The two tuning methods follow different methods for determining the optimal parameters. While the `tune` version tests different values on a grid in the parameter space,

the `tunev2` version uses a bisection to determine the optimal parameters. In general, for small systems the `tune` version is faster, while for large systems `tunev2` is faster. The results of `tunev2` are always at least as good as the parameters from the `tune` version, and normally the obtained accuracy is much closer to the desired value.

During execution the tuning routines report the tested parameter sets, the corresponding k-space and real-space errors and the timings needed for force calculations (the setmd variable *timings* controls the number of test force calculations). Since the error depends on $r_{\text{cut}}/box_l$ and $\alpha box_l$ the output is given in these units.

Note that the previous setting of $r_{\text{cut}}$, *cao* and *mesh* will be remembered. If you want to retune your electrostatics, *e.g.* after a major system change, you should use

`inter coulomb` $l_B$ `p3m tune accuracy` *acc* `r_cut 0 mesh 0 cao 0`

#### Additional P3M parameters

*TCL Syntax*

`inter coulomb` [`epsilon` ( `metallic` | *epsilon* )] [`n_interpol` *points*] [`mesh_off` *xoff yoff zoff*]

*Description*

Once P3M algorithm has been set up, it is possible to set some additional P3M parameters with this command. The different parameters have the following meaning:

`epsilon` *epsilon* The dielectric constant of the surrounding medium, metallic (*i.e.* infinity) or some finite positive number. Defaults to `metallic`.

`n_interpol` $n_i nterpol$ Number of interpolation points for the charge assignment function. When this is set to 0, interpolation is turned off and the function is computed directly. Defaults to 32768.

`mesh_off` $mesh_o ff$ Offset of the first mesh point from the lower left corner of the simulation box in units of the mesh constant. Defaults to `0.5 0.5 0.5`.

### 5.7.2. Coulomb Ewald GPU

*Python Syntax* (16)

```
electrostatics.EwaldGpu(
  bjerrum_length = ⟨float⟩,
  accuracy = ⟨float⟩,
  precision = ⟨float⟩,
  K_max = ⟨int⟩ | ⟨array of 3 ints⟩,
  alpha = ⟨float⟩)
```

Required features: `ELECTROSTATICS CUDA EWALD_GPU`

*TCL Syntax*

> `inter coulomb` $l_B$ `ewaldgpu` $r_{\text{cut}}$ ( $K_{\text{cut}}$ | $\{K_{\text{cut,x}}$ $K_{\text{cut,y}}$ $K_{\text{cut,x}}\}$ ) *alpha*
>
> Required features: `ELECTROSTATICS`

*Description*

This command activates the Ewald method to compute the electrostatic interactions between charged particles. The far field is computed by the GPU with single precision and the near field by the CPU with double precision. It only works for the case of cubic boxes.

$l_B$ Bjerrum length as positive floating point number

$r_{\text{cut}}$ Real space cutoff as positive floating point number

$K_{\text{cut}}$ Reciprocal space cutoff as single positive integer

$K_{\text{cut,xyz}}$ Reciprocal space cutoff in x, y and z direction (relevant for noncubic boxes)

*alpha* Ewald parameter as positive floating point number

### Tuning Ewald GPU

*TCL Syntax*

> `inter coulomb` $l_B$ `ewaldgpu tune accuracy` *accuracy* `precision` *precision* `K_max` $K_{\text{max}}$
>
> Required features: `ELECTROSTATICS`

*Description*

The tuning algorithm first computes the optimal $r_{\text{cut}}$ and *alpha* for every $K_{\text{cut}}$ between one and $K_{\text{max}}$ as described in [36]. Then the performance for all those ($K_{\text{cut}}$, $r_{\text{cut}}$, *alpha*)-triplets will be measured via a short test simulation and the fastest will be chosen.

*accuracy* Maximal allowed root mean square error regarding the forces

*precision* Determines how precise alpha will be computed

$K_{\text{max}}$ Maximal reciprocal space cutoff $K_{\text{cut}}$ to be tested in the tuning algorithm

### Tuning Alpha Ewald GPU

*TCL Syntax*

> `inter coulomb` $l_B$ `ewaldgpu tunealpha` $r_{\text{cut}}$ ( $K_{\text{cut}}$ | $\{K_{\text{cut,x}}$ $K_{\text{cut,y}}$ $K_{\text{cut,x}}\}$ ) *precision*
>
> Required features: `ELECTROSTATICS`

*Description*

If $K_{\text{cut}}$ and $r_{\text{cut}}$ are given by the user, then `tunealpha` computes the optimal *alpha* with the chosen *precision* as described in [36]. But in general `tune` should be chosen for tuning.

### 5.7.3. Debye-Hückel potential

*Python Syntax* (17)

```
electrostatics.DH(
  bjerrum_length = ⟨float⟩,
  kappa = ⟨float⟩,
  r_cut = ⟨float⟩)
```

Required features: `ELECTROSTATICS`

```
electrostatics.CDH(
  bjerrum_length = ⟨float⟩,
  kappa = ⟨float⟩,
  r_cut = ⟨float⟩,
  eps_int = ⟨float⟩,
  eps_ext = ⟨float⟩,
  r0 = ⟨float⟩,
  r1 = ⟨float⟩,
  alpha = ⟨float⟩)
```

Required features: `ELECTROSTATICS` `COULOMB_DEBYE_HUECKEL`

*TCL Syntax*

(1) `inter coulomb` $l_\mathrm{B}$ `dh` $\kappa$ $r_\mathrm{cut}$ [1]
(2) `inter coulomb` $l_\mathrm{B}$ `dh` $\kappa$ $r_\mathtt{cut}$ $\varepsilon_\mathtt{int}$ $\varepsilon_{ext}$ $r_0$ $r_1$ $\alpha$ [1,2]
Required features: [1] `ELECTROSTATICS` [2] `COULOMB_DEBYE_HUECKEL`

*Description*

NOTE: the two variants are mutually exclusive. If "COULOMB_DEBYE_HUECKEL" is defined in the configuration file, variant (1) would not work.

Defines the electrostatic potential by

$$U^{C-DH} = l_B k_B T \frac{q_1 q_2 exp(-\kappa r)}{r} \quad \text{for} \quad r < r_\mathrm{cut} \tag{5.38}$$

The Debye-Hückel potential is an approximate method for calculating electrostatic interactions, but technically it is treated as other short-ranged non-bonding potentials. For $r > r_{cut}$ it is set to zero which introduces a step in energy. Therefore, it introduces fluctuations in energy.

For $\kappa = 0$, this corresponds to the plain coulomb potential.

The second variant combines the coulomb interaction for charges that are closer than $r_0$ with the Debye-Hueckel approximation for charges that are further apart than $r_1$ in

a continous way. The used potential is

$$U(r)^{C-DHC} = \begin{cases} \frac{l_B k_B T q_1 q_2}{\varepsilon_{\text{int}} r} & \text{if } r < r_0, \\ \frac{l_B k_B T q_1 q_2 e^{-\alpha(r-r_0)}}{\varepsilon_{\text{int}} r} & \text{if } r_0 < r < r_1, \\ \frac{l_B k_B T q_1 q_2 e^{-\kappa r}}{\varepsilon_{\text{ext}} r} & \text{if } r_{\text{cut}} > r > r_1, \\ 0 & \text{if } r > r_{\text{cut}}. \end{cases} \tag{5.39}$$

The parameter $\alpha$ that controlls the transition from Coulomb- to Debye-Hückel potential should be chosen such that the force is continous.

### 5.7.4. MMM2D

Please cite [5] (BibTeX-key `mmm2d` in file `doc/ug/citations.bib`) when using MMM2D, and [63] (BibTeX-key `icmmm2d` in file `doc/ug/citations.bib`) when using dielectric interfaces.

*TCL Syntax*

`inter coulomb` $l_B$ `mmm2d` *maximal_pairwise_error* [*fixed_far_cutoff*] [`dielectric` $\epsilon_t$ $\epsilon_m$ $\epsilon_b$] [`dielectric-contrasts` $\Delta_t$ $\Delta_b$] [`capacitor` $U$]

Required features: `ELECTROSTATICS`

*Description*

*Python Syntax* (18)

```
electrostatics.MMM2D(
  bjerrum_length = ⟨float⟩,
  maxPWerror = ⟨float⟩,
  dielectric = 1,
  top = ⟨float⟩,
  mid = ⟨float⟩,
  bot = ⟨float⟩,
  far_cut = ⟨float⟩)
electrostatics.MMM2D(
  bjerrum_length = ⟨float⟩,
  maxPWerror = ⟨float⟩,
  capacitor = 1,
  pot_diff = ⟨float⟩,
  far_cut = ⟨float⟩)
electrostatics.MMM2D(
  bjerrum_length = ⟨float⟩,
  maxPWerror = ⟨float⟩,
  dielectric_contrast_on = 1,
  delta_mid_top = ⟨float⟩,
  delta_mid_bot = ⟨float⟩,
  far_cut = ⟨float⟩)
```

MMM2D coulomb method for systems with periodicity 1 1 0. Needs the layered cell system. The performance of the method depends on the number of slices of the cell system, which has to be tuned manually. It is automatically ensured that the maximal pairwise error is smaller than the given bound. The far cutoff setting should only be used for testing reasons, otherwise you are more safe with the automatical tuning. If you even don't know what it is, do not even think of touching the far cutoff. For details on the MMM family of algorithms, refer to appendix E on page 281.

The last two, mutually exclusive arguments "dielectric" and "dielectric-constants" allow to specify dielectric contrasts at the upper and lower boundaries of the simulation box. The first form specifies the respective dielectric constants in the media, which however is only used to calculate the contrasts. That is, specifying $\epsilon_t = \epsilon_m = \epsilon_b = \text{const}$ is always identical to $\epsilon_t = \epsilon_m = \epsilon_b = 1$. The second form specifies only the dielectric contrasts at the boundaries, that is $\Delta_t = \frac{\epsilon_m - \epsilon_t}{\epsilon_m + \epsilon_t}$ and $\Delta_b = \frac{\epsilon_m - \epsilon_b}{\epsilon_m + \epsilon_b}$. Using this form allows to choose $\Delta_{t/b} = -1$, corresponding to metallic boundary conditions.

Using `capacitor` $U$ allows to maintain a constant electric potential difference $U$ between the xy-plane at $z = 0$ and $z = L$, where $L$ denotes the box length in $z$-direction. This is done by countering the total dipol moment of the system with the electric field $E_{induced}$ and superposing a homogeneous electric field $E_{applied} = \frac{U}{L}$ to retain $U$. This mimics the induction of surface charges $\pm\sigma = E_{induced} \cdot \epsilon_0$ for planar electrodes at $z = 0$ and $z = L$ in a capacitor connected to a battery with voltage $U$. Using *capacitor* 0 is equivalent to $\Delta_{t/b} = -1$.

*TCL Syntax*

`efield_caps ( total | induced | applied )`

Required features: `ELECTROSTATICS`

*Description*

The electric fields added by *capacitor* $U$ can be obtained by calling the above command, where *induced* returns $E_{induced}$, *applied* returns $E_{applied}$ and *total* their sum.

### 5.7.5. MMM1D

Please cite [3] (BibTeX-key `mmm1d` in file `doc/ug/citations.bib`) when using MMM1D.

*Python Syntax* (19)

```
electrostatics.MMM1D(
  bjerrum_length=⟨float⟩,
  maxPWerror=⟨float⟩,
  far_switch_radius=⟨float⟩)
```

Required features: `ELECTROSTATICS PARTIAL_PERIODIC`

*TCL Syntax*

(1) `inter coulomb` $l_B$ `mmm1d` *switch_radius maximal_pairwise_error*

(2) `inter coulomb` $l_B$ `mmm1d tune` *maximal_pairwise_error*

Required features: `ELECTROSTATICS PARTIAL_PERIODIC`

*Description*

MMM1D coulomb method for systems with periodicity 0 0 1. Needs the nsquared cell system (see section 6.4 on page 110). The first form sets parameters manually. The switch radius determines at which xy-distance the force calculation switches from the near to the far formula. The Bessel cutoff does not need to be specified as it is automatically determined from the particle distances and maximal pairwise error. The second tuning form just takes the maximal pairwise error and tries out a lot of switching radii to find out the fastest one. If this takes too long, you can change the value of the setmd variable `timings`, which controls the number of test force calculations.

*Python Syntax* (20)

```
electrostatics.MMM1D_GPU(
  bjerrum_length = ⟨float⟩,
  maxPWerror = ⟨float⟩,
  far_switch_radius = ⟨float⟩,
  bessel_cutoff = ⟨int⟩)
```

Required features: `ELECTROSTATICS CUDA PARTIAL_PERIODIC MMM1D_GPU`

*TCL Syntax*

(1) `inter coulomb` $l_B$ `mmm1dgpu` *switch_radius* [*bessel_cutoff*] *maximal_pairwise_error*

(2) `inter coulomb` $l_B$ `mmm1dgpu tune` *maximal_pairwise_error*

Required features: `CUDA ELECTROSTATICS PARTIAL_PERIODIC MMM1D_GPU`

*Description*

MMM1D is also available in a GPU implementation. Unlike its CPU counterpart, it does not need the nsquared cell system. The first form sets parameters manually. The switch radius determines at which xy-distance the force calculation switches from the near to the far formula. If the Bessel cutoff is not explicitly given, it is determined from the maximal pairwise error, otherwise this error only counts for the near formula. The second tuning form just takes the maximal pairwise error and tries out a lot of switching radii to find out the fastest one.

For details on the MMM family of algorithms, refer to appendix E on page 281.

### 5.7.6. Maxwell Equation Molecular Dynamics (MEMD)

*TCL Syntax*

`inter coulomb` $l_B$ `memd` *f_mass mesh* [`epsilon` $\varepsilon_\infty$]

Required features: `ELECTROSTATICS`

*Description*

This is an implementation of the instantaneous 1/r Coulomb interaction

$$U = l_B k_B T \frac{q_1 q_2}{r} \tag{5.40}$$

as the potential of mean force between charges which are dynamically coupled to a local electromagnetic field.

The algorithm currently works with the following constraints:

- cellsystem has to be domain decomposition but *without* Verlet lists!
- system has to be periodic in three dimensions.

*Arguments*

- *f_mass* is the mass of the field degree of freedom and equals to the square root of the inverted speed of light.
- *mesh* is the number of mesh points for the interpolation of the electromagnetic field in one dimension.
- $\varepsilon_\infty$ is the background dielectric permittivity at infinity. This defaults to metallic boundary conditions, to match the results of P3M.

The arising self-interactions are treated with a modified version of the exact solution of the lattice Green's function for the problem.

Currently, forces have large errors for two particles within the same lattice cube. This may be fixed in future development, but right now leads to the following rule of thumb for the parameter choices:

- The lattice should be of the size of your particle size (i.e. the lennard jones epsilon). That means: mesh $\approx$ box_l/lj_sigma
- The integration timestep should be in a range where no particle moves more than one lattice box (i.e. lennard jones sigma) per timestep.
- The speed of light should satisfy the stability criterion $c \ll a/dt$, where $a$ is the lattice spacing and $dt$ is the timestep. For the second parameter, this means f_mass $\gg dt^2/a^2$.

The main error of the MEMD algorithm stems from the lattice interpolation and is proportional to the lattice size in three dimensions, which means $\Delta_{\text{lattice}} \propto a^3$.

Without derivation here, the algorithmis error is proportional to $1/c^2$, where $c$ is the adjustable speed of light. From the stability criterion, this yields

$$\Delta_{\text{maggs}} = A \cdot a^3 + B \cdot dt^2/a^2 \tag{5.41}$$

This means that increasing the lattice will help the algorithmic error, as we can tune the speed of light to a higher value. At the same time, it increases the interpolation error at an even higher rate. Therefore, momentarily it is advisable to choose the lattice with a rather fine mesh of the size of the particles. As a rule of thumb, the error will then be less than $10^{-5}$ for the particle force.

For a more detailed description of the algorithm, see appendix D on page 275 or the publications [41, 46].

### Spatially varying dielectrics with MEMD

Since MEMD is a purely local algorithm, one can apply local changes to some properties and the propagation of the Coulomb force is still valid. In particular, it is possible to arbitrarily select the dielectric permittivity on each site of the interpolating lattice.

*TCL Syntax*

> `inter coulomb` $l_B$ `memd localeps node` *node_x node_y node_z* `dir` $X/Y/Z$ `eps` $\varepsilon$
>
> Required features: `ELECTROSTATICS`

*Description*

The keyword `localeps` after the `inter coulomb` command offers the possibility to assign any value of $\varepsilon$ to any lattice site.

*Arguments*

- $l_B$ is the bjerrum length of the background. It defines the reference value $\varepsilon_{\text{bg}}$ via the formula (5.42). This is a global variable.
- *node_x* is the index of the node in $x$ direction that should be changed
- *node_y* is the index of the node in $y$ direction that should be changed
- *node_z* is the index of the node in $z$ direction that should be changed
- $X/Y/Z$ is the direction in which the lattice site to be changed is pointing. Has to be one of the three (X, Y or Z).
- $\varepsilon$ is the relative permittivity change in respect to the background permittivity set by the parameter $l_B$.

The permittivity on each lattice site is set relatively. By defining the (global) bjerrum length of the system, the reference permittivity $\varepsilon$ is fixed via the formula

$$l_B = e^2/(4\pi\varepsilon k_B T) \tag{5.42}$$

The local changes of $\varepsilon$ are in reference to this value and can be seen as a spatially dependent prefactor to this epsilon. If left unchanged, this prefactor is 1.0 for every site by default.

### Adaptive permittivity with MEMD

In addition to setting the local permittivity manually as described in section 5.7.6, MEMD is capable of adapting the local permittivity at each lattice site, dependent on the concentration of surrounding charges. More information on this can be found in article [25], which you should cite if you use this algorithm.

To achieve this, the local salt concentration around each lattice cell is measured and then mapped to an according dielectric permittivity using the empirical formula

$$\varepsilon = \frac{78.5}{1 + 0.278 \cdot C}, \tag{5.43}$$

where $C$ is the concentration in molar [M], or moles per liter [mol/l]. The algorithm averages over a volume of $7^3$ lattice cubes and expects a concentration in molar within the simulation. In more MD-friendly units, this would mean that the units expected by the formula correspond to a lattice size of roughly 0.6 nanometers for MEMD. Any other length unit is possible but needs to be scaled by a prefactor. This is perfectly reasonable and will not break the algorithm, since the permittivity $\varepsilon$ is dimensionless. The scaling factor $S_{\text{adaptive}}$ is thus defined via the used MEMD lattice spacing $a_{\text{used}}$:

$$S_{\text{adaptive}} \times a_{\text{used}} = 0.6\,\text{nm} \tag{5.44}$$

To use MEMD with adaptive permittivity to calculate Coulomb interactions in the system, use the following command.

*TCL Syntax*

`inter coulomb` $l_B$ `memd adaptive` *scaling* `parameters` $f_mass$ *mesh*

Required features: `ELECTROSTATICS`

*Description*

The keyword `adaptive` after the `inter coulomb` command will use the implementation with dielectric permittivity dependent on the local salt concentration.

*Arguments*

- $l_B$ is the bjerrum length of the background. It defines the reference value $\varepsilon_{\text{bg}}$ via the formula (5.42). Since the permittivity in this case is set adaptively, it essentially determined the temperature for the Coulomb interaction. This is a global variable and for this particular algorithm should most likely be set as the permittivity of pure water.
- *scaling* is the scaling of the used length unit to match the expected unit system. For more details see equation 5.44 and the paragraph before.
- $f_mass$ is the mass of the field degree of freedom and equals to the square root of the inverted speed of light.
- *mesh* is the number of mesh points for the interpolation of the electromagnetic field in one dimension.

It should be mentioned that this algorithm is not a black box and should be understood to a degree if used. Small changes in the parameters, especially the mesh size, can quickly lead to unphysical results. This is not only because of the retarded electrodynamics solution offered by the MEMD algorithm in general, but because of the sensitivity of the dielectric response to the volume over which the local salt concentration is sampled. If this volume is set too small, harsh changes in the local dielectric properties can occur and the algorithm may become unstable, or worse, produce incorrect electrostatic forces.

The calculation of local permittivity will for the same parameters – depending on your computer – run roughly a factor of 2 to 4 longer than MEMD without temporally varying dielectric properties.

### 5.7.7. Scafacos

Espresso can use the electrostatics methods from the Scafacos *Scalable fast Coulomb solvers* library.

*TCL Syntax*

`scafacos_methods`

Required features: `ELECTROSTATICS SCAFACOS`

*Description*

*Python Syntax* (21)

`scafacos.available_methods()`

Required features: `ELECTROSTATICS SCAFACOS` This shows the methods available at the compile time of ESPResSo. Scafacos can be used as Coulomb solver for the system.

*TCL Syntax*

`inter coulomb` $l_b$ `scafacos` *method* [*parameters*] [`tolerance_field` *prec*]

Required features: `ELECTROSTATICS SCAFACOS`

*Description*

*Python Syntax* (22)

```
electrostatics.Scafacos(
  bjerrum_length=⟨double⟩,
  method_name=⟨string⟩,
  method_params=⟨dict⟩)
```

Required features: `SCAFACOS`

Here *method* is a scafacos method as returned by `scafacos_methods`. `tolerance_field` sets the desisired rms accuracy for the electric field if supported by the method. *parameters* is a list of parameters as described by the Scafacos manual. If parameters of the solver are not set, and the method supports it, the open parameteres are tuned. To use the `ewald` solver from scafacos as electrostatics solver for your system, set its cutoff to 1.5 and tune the other parameters for an accuracy of $10^{-3}$, use the command

```
inter coulomb 1.0 scafacos ewald ewald_r_cut 1.5 tolerance_field 1e-3
```

For details of the various methods and their parameters please refer to the Scafacos manual. Note that the `SCAFACOS` feature is only available if you build with cmake. You need to build Scafacos as a shared library. Scafacos can be used only once, either for coulomb or for dipolar interactions.

### 5.7.8. Electrostatic Layer Correction (ELC)

Please cite [6] (BibTeX-key `elc` in file `doc/ug/citations.bib`) when using ELC, and in addition [64] (BibTeX-key `icelc` in file `doc/ug/citations.bib`) if you use dielectric interfaces.

*Python Syntax* (23)

```
espressomd.electrostatic_extensions.ELC(
  gap_size = ⟨float⟩,
  maxPWerror = ⟨float⟩,
  neutralize = ⟨bool⟩,
  far_cut = ⟨float⟩)
```

*TCL Syntax*

```
inter coulomb elc maximal_pairwise_error gap_size
      [far_cutoff] [noneutralization] [dielectric ϵ_t ϵ_m ϵ_b]
      [dielectric-contrasts Δ_t Δ_b] [capacitor U]
```

Required features: `ELECTROSTATICS`

*Description*

This is a special procedure that converts a 3d method, to a 2d method, in computational order N. Currently, it only supports P3M. This means, that you will first have to set up the P3M algorithm (via `inter coulomb p3m` *params*) before using ELC. The algorithm is definitely faster than MMM2D for larger numbers of particles ($> 400$ at reasonable accuracy requirements). The maximal pairwise error *maximal_pairwise_error* sets the LUB error of the force between any two charges without prefactors (see the papers). The algorithm tries to find parameters to meet this LUB requirements or will throw an error if there are none.

The gap size *gap_size* gives the height of the empty region between the system box and the neighboring artificial images (again, see the paper). ESPResSo does not make sure that the gap is actually empty, this is the users responsibility. The method will compute fine of the condition is not fulfilled, however, the error bound will not be reached. Therefore you should really make sure that the gap region is empty (e. g. by constraints).

The setting of the far cutoff *far_cutoff* is only intended for testing and allows to directly set the cutoff. In this case, the maximal pairwise error is ignored. The periodicity has to be set to `1 1 1` still, and the 3d method has to be set to epsilon metallic, i.e. metallic boundary conditions. For details, see appendix E on page 281.

By default, ELC just as P3M adds a homogeneous neutralizing background to the system in case of a net charge. However, unlike in three dimensions, this background adds a parabolic potential across the slab [8]. Therefore, under normal circumstance, you will probably want to disable the neutralization using [noneutralization]. This corresponds then to a formal regularization of the forces and energies [8]. Also, if you add neutralizing walls explicitely as constraints, you have to disable the neutralization.

The dielectric contrast features work exactly the same as for MMM2D, see the documentation above. Same accounts for `capacitor` $U$, but the constant potential is maintained between the xy-plane at $z = 0$ and $z = L - gap_size$. The command *efield_caps*

to read out the electric fields added by *capacitor U* also applies for the capacitor-feature of ELC.

Make sure that you read the papers on ELC ([6, 64]) before using it.

### 5.7.9. Dielectric interfaces with the ICC⋆ algorithm

*TCL Syntax*

> `iccp3m` *n_induced_charges* `convergence` *convergence_criterion* `areas` *areas* `normals` *normals* `sigmas` *sigmas* `epsilons` *epsilons* [`eps_out` *eps_out* ] [`relax` *relaxation_parameter* ] [`max_iterations` *max_iterations* ] [`ext_field` *ext_field*]
>
> Required features: `ELECTROSTATICS`

*Description*

The ICC⋆ algorithm allows to take into account arbitrarily shaped dielectric interfaces. This is done by iterating the charge on the particles with the ids 0 to *n_induced_particles − 1* until the correctly represent the influence of the dielectric discontinuity. It relies on a coulomb solver that is already initialized. This Coulomb solver can be P3M, P3M+ELC, MMM2D or MMM1D. As most of the times, ICC⋆ will be used with P3M the corresponding command is called `iccp3m`.

Please make sure to read the corresponding articles, mainly[7, 62, 35] before using it.

The particles with ids 0 to *n_induced_particles − 1* are treated as iterated particles by ICC⋆. The constitute the dielectric interface and should be fixed in space. The parameters *areas* and *epsilons* are Tcl lists containing one floating point number describing each surface elements area and dielectric constant. *sigmas* allows to take into account a (bare) charge density, thus a surface charge density in absence of any charge induction. *normals* is a Tcl list of Tcl lists with three floating point numbers describing the outward pointing normal vectors for every surface element. The parameter *convergence_criterion* allows to specify the accuracy of the iteration. It corresponds to the maximum relative change of any of the interface particle's charge. After *max_iterations* the iteration stops anyways. The dielectric constant in bulk, i. e. outside the dielectric walls is specified by *eps_out*. A homogenous electric field can be added to the calculation of dielectric boundary forces by specifying it in the parameter *ext_field*.

### Quick setup of dielectric interfaces

*TCL Syntax*

(1) `dielectric sphere center` *cx cy cz* `radius` *r* `res` *res*
(2) `dielectric wall normal` *nx ny nz* `dist` *d* `res` *res*
(3) `dielectric cylinder center` *cx cy cz* `axis` *ax ay az* `radius` *r* `direction` *d*
(4) `dielectric pore center` *cx cy cz* `axis` *ax ay az* `radius` *r* `length` *l* `smoothing_radius` *rs* `res` *res*
(5) `dielectric slitpore pore_mouth` *z* `channel_width` *c* `pore_width` *w* `pore_length` *l* `upper_smoothing_radius` *us* `lower_smoothing_radius` *ls*

*Description*

The command `dielectric` allows to conveniently create dielectric interfaces similar to the constraint and the lbboundary command. Currently the creation of spherical, cylindrical and planar geometries as well as a pore and slitpore geometry is supported. Please check the documentation of the corresponding constraint for the detailed geometry. It is implemented in Tcl and places particles in the right positions and adds the correct values to the global Tcl variables *icc_areas icc_normals icc_sigmas icc_epsilons* and increases the global Tcl variable varn_induced_charges. Thus after setting up the shapes, it is still necessary to register them by calling `iccp3m`, usually in the following way:

```
iccp3m $n_induced_charges epsilons $icc_epsilons normals
$icc_normals areas $icc_areas sigmas $icc_sigmas
```

## 5.8. Dipolar interaction

*TCL Syntax*

(1) `inter magnetic 0.0`
(2) `inter magnetic`
(3) `inter magnetic` *parameters*

*Description*

These commands can be used to set up magnetostatic interactions, which is defined as follows:

$$U^{D-P3M}(\vec{r}) = l_B k_B T \left( \frac{(\vec{\mu}_i \cdot \vec{\mu}_j)}{r^3} - \frac{3(\vec{\mu}_i \cdot \vec{r})(\vec{\mu}_j \cdot \vec{r})}{r^5} \right) \tag{5.45}$$

where $r = |\vec{r}|$.

$l_B$ is a dimensionless parameter similar to the Bjerrum length in electrostatics which helps to tune the effect of the medium on the magnetic interaction between two magnetic dipoles.

Computing magnetostatic interactions is computationally very expensive. ESPResSo features some state-of-the-art algorithms to deal with these interactions as efficiently as

possible, but almost all of them require some knowledge to use them properly. Uneducated use can result in completely unphysical simulations.

The commands above work as their couterparts for the electrostatic interactions (see section 5.7.1 on page 78). Variant (1) disables dipolar interactions. Variant (2) returns the current parameters of the dipolar interaction as a Tcl-list using the same syntax as used to setup the method, *e.g.*

```
{coulomb 1.0 p3m 7.75 8 5 0.1138 0.0}
{coulomb epsilon 0.1 n_interpol 32768 mesh_off 0.5 0.5 0.5}
```

Variant (3) is the generic syntax to set up a specific method or its parameters, the details of which are described in the following subsections. Note that using the magnetostatic interaction also requires assigning dipole moments to the particles. This is done using the `part` command to set the dipole moment `dip`, *e.g.*

```
inter coulomb 1.0 p3m tune accuracy 1e-4
part 0 dip 1 0 0; part 1 dip 0 0 1
```

### 5.8.1. Dipolar P3M

*TCL Syntax*

`inter magnetic` $l_B$ `p3m` $r_{\text{cut}}$ *mesh cao alpha*

Required features: `DIPOLES`

*Description*

This command activates the P3M method to compute the dipolar interactions between charged particles. The different parameters are described in more detail in [12].

$r_{\text{cut}}$ The real space cutoff as a positive floating point number.

*mesh* The number of mesh points, as a single positive integer.

*cao* The *charge-assignment order*, an integer between 0 and 7.

*alpha* The Ewald parameter as a positive floating point number.

Make sure that you know the relevance of the P3M parameters before using P3M! If you are not sure, read the following references [24, 30, 36, 18, 19, 20, 17].

Note that dipolar P3M does not work with non-cubic boxes.

#### Tuning dipolar P3M

*TCL Syntax*

`inter magnetic` $l_B$ `p3m` ( `tune` | `tunev2` ) `accuracy` *accuracy*
[`r_cut` $r_{\text{cut}}$] [`mesh` *mesh*] [`cao` *cao*] [`alpha` $\alpha$]

Required features: `DIPOLES`

*Description*

Tuning dipolar P3M works exactly as tuning Coulomb P3M. Therefore, for details on how to tune the algorothm, refer to the documentation of Coulomb P3M (see section 5.7.1 on page 79).

For the magnetic case, the expressions of the error estimate are given in [12].

### 5.8.2. Dipolar Layer Correction (DLC)

*TCL Syntax*

`inter magnetic mdlc` *accuracy gap_size* [*far_cutoff*]

Required features: `DIPOLES`

*Description*

Like ELC but applied to the case of magnetic dipoles, but here the accuracy is the one you wish for computing the energy. $far_cutoff$ is set to a value that, assuming all dipoles to be as larger as the largest of the dipoles in the system, the error for the energy would be smaller thant the value given by accuracy. At this moment you cannot compute the accuracy for the forces, or torques, nonetheless, usually you will have an error for forces and torques smaller than for energies. Thus, the error for the energies is an upper boundary to all errors in the calculations.

At present, the program assumes that the gap without particles is along the z-direction. The gap-size is the length along the z-direction of the volume where particles are not allowed to enter.

As a reference for the DLC method, see [11].

### 5.8.3. Dipolar all-with-all and no replicas (DAWAANR)

*TCL Syntax*

`inter magnetic` $l_B$ `dawaanr`

Required features: `DIPOLES`

*Description*

This interaction calculates energies and forces between dipoles by explicitly summing over all pairs. For the directions in which the system is periodic (as defined by `setmd periodic`), it applies the minimum image convention, i.e. the interaction is effectively cut off at half a box length.

In periodic systems, this method should only be used if it is not possible to use dipolar P3M or DLC, because those methods have a far better accuracy and are much faster. In a non-periodic system, the DAWAANR-method gives the exact result.

### 5.8.4. Magnetic Dipolar Direct Sum (MDDS) on CPU

*TCL Syntax*

`inter magnetic` $l_B$ `mdds n_cut` *value_n_cut*

Required features: `DIPOLES MAGNETIC_DIPOLAR_DIRECT_SUM`

*Description*

The command enables the "magnetic dipolar direct sum". The dipole-dipole interaction is computed by explicitly summing over all pairs. If the system is periodic in one or more directions, the interactions with further *value_n_cut* replicas of the system in all periodic directions is explicitly computed.

As it is very slow, this method is not intended to do simulations, but rather to check the results you get from more efficient methods like P3M.

### 5.8.5. Dipolar direct sum on gpu

*TCL Syntax*

`inter magnetic` $l_B$ `dds-gpu`

Required features: `DIPOLES CUDA`

*Description*

This interaction calculates energies and forces between dipoles by explicitly summing over all pairs. For the directions in which the system is periodic (as defined by `setmd periodic`), it applies the minimum image convention, i.e. the interaction is effectively cut off at half a box length.

The calculations are performed on the gpu in single precision. The implementation is optimized for large systems of several thousand particles. It makes use of one thread per particle. When there are fewer particles than the number of threads the gpu can execute simultaneously, the rest of the gpu remains idle. Hence, the method will perform poorly for small systems.

### 5.8.6. Scafacos

Espresso can use the methods from the Scafacos *Scalable fast Coulomb solvers* library for dipoles, if the methods support dipolar calculations. The feature SCAFACOS_DIPOLES has to be added to myconfig.hpp to activate this feature. At the time of this writing (Apr 2016) dipolar calculations are not part of the official Scafacos development branch.

*TCL Syntax*

`scafacos_methods`

Required features: `DIPOLES SCAFACOS_DIPOLES SCAFACOS`

*Description*

This shows the methods available at the compile time of ESPResSo. That a method is listed there, does not imply that it supports dipolar calculations.

*TCL Syntax*

`inter magnetic` $l_b$ `scafacos` *method* [*parameters*] [`tolerance_field` *prec*]

Required features: `DIPOLES SCAFACOS_DIPOLES SCAFACOS`

*Description*

Here *method* is a scafacos method as returned by `scafacos_methods`. `tolerance_field` sets the desisired rms accuracy for the electric field if supported by the method. *parameters* is a list of parameters as described by the Scafacos manual. If parameters of the solver are not set, and the method supports it, the open parameteres are tuned. For details of the various methods and their parameters please refer to the Scafacos manual. Note that the `SCAFACOS` feature is only available if you build with cmake. You need to build Scafacos as a shared library. Scafacos can be used only once, either for coulomb or for dipolar interactions.

## 5.9. Special interaction commands

### 5.9.1. Tunable-slip boundary interaction

*TCL Syntax*

`inter` *type1 type2* `tunable_slip` $T$ $\gamma_L$ $r_{\text{cut}}$ $\delta t$ $v_x$ $v_y$ $v_z$

Required features: `TUNABLE_SLIP`

*Description*

Simulating microchannel flow phenomena like the Plane Poiseuille and the Plane Couette Flow require accurate boundary conditions. There are two main boundary conditions in use:

1. *slip boundary condition* which means that the flow velocity at the the hydrodynamic boundaries is zero.
2. *partial-slip boundary condition* which means that the flow velocity at the hydrodynamic boundaries does not vanish.

In recent years, experiments have indicated that the no-slip boundary condition is indeed usually not valid on the micrometer scale. Instead, it has to be replaced by the *partial-slip boundary condition*

$$\delta_B \;\; \partial_{\mathbf{n}} v_{\parallel} \|_{\mathbf{r}_B} = v_{\parallel} \|_{\mathbf{r}_B},$$

where $v_{\parallel}$ denotes the tangential component of the velocity and $\partial_{\mathbf{n}} v_{\parallel}$ its spatial derivative normal to the surface, both evaluated at the position $\mathbf{r}_B$ of the so-called *hydrodynamic boundary*. This boundary condition is characterized by two effective parameters, namely (i) the slip length $\delta_B$ and (ii) the hydrodynamic boundary $\mathbf{r}_B$.

Within the approach of the tunable-slip boundary interactions it is possible to tune the slip length systematically from full-slip to no-slip. A coordinate-dependent Langevin-equation describes a viscous layer in the vicinity of the channel walls which exerts an

additional friction on the fluid particles. $T$ is the temperature, $\gamma_L$ the friction coefficient and $r_{\text{cut}}$ is the cut-off radius of this layer. $\delta t$ is the timestep of the integration scheme. With $v_x$ $v_y$ and $v_z$ it is possible to give the layer a reference velocity to create a Plane Couette Flow. Make sure that the cutoff radius $r_{\text{cut}}$ is larger than the cutoff radius of the constraint Lennard-Jones interactions. Otherwise there is no possibility that the particles feel the viscous layer.

This method was tested for Dissipative Particle Dynamics but it is intended for mesoscopic simulation methods in general. Note, that to use tunable-slip boundary interactions you have to apply **two** wall constraints with Lennard-Jones in addition to the tunable-slip interaction. Make sure that the cutoff radius $r_{\text{cut}}$ is larger than the cutoff radius of the constraint Lennard-Jones interactions. Otherwise there is no possibility that the particles feel the viscous layer. Please read reference [55] before using this interaction.

### 5.9.2. DPD interaction

*TCL Syntax*

`inter` *type1 type2* `inter_dpd` *gamma r_cut wf tgamma tr_cut twf*

Required features: `INTER_DPD`

*Description*

This is a special interaction that is to be used in conjunction with the Dissipative Particle Dynamics algorithm 6.2.3 when the `INTER_DPD` implementation is used. The parameters correspond to the parameters of the DPD thermostat 5.9.2, but can be set individually for the different interactions.

### 5.9.3. Fixing the center of mass

*TCL Syntax*

`inter` *typeid1 typeid1* `comfixed` *flag*

Required features: `COMFIXED`

*Description*

This interaction type applies a constraint on particles of type *typeid1* such that during the integration the center of mass of these particles is fixed. This is accomplished as follows: The sum of all the forces acting on particles of type *typeid1* are calculated. These include all the forces due to other interaction types and also the thermostat. Next a force equal in magnitude, but in the opposite direction is applied to all the particles. This force is divided on the particles of type *typeid1* relative to their respective mass. Under periodic boundary conditions, this fixes the itinerant center of mass, that is, the one obtained from the unfolded coordinates.

Note that the syntax of the declaration of comfixed interaction requires the same particle type to be input twice. If different particle types are given in the input, the

program exits with an error message. *flag* can be set to 1 (which turns on the interaction) or 0 (to turn off the interaction).

Since the necessary communication is lacking at present, this interaction only works on a single node.

### 5.9.4. Pulling particles apart

*TCL Syntax*

> `inter` *typeid1 typeid2* `comforce` *flag dir force fratio*
>
> Required features: `COMFORCE`

*Description*

The comforce interaction type enables one to pull away particle groups of two different types. It is mainly designed for pulling experiments on bundles. Within a bundle of molecules of type number *typeid1* lets mark one molecule as of type *typeid2*. Using comforce one can apply a force such that t2 can be pulled away from the bundle. The $comforce_flag$ is set to 1 to turn on the interaction, and to 0 otherwise. The pulling can be done in two different directions. Either parallel to the major axis of the bundle ($dir = 0$) or perpendicular to the major axis of the bundle ($dir = 1$). *force* is used to set the magnitude of the force. *fratio* is used to set the ratio of the force applied on particles of *typeid1* vs. *typeid2*. This is useful if one has to keep the total applied force on the bundle and on the target molecule the same. A force of magnitude *force* is applied on *typeid2* particles, and a force of magnitude (*force* * *fratio*) is applied on *typeid1* particles.

### 5.9.5. Capping the force during warmup

*TCL Syntax*

> `inter forcecap` ( $F_{\max}$ | `individual` )

*Description*

Non-bonded interactions are often used to model the hard core repulsion between particles. Most of the potentials in the section are therefore singular at zero distance, and forces usually become very large for distances below the particle size. This is not a problem during the simulation, as particles will simply avoid overlapping. However, creating an initial dense random configuration without overlap is often difficult.

By artificially capping the forces, it is possible to simulate a system with overlaps. By gradually raising the cap value $F_{\max}$, possible overlaps become unfavorable, and the system equilibrates to a overlap free configuration.

This command will cap the force to $F_{max}$, *i.e.* for particle distances which would lead to larger forces than $F_{\max}$, the force remains at $F_{\max}$. Accordingly, the potential is replaced by $rF_{\max}$. Particles placed exactly on top of each other will be subject to a force of magnitude $F_{\max}$ along the first coordinate axis.

The force capping is switched off by setting $F_{\max} = 0$. Note that force capping always applies to all Lennard-Jones, tabulated, Morse and Buckingham interactions regardless of the particle types.

If instead of a force capping value, the string "individual" is given, the force capping can be set individually for each interaction. The capping radius is in this case not derived from the potential parameters, but is given by an additional signal floating point parameter to the interaction.

# 6. Setting up the system

## 6.1. `setmd`: Setting global variables.

*TCL Syntax*

(1) `setmd` *variable*
(2) `setmd` *variable* [*value*]`+`

*Description*

Variant (1) returns the value of the ESPResSo global variable *variable*, variant (2) can be used to set the variable *variable* to *value*. The '+' in variant (2) means that for some variables more than one *value* can be given (example: setmd boxl 5 5 5). The following global variables can be set:

Better throw some out (*e.g.* switches)?

Missing: lattice_switch, dpd_tgamma, n_rigid_bonds

Which commands can be used to set the *read-only* variables?

`box_l` (double[3]) Simulation box lengths of the cuboid box used by Espresso. Note that if you change the box length during the simulation, the folded particle coordinates will remain the same, i.e., the particle stay in the same image box, but at the same relative position in their image box. If you want to scale the positions, use the `change_volume` command.

`cell_grid` (int[3], *read-only*) Dimension of the inner cell grid.

`cell_size` (double[3], *read-only*) Box-length of a cell.

`dpd_gamma` (double, *read-only*) Friction constant for the DPD thermostat.

`dpd_r_cut` (double, *read-only*) Cutoff for DPD thermostat.

`dpd_ignore_fixed_particles` (int, *read-only*) Switches fixed particle DPD force calculation ON (0) or OFF (1 default).

`gamma` (double, *read-only*) Friction constant for the Langevin thermostat.

`integ_switch` (int, *read-only*) Internal switch which integrator to use.

`lb_components` (int, *read-only*) Number of fluid components.

`local_box_l` (int[3], *read-only*) Local simulation box length of the nodes.

`max_cut` (double, *read-only*) Maximal cutoff of real space interactions.

`max_cut_nonbonded` (double, *read-only*) Maximal cutoff of nonbonded real space interactions.

`max_cut_bonded` (double, *read-only*) Maximal cutoff of bonded real space interactions.

`max_num_cells` (int) Maximal number of cells for the link cell algorithm. Reasonable values are between 125 and 1000, or for some problems ($n_total_particles$ / $n_nodes$).

`max_part` (int, *read-only*) Maximal identity of a particle. *This is in general not related to the number of particles!*

`max_range` (double, *read-only*) Maximal range of real space interactions: $max_cut$ + *skin*.

`max_skin` (double, *read-only*) Maximal skin to be used for the link cell/verlet algorithm. This is the minimum of *cell_size* - *max_range*.

`min_global_cut` (double) Minimal total cutoff for real space. Effectively, this plus the skin is the minimally possible cell size. Espresso typically determines this value automatically, but some algorithms, *e.g.* virtual sites, require you to specify it manually.

`min_num_cells` (int) Minimal number of cells for the link cell algorithm. Reasonable values range in $10^{-6}N^2$ to $10^{-7}N^2$. In general just make sure that the Verlet lists are not incredibly large. By default the minimum is 0, but for the automatic P3M tuning it may be wise to set larger values for high particle numbers.

`n_layers` (int, *read-only*) Number of layers in cell structure LAYERED (see section 6.4 on page 110).

`n_nodes` (int, *read-only*) Number of nodes.

`n_part` (int, *read-only*) Total number of particles.

`n_part_types` (int, *read-only*) Number of particle types that were used so far in the `inter` command (see chaptertcl:inter).

`node_grid` (int[3]) 3D node grid for real space domain decomposition (optional, if unset an optimal set is chosen automatically).

`nptiso_gamma0` (double, *read-only*) Docs missing.

`nptiso_gammav` (double, *read-only*) Docs missing.

`npt_p_ext` (double, *read-only*) Pressure for NPT simulations.

`npt_p_inst` (double) Pressure calculated during an NPT_isotropic integration.

`npt_piston` (double, *read-only*) Mass off the box when using NPT_isotropic integrator.

`periodicity` (bool[3]) Specifies periodicity for the three directions. If the feature PARTIAL_PERIODIC is set, Espresso can be instructed to treat some dimensions

as non-periodic. Per default espresso assumes periodicity in all directions which equals setting this variable to (1,1,1). A dimension is specified as non-periodic via setting the periodicity variable for this dimension to 0. E.g. Periodicity only in z-direction is obtained by (0,0,1). Caveat: Be aware of the fact that making a dimension non-periodic does not hinder particles from leaving the box in this direction. In this case for keeping particles in the simulation box a constraint has to be set.

`skin` (double) Skin for the Verlet list.

`temperature` (double, *read-only*) Temperature of the simulation.

`thermo_switch` (double, *read-only*) Internal variable which thermostat to use.

`time` (double) The simulation time.

`time_step` (double) Time step for MD integration.

`timings` (int) Number of samples to (time-)average over.

`transfer_rate` (int, *read-only*) Transfer rate for VMD connection. You can use this to transfer any integer value to the simulation from VMD.

`verlet_flag` (bool) Indicates whether the Verlet list will be rebuild. The program decides this normally automatically based on your actions on the data.

`verlet_reuse` (double) Average number of integration steps the verlet list has been re-used.

`warnings` (int) if non-zero (default), some warnings are printed out. Set this to zero if you get annoyed by them.

`sd_viscosity` (double) viscosity of the fluid for the Stokesian Dynamics simulation.

`sd_radius` (double) hydrodynamic radius of the particles used in Stokesian Dynamics.

`sd_seed` (int[2]) seed of the Stokes Dynamics random number generator.

`sd_random_state` (int[2]) offset of the random number generator. Together with the seed, the state of the random number generator is well defined.

`sd_precision_random` (double) precision used for the approximation of the square root of the mobility. Sometimes higher accuracy can speedup the simulation.

## 6.2. `thermostat`: Setting up the thermostat

*TCL Syntax*

(1) `thermostat`
(2) `thermostat off`
(3) `thermostat` *parameters*

*Description*

The `thermostat` command is used to change settings of the thermostat.

The different available thermostats will be described in the following subsections. Note that for a simulation of the NPT ensemble, you need to use a standard thermostat for the particle velocities (*e.g.* Langevin or DPD), and a thermostat for the box geometry (*e.g.* the isotropic NPT thermostat).

You may combine different thermostats at your own risk by turning them on one by one. Note that there is only one temperature for all thermostats, although for some thermostats like the Langevin thermostat, particles can be assigned individual temperatures.

Since ESPResSo does not enforce a particular unit system, it cannot know about the current value of the Boltzmann constant. Therefore, when specifying the temperature of a thermostat, you actually do not define the temperature, but the value of the thermal energy $k_BT$ in the current unit system (see the discussion on units, Section 1.4).

Variant (1) returns the thermostat parameters. A Tcl list is given containing all the parameters needed to set the specific thermostat. (exactly the same as the input command line, without the preceding `thermostat`).

Variant (2) turns off all thermostats and sets all thermostat variables to zero. Setting temperature to zero also affects the way in which electrostatics are handled (see also Section 5.7).

Variant (3) sets up one of the thermostats described below.

Note that their are three different types of noise which can be used in ESPResSo. The one used typically in simulations is flat noise with the correct variance and it is the default used in ESPResSo, though it can be explicitly specified using the feature `FLATNOISE`. You can also employ Gaussian noise which is, in some sense, more realistic. Notably Gaussian noise (activated using the feature `GAUSSRANDOM`) does a superior job of reproducing higher order moments of the Maxwell-Boltzmann distribution. For typical generic coarse-grained polymers using FENE bonds the Gaussian noise tends to break the FENE bonds. We thus offer a third type of noise, activate using the feature `GAUSSRANDOMCUT`, which produces Gaussian random numbers but takes anything which is two standard deviations ($2\sigma$) below or above zero and set it to $-2\sigma$ or $2\sigma$ respectively. In all three cases the distribution is made such that the second moment of the distribution is the same and thus results in the same temperature.

### 6.2.1. Langevin thermostat

*TCL Syntax*

```
thermostat langevin temperature gamma_trans [gamma_rotate] [on/off]
      [on/off]
```

*Description*

**Warning:** The behavior of the Langevin Thermostat has changed in this version of ESPResSo. Before the term *gamma* was called the 'friction coefficient', but was in fact the inverse relaxation time. The code has been modified such that this value is now a proper friction coefficient.
A warning message has been added to the source (core/thermostat.cpp), which can be commented out if you have verified that your simulation results are not affected by this change.

The Langevin thermostat consists of a friction and noise term coupled via the fluctuation-dissipation theorem. The friction term is a function of the particle velocities. By specifying *gamma_trans* the diffusion coefficient for the particle becomes

$$D = \frac{temperature}{gamma_trans}. \tag{6.1}$$

The relaxation time is given by *gamma_trans*/MASS, with MASS the particle's mass. For a more detailed explanation, refer to [28]. In the near future an anisotropic diffusion coefficient tensor will be made available to simulate anisotropic colloids (rods, etc.) properly.

If the feature `ROTATION` is compiled in, the rotational degrees of freedom are also coupled to the thermostat. If only the first two arguments are specified then the diffusion coefficient for the rotation is set to the same value as that for the translation.

**Warning:** This is again a change, since originally it was set to 1/3 of the value of *gamma_trans* for some (unclear) reason. If you want to reproduce this behavior, this can be simply done by setting the value of *gamma_rotate* appropriately.

A separate rotational diffusion coefficient can be set by inputting *gamma_rotate*. This also allows one to properly match the translational and rotational diffusion coefficients of a sphere. Finally, the two options [on/off] allow one to switch the translational and rotational thermalization on or off separately, maintaining the frictional behavior. This can be useful, for instance, in high Péclet number active matter systems, where one only wants to thermalize the rotational degrees of freedom and translational motion is effected by the self-propulsion.

Using the Langevin thermostat, it is posible to set a temperature and a friction coefficient for every particle individually via the feature `LANGEVIN_PER_PARTICLE`. Consult the reference of the `part` command (chapter 4) for information on how to achieve this.

**Warning:** The behavior of the `LANGEVIN_PER_PARTICLE` thermostat has undergone a similar change.

### 6.2.2. GHMC thermostat

*TCL Syntax*

```
thermostat ghmc temperature n_md phi [-no_flip | -flip | -random_flip]
      [-no_scale | -scale]
```

*Description*

ESPResSo implements Generalized Hybrid Monte Carlo (GHMC) as a thermostat. GHMC is a simulation method for sampling the canonical ensemble [44]. The method consists of MC cycles that combine a few constant energy MD steps, specified by $n_md$, followed by a Metropolis criterion for their acceptance. Prior to integration, the particles momenta are mixed with momenta sampled from the appropriate Boltzmann distribution.

Given the particles momenta $\mathbf{p}^j$ from the last $j^{th}$ GHMC cycle the new momenta are generated by: $\mathbf{p}^{j+1} = \cos(\phi)\mathbf{p}^j + \sin(\phi)\boldsymbol{\xi}$, where $\boldsymbol{\xi}$ is a noise vector of random Gaussian variables with zero mean and variance $1/temperature$ (see [32] for more details). The momenta mixing parameter $\cos(\phi)$ corresponds to *phi* in the implementation.

In case the MD step is rejected, the particles momenta may be flipped. This is specified by setting the `-no_flip` / `-flip` option, for the `-random_flip` option half of the rejected MD steps randomly result in momenta flip. The default for momenta flip is `-no_flip`. The $\boldsymbol{\xi}$ noise vector's variance van be tuned to exactly $1/temperature$ by specifying the `-scale` option. The default for temperature scaling is `-no_scale`.

### 6.2.3. Dissipative Particle Dynamics (DPD)

ESPResSo implements Dissipative Particle Dynamics (DPD) either via a global thermostat, or via a thermostat and a special DPD interaction between particle types. The latter allows the user to specify friction coefficients on a per-interaction basis.

#### Thermostat DPD

*TCL Syntax*

```
thermostat dpd temperature gamma r_cut [ WF wf tgamma tr_cut TWF twf]
```

Required features: `DPD or TRANS_DPD`

*Description*

ESPResSo's standard DPD thermostat implements the thermostat exactly as described in [58]. We use the standard *Velocity-Verlet* integration scheme, *e.g.* DPD only influences the calculation of the forces. No special measures have been taken to self-consistently determine the velocities and the dissipative forces as it is for example described in [45]. DPD adds a velocity dependent dissipative force and a random force to the usual conservative pair forces (*e.g.* Lennard-Jones).
The dissipative force is calculated by

$$\vec{F}_{ij}^{D} = -\zeta w^{D}(r_{ij})(\hat{r}_{ij} \cdot \vec{v}_{ij})\hat{r}_{ij} \tag{6.2}$$

The random force by

$$\vec{F}_{ij}^{R} = \sigma w^{R}(r_{ij})\Theta_{ij}\hat{r}_{ij} \tag{6.3}$$

where $\Theta_{ij} \in [-0.5, 0.5[$ is a uniformly distributed random number. The connection of $\sigma$ and $\zeta$ is given by the dissipation fluctuation theorem:

$$(\sigma w^{R}(r_{ij})^2 = \zeta w^{D}(r_{ij})\mathrm{k_B}T \tag{6.4}$$

The parameters *gamma* and $r_cut$ define the strength of the friction $\zeta$ and the cutoff radius.

According to the optional parameter WF (can be set to 0 or 1, default is 0) of the thermostat command the functions $w^D$ and $w^R$ are chosen in the following way ( $r_{ij} <$ $r_cut$ ) :

$$w^{D}(r_{ij}) = (w^{R}(r_{ij}))^2 = \begin{cases} (1 - \frac{r_{ij}}{r_c})^2 & , \ wf = 0 \\ 1 & , \ wf = 1 \end{cases} \tag{6.5}$$

For $r_{ij} \geq r_cut$ $w^D$ and $w^R$ are identical to 0 in both cases.

The friction (dissipative) and noise (random) term are coupled via the fluctuation-dissipation theorem. The friction term is a function of the relative velocity of particle pairs. The DPD thermostat is better for dynamics than the Langevin thermostat, since it mimics hydrodynamics in the system.

When using a Lennard-Jones interaction, $r_cut = 2^{\frac{1}{6}}\sigma$ is a good value to choose, so that the thermostat acts on the relative velocities between nearest neighbor particles. Larger cutoffs including next nearest neighbors or even more are unphysical.

*gamma* is basically an inverse timescale on which the system thermally equilibrates. Values between 0.1 and 1 are o.k, but you propably want to try this out yourself to get a feeling for how fast temperature jumps during a simulation are. The dpd thermostat does not act on the system center of mass motion. Therefore, before using dpd, you have to stop the center of mass motion of your system, which you can achieve by using the command `galilei_transform` 6.8. This may be repeated once in a while for long runs due to round off errors (check this with the command `system_CMS_velocity`) 6.8.

Two restrictions apply for the dpd implementation of ESPResSo:

- As soon as at least one of the two interacting particles is fixed (see 4 on how to fix a particle in space) the dissipative and the stochastic force part is set to zero for both particles (you should only change this hardcoded behaviour if you are sure not to violate the dissipation fluctuation theorem).

- `DPD` does not take into account any internal rotational degrees of freedom of the particles if `ROTATION` is switched on. Up to the current version DPD only acts on the translatorial degrees of freedom.

**Transverse DPD thermostat** This is an extension of the above standard DPD thermostat [34], which dampens the degrees of freedom perpendicular on the axis between

two particles. To switch it on, the feature `TRANS_DPD` is required instead of the feature `DPD`.

The dissipative force is calculated by

$$\vec{F}_{ij}^{D} = -\zeta w^{D}(r_{ij})(I - \hat{r}_{ij} \otimes \hat{r}_{ij}) \cdot \vec{v}_{ij} \tag{6.6}$$

The random force by

$$\vec{F}_{ij}^{R} = \sigma w^{R}(r_{ij})(I - \hat{r}_{ij} \otimes \hat{r}_{ij}) \cdot \vec{\Theta}_{ij} \tag{6.7}$$

The parameters *tgamma tr_cut* define the strength of the friction and the cutoff in the same way as above. Note: This thermostat does *not* conserve angular momentum.

### Interaction DPD

*TCL Syntax*

`thermostat inter_dpd` *temperature*

Required features: `INTER_DPD`

*Description*

Another way to use DPD is by using the interaction DPD. In this case, DPD is implemented via a thermostat and corresponding interactions. The above command will set the global temperature of the system, while the friction and other parameters have to be set via the command `inter inter_dpd` (see 5.9.2 on page 97). This allows to set the friction on a per-interaction basis.

DPD interactions with fixed particles is switched off by default, because it is not clear if the results obtained with that method are physically correct. If you want activate `inter_dpd` with fixed particles please use:

*TCL Syntax*

`setmd dpd_ignore_fixed_particles 0`

Required features: `INTER_DPD`

*Description*

By default the fixed particles are ignored (`dpd_ignore_fixed_particles` is 1).

### Other DPD extensions

The features `DPD_MASS_RED` or `DPD_MASS_LIN` make the friction constant mass dependent:

$$\zeta \to \zeta M_{ij}$$

There are two implemented cases.

- `DPD_MASS_RED` uses the reduced mass: $M_{ij} = 2\frac{m_i m_j}{m_i + m_j}$
- `DPD_MASS_LIN` uses the real mass: $M_{ij} = \frac{m_i + m_j}{2}$

The prefactors are such that equal masses result in a factor 1.

### 6.2.4. Isotropic NPT thermostat

*TCL Syntax*

> `thermostat npt_isotropic` *temperature gamma0 gammaV*
> Required features: `NPT`

*Description*

This theormstat is based on the Anderson thermostat (see [1, 42]) and will thermalize the box geometry. It will only do isotropic changes of the box.

Be aware that this feature is neither properly examined for all systems nor is it maintained regularly. If you use it and notice strange behaviour, please contribute to solving the problem.

### 6.2.5. CPU thermostat

*TCL Syntax*

> `thermostat cpu` *temperature*

*Description*

The CPU thermostat consists of a number crunching and a time wasting term coupled via the central processing unit's thermal diode. For a more detailed explanation, refer to German Patent No. DE 10 2015 04F00L.

Using the CPU thermostat, it is posible to keep your central processing unit operating at a constant temperature.

Be aware that this thermostat requires *temperature* to be given in Kelvin.

### 6.2.6. Stokesian Dynamics thermostat

*TCL Syntax*

> `thermostat sd` *temperature*
> Required features: `CUDA and SD`

*Description*

This thermostat should be used together with the Stokesian Dynamics implementation. No other thermostat is able to thermalize SD correctly. The precision of the farfield contribution of the thermostat can be tuned with

## 6.3. `nemd`: Setting up non-equilibrium MD

*TCL Syntax*

(1) `nemd exchange` *n_slabs n_exchange*
(2) `nemd shearrate` *n_slabs shearrate*
(3) `nemd off`
(4) `nemd`
(5) `nemd profile`
(6) `nemd viscosity`

Required features: `NEMD`

*Description*

Use NEMD (Non Equilibrium Molecular Dynamics) to simulate a system under shear with help of an unphysical momentum change in two slabs in the system.

Variants (1) and (2) will initialise NEMD. Two distinct methods exist. Both methods divide the simulation box into *n_slab* slabs that lie parallel to the x-y-plane and apply a shear in x direction. The shear is applied in the top and the middle slabs. Note, that the methods should be used with a DPD thermostat or in an NVE ensemble. Furthermore, you should not use other special features like `part fix` or `constraints` inside the top and middle slabs. For further reference on how NEMD is implemented into ESPResSo see [57].

Variant (1) chooses the momentum exchange method. In this method, in each step the *n_exchange* largest positive x-components of the velocity in the middle slab are selected and exchanged with the *n_exchange* largest negative x-components of the velocity in the top slab.

Variant (2) chooses the shear-rate method. In this method, the targetted x-component of the mean velocity in the top and middle slabs are given by

$$target_velocity = \pm shearrate\,\frac{L_z}{4} \tag{6.8}$$

where $L_z$ is the simulation box size in z-direction. During the integration, the x-component of the mean velocities of the top and middle slabs are measured. Then, the difference between the mean x-velocities and the target x-velocities are added to the x-component of the velocities of the particles in the respective slabs.

Variant (3) will turn off NEMD, variant (4) will print usage information of the parameters of NEMD. Variant (5) will return the velocity profile of the system in x-direction (mean velocity per slab).

Variant (6) will return the viscosity of the system, that is computed via

$$\eta = \frac{F}{\dot{\gamma} L_x L_y} \tag{6.9}$$

where $F$ is the mean force (momentum transfer per unit time) acting on the slab, $L_x L_y$ is the area of the slab and $\dot{\gamma}$ is the shearrate.

NEMD as implemented generates a Pouseille flow, with shear flow rate varying over a finite wavelength determined by the box. For a planar Couette flow (constant shear, infinite wavelength), consider using Lees-Edwards boundary conditions (see 7.7 on page 121) to drive the shear.

## 6.4. `cellsystem`: Setting up the cell system

This section deals with the flexible particle data organization of ESPResSo. Due to different needs of different algorithms, ESPResSo is able to change the organization of the particles in the computer memory, according to the needs of the used algorithms. For details on the internal organization, refer to section 17.1 on page 248.

### 6.4.1. Domain decomposition

*TCL Syntax*

```
cellsystem domain_decomposition [-no_verlet_list]
```

*Description*

This selects the domain decomposition cell scheme, using Verlet lists for the calculation of the interactions. If you specify `-no_verlet_list`, only the domain decomposition is used, but not the Verlet lists.

The domain decomposition cellsystem is the default system and suits most applications with short ranged interactions. The particles are divided up spatially into small compartments, the cells, such that the cell size is larger than the maximal interaction range. In this case interactions only occur between particles in adjacent cells. Since the interaction range should be much smaller than the total system size, leaving out all interactions between non-adjacent cells can mean a tremendous speed-up. Moreover, since for constant interaction range, the number of particles in a cell depends only on the density. The number of interactions is therefore of the order N instead of order $N^2$ if one has to calculate all pair interactions.

### 6.4.2. N-squared

*TCL Syntax*

```
cellsystem nsquare
```

*Description*

This selects the very primitive nsquared cellsystem, which calculates the interactions for all particle pairs. Therefore it loops over all particles, giving an unfavorable computation time scaling of $N^2$. However, algorithms like MMM1D or the plain Coulomb interaction in the cell model require the calculation of all pair interactions.

In a multiple processor environment, the nsquared cellsystem uses a simple particle balancing scheme to have a nearly equal number of particles per CPU, *i.e.* $n$ nodes have

$m$ particles, and $p-n$ nodes have $m+1$ particles, such that $n*m+(p-n)*(m+1) = N$, the total number of particles. Therefore the computational load should be balanced fairly equal among the nodes, with one exception: This code always uses one CPU for the interaction between two different nodes. For an odd number of nodes, this is fine, because the total number of interactions to calculate is a multiple of the number of nodes, but for an even number of nodes, for each of the $p-1$ communication rounds, one processor is idle.

E.g. for 2 processors, there are 3 interactions: 0-0, 1-1, 0-1. Naturally, 0-0 and 1-1 are treated by processor 0 and 1, respectively. But the 0-1 interaction is treated by node 1 alone, so the workload for this node is twice as high. For 3 processors, the interactions are 0-0, 1-1, 2-2, 0-1, 1-2, 0-2. Of these interactions, node 0 treats 0-0 and 0-2, node 1 treats 1-1 and 0-1, and node 2 treats 2-2 and 1-2.

Therefore it is highly recommended that you use nsquared only with an odd number of nodes, if with multiple processors at all.

### 6.4.3. Layered cell system

*TCL Syntax*

| `cellsystem layered` *n_layers*

*Description*

This selects the layered cell system, which is specifically designed for the needs of the MMM2D algorithm. Basically it consists of a nsquared algorithm in x and y, but a domain decomposition along z, i. e. the system is cut into equally sized layers along the z axis. The current implementation allows for the cpus to align only along the z axis, therefore the processor grid has to have the form 1x1xN. However, each processor may be responsible for several layers, which is determined by *n_layers*, i. e. the system is split into N*$n_layers$ layers along the z axis. Since in x and y direction there are no processor boundaries, the implementation is basically just a stripped down version of the domain decomposition cellsystem.

## 6.5. CUDA

*TCL Syntax*

| (1) `cuda list`
| (2) `cuda setdevice` *id*
| (3) `cuda getdevice`

*Description*

This command can be used to choose the GPU for all subsequent GPU-computations. Note that due to driver limitations, the GPU cannot be changed anymore after the first GPU-using command has been issued, for example `lbfluid`. If you do not choose the GPU manually before that, CUDA internally chooses one, which is normally the most powerful GPU available, but load-independent.

Variant (1) lists the available devices by their ids and brand names. Variant (2) allows to choose the device by its id, which can be determined using `cuda list`, or for example the `deviceQuery` example code in the CUDA SDK. Variant (3) finally gives the id of the currently active GPU.

## 6.6. Creating bonds when particles collide

Please cite [7] (BibTeX-key `espresso2` in file `doc/ug/citations.bib`) when using dynamic bonding.

*TCL Syntax*

(1) `on_collision`
(2) `on_collision off`
(3) `on_collision` [`exception`] `bind_centers` *d bond1*
(4) `on_collision` [`exception`] `bind_at_point_of_collision` *d bond1 bond2 type*
(5) `on_collision` [`exception`] `glue_to_surface` *d bond1 bond2 type type2 type3 type4 d2*
(6) `on_collision` [`exception`] `bind_three_particles` *d bond1 bond2 res*

*Description*

With the help of the feature `COLLISION_DETECTION`, bonds between particles can be created automatically during the simulation, every time two particles collide. This is useful for simulations of chemical reactions and irreversible adhesion processes.

Two methods of binding are available:

- `bind_centers` adds a bonded interaction between the colliding particles at the first collision. This leads to the distance between the particles being fixed, the particles can, however still slide around each other.

  The parameters are as follows: *d* is the distance at which the bond is created. *bond1* denotes a pair bond and is the type of the bond created between the colliding particles. Particles that are already bound by a bond of this type do not get a new bond, in order to avoid creating multiple bonds.

- `bind_at_point_of_collision` prevents sliding of the particles at the contact. This is achieved by creating two virtual sites at the point of collision. They are rigidly connected to the colliding particles, respectively. A bond is then created between the virtual sites, or an angular bond between the two real particles and the virtual particles. In the latter case, the virtual particles are the centers of the angle potentials (particle 2 in the description of the angle potential, see 5.5). Due to the rigid connection between each of the particles in the collision and its respective virtual site, a sliding at the contact point is no longer possible. See the documentation on rigid bodies for details. In addition to the bond between the virtual sites, the bond between the colliding particles is also created. You

can either use a real bonded interaction to prevent wobbling around the point of contact or you can use a virtual bond to prevent additional force contributions, at the expense of RATTLE, see 5.3.7.

The parameters *d* and *bond1* are the same as for the `bind_centers` method. *bond2* determines the type of the bond created between the virtual sites (if applicable), and can be either a pair or a triple (angle) bond. If it is a pair bond, it connects the two virtual particles, otherwise it constraints the angle between the two real particles around the virtual ones. *type* denotes the particle type of the virtual sites created at the point of collision (if applicable). Be sure not to define a short-ranged interaction for this particle type, as two particles will be generated in the same place.

- `glue_to_surface` is used to fix a particle of type *type2* onto the surface of a particle of type *type3*. This is achieved by creating a virtual site (particle type *type*) which is rigidly connected to the particle of *type3*. A bond of type *bond2* is then created between the virtual site and the particle of *type*. Additionally, a bond of type *bond1* between the colliding particles is also created. After the collision, the particle of type *type3* is changed to type *type4*.

- `bind_three_particles` allows for the creation of agglomerates which maintain their shape similarly to those create by the `bind_at_point_of_collision` method. The present approach works without virtual sites. Instead, for each two-particle collision, the surrounding is searched for a third particle. If one is found, angular bonds are placed on each of the three particles in addition to the distance based bonds between the particle centers. The id of the angular bonds is determined from the angle between the particles. Zero degrees corresponds to bond id *bond2*, whereas 180 degrees corresponds to bond id *bond2 +res*. This method das not depend on the particles' rotational degrees of freedom being integrated. Virtual sites are also not required, and the method is implemented to run on more than one cpu core.

The code can throw an exception (background error) in case two particles collide for the first time, if the `exception` keyword is added to the invocation. In conjunction with the `catch` command of Tcl, this can be used to intercept the collision:

```
if {[catch {integrate 0} err]} {
  foreach exception [lrange $err 2 end] {
    if {[lrange $exception 0 2] == "collision between particles"} {
      set i [lindex $exception 3]
      set j [lindex $exception 5]
      puts "particles $i and $j collided"
    }
  }
}
```

The following limitations currently apply for the collision detection:

- The method is currently limited to simulations with a single cpu
- No distinction is currently made between different particle types
- The "bind at point of collision" approach requires the `VIRTUAL_SITES_RELATIVE` feature
- The "bind at point of collision" approach cannot handle collisions between virtual sites

## 6.7. Catalytic Reactions

With the help of the feature `CATALYTIC_REACTIONS`, one can define three particle types to act as reactant, catalyzer, and product. Using these reaction categories, we model the following chemical reaction system:

$$rt \rightleftharpoons pr; \tag{6.10}$$

$$rt \xrightarrow{ct} pr, \tag{6.11}$$

where the first line indicates that there is a reversible chemical reaction that converts the reactant particles ($rt$) into product ($pr$) particles, leading to an equilibrium state. The second line indicates that in the presence of a catalyst ($ct$) the forward reaction pathway is favored, i.e., conversion of reactants into products. The equilibrium reaction is described by the equilibrium constant

$$K_{\mathrm{eq}} = \frac{k_{\mathrm{eq},+}}{k_{\mathrm{eq},\text{-}}} = \frac{[pr]}{[rt]}, \tag{6.12}$$

with $[rt]$ and $[pr]$ the reactant and product concentration and $k_{\mathrm{eq},\pm}$ the forward and backward reaction rate constants, respectively. The rate constants that specify the change in concentration for the equilibrium and catalytic reaction are given by

$$\frac{d[rt]}{dt} = k_{\mathrm{eq},\text{-}}[pr] - k_{\mathrm{eq},+}[rt]; \tag{6.13}$$

$$\frac{d[pr]}{dt} = k_{\mathrm{eq},+}[rt] - k_{\mathrm{eq},\text{-}}[pr]; \tag{6.14}$$

$$-\frac{d[rt]}{dt} = \frac{d[pt]}{dt} = k_{\mathrm{ct}}[rt], \tag{6.15}$$

respectively.

In the current ESPResSo implementation we assume $k_{\mathrm{eq},+} = k_{\mathrm{eq},\text{-}} \equiv k_eq$ and therefore $K_{\mathrm{eq}} = 1$. The user can specify $k_eq \geq 0$ and $k_ct \equiv k_{\mathrm{ct}} > 0$. The former rate constant is applied to all reactant and product particles in the system, whereas the latter is applied only to the reactant particles in the vicinity of a catalyst particle. Reactant particles that have a distance of $r$ or less to at least one catalyzer particle are therefore converted

into product particles with rate constant $k_eq + k_ct$. The conversion of particles is done stochastically on the basis of the relevant rate constant ($k \geq 0$):

$$P_{\text{cvt}} = 1 - \exp\left(-k\Delta t\right), \tag{6.16}$$

with $P_{\text{cvt}}$ the probability of the conversion and $\Delta t$ the integration time step. If the equilibrium rate constant is not specified it is assumed that $k_eq = 0$.

*Python Syntax* (24)

```
espressomd.reaction.Reaction(
  product_type = ⟨int⟩,
  reactant_type = ⟨int⟩,
  catalyzer_type = ⟨int⟩,
  ct_range = ⟨float⟩,
  ct_rate = ⟨float⟩,
  eq_rate = ⟨float⟩,
  react_once = ⟨bool⟩,
  swap = ⟨bool⟩)
```

*TCL Syntax*

(0) `reaction reactant_type` *rt* `catalyzer_type` *ct* `product_type` *pt* `range` *r* `ct_rate` *k_ct* [`eq_rate` *k_eq*] [`react_once` *on/off*] [`swap` *on/off*]
(1) `reaction off`
(2) `reaction print`

Required features: `CATALYTIC_REACTIONS`[a]

[a] The current implementation also requires the use of verlet lists and domain decomposition.

*Description*

- Variant (0) defines a reaction with particles of type number *rt* as reactant, type *ct* as catalyzer and type *pt* as product[1]. The catalytic reaction rate constant is given by *k_ct*[2] and to override the default rate constant for the equilibrium reaction ($k_eq = 0$), one can specify it by *k_eq*. By default each reactant particle is checked against each catalyst particle (`react_once` *off*). However, when creating smooth surfaces using many catalyst particles, it can be desirable to let the reaction rate be independent of the surface density of these particles. That is, each particle has a likelihood of reacting in the vicinity of the surface (distance is less than *r*) as

[1] Only one type of particle can be assigned to each of these three reaction species and no particle type may be assigned to multiple species. That is, ESPResSo currently does not support particles of type 1 and 2 both to be reactants, nor can particles of type 1 be a reactant as well as a catalyst. Moreover, only one of these reactions can be implemented in a single Tcl script. If, for instance, there is a reaction involving particle types 1, 2, and 4, there cannot be a second reaction involving particles of type 5, 6, and 8. It is however possible to modify the reaction properties for a given set of types during the simulation.

[2] Currently only strictly positive values of the catalytic conversion rate constant are allowed. Setting the value to zero is equivalent to `reaction off`.

specified by the rate constant, i.e., *not* according to $P_{\mathrm{cvt}} = 1 - \exp(-nk\Delta t)$, with $n$ the number of local catalysts. To accomplish this, each reactant is considered only once each time step by using the option `react_once` *on*. The reaction command is set up such that the different properties may be influenced individually.

- Variant (1) disables the reaction. Note that at the moment, there can only be one reaction in the simulation.
- Variant (2) returns the current reaction parameters.

The Python interface has some modified capabilities with respect to the TCL interface. For example, you can alter parameters using the `.setup()` method of the reaction instance. The reaction mechanism can be inhibited and restarted using `.stop()` and `.start()`.

In future versions of ESPResSo the capabilities of the `CATALYTIC_REACTIONS` feature may be generalized to handle multiple reactant, catalyzer, and product types, as well as more general reaction schemes. Other changes may involve merging the current implementation with the `COLLISION_DETECTION` feature.

## 6.8. Galilei Transform and Particle Velocity Manipulation

The following commands may be useful in effecting the velocity of the system.

### 6.8.1. Particle motion and rotation

*Python Syntax* (25)

```
espressomd.galilei.kill_particle_motion(
  rotation = 0 or 1)
```

*TCL Syntax*

`kill_particle_motion [rotation]` [1]

Required features: [1] `ROTATION`

*Description*

This command halts all particles in the current simulation, setting their velocities to zero, as well as their angular momentum if the option `rotation` is specified and the feature ROTATION has been compiled in.

### 6.8.2. Forces and torques acting on the particles

*Python Syntax* (26)

```
espressomd.galilei.kill_particle_forces(
  torques = 0 or 1)
```

*TCL Syntax*

```
kill_particle_forces [torques] [1]
```

Required features: [1] `ROTATION`

*Description*

This command sets all forces on the particles to zero, as well as all torques if the option `torque` is specified and the feature ROTATION has been compiled in.

### 6.8.3. The centre of mass of the system

*Python Syntax* (27)

```
espressomd.galilei.system_CMS()
```

*TCL Syntax*

```
system_CMS
```

*Description*

Returns the center of mass of the whole system. It currently does not factor in the density fluctuations of the Lattice-Boltzman fluid.

### 6.8.4. The centre-of-mass velocity

*Python Syntax* (28)

```
espressomd.galilei.system_CMS_velocity()
```

*TCL Syntax*

```
system_CMS_velocity
```

*Description*

Returns the velocity of the center of mass of the whole system.

### 6.8.5. The Galilei transform

*Python Syntax* (29)

```
espressomd.galilei.galilei_transform()
```

*TCL Syntax*

```
galilei_transform
```

*Description*

Substracts the velocity of the center of mass of the whole system from every particle's velocity, thereby performing a Galilei transform into the reference frame of the center of mass of the system. This transformation is useful for example in combination with the DPD thermostat, since there, a drift in the velocity of the whole system leads to an offset in the reported temperature.

# 7. Running the simulation

## 7.1. `integrate`: Running the simulation

*TCL Syntax*

(1) `integrate` *steps* [`recalc_forces`] [`reuse_forces`]
(2) `integrate set` [`nvt`]
(3) `integrate set npt_isotropic` $p_{ext}$ *piston* [$x$ $y$ $z$] [`-cubic_box`]

*Description*

ESPResSo uses the Velocity Verlet algorithm for the integration of the equations of motion. The command `integrate` with an integer `steps` as parameter integrates the system for `steps` time steps.

Note that this implementation of the Velocity Verlet algorithm reuses forces, that is, they are computed once in the middle of the time step, but used twice, at the beginning and end. However, in the first time step after setting up, there are no forces present yet. Therefore, ESPResSo has to compute them before the first time step. That has two consequences: first, random forces are redrawn, resulting in a narrower distribution of the random forces, which we compensate by stretching. Second, coupling forces of e. g. the Lattice Boltzmann fluid cannot be computed and are therefore lacking in the first half time step. In order to minimize these effects, ESPResSo has a quite conservative heuristics to decide whether a change makes it necessary to recompute forces before the first time step. Therefore, calling hundred times `integrate 1` does the same as `integrate 100`, apart from some small calling overhead.

However, for checkpointing, there is no way for ESPResSo to tell that the forces that you read back in actually match the parameters that are set. Therefore, ESPResSo would recompute the forces before the first time step, which makes it essentially impossible to checkpoint LB simulations, where it is vital to keep the coupling forces. To work around this, `integrate` has an additional parameter [reuse_forces], which tells integrate to not recalculate the forces for the first time step, but use that the values still stored with the particles. Use this only if you are absolutely sure that the forces stored match your current setup!

The opposite problem occurs when timing interactions: In this case, one would like to recompute the forces, despite the fact that they are already correctly calculated. To this aim, the option [recalc_forces] can be used to enforce force recalculation.

Two methods for the integration can be set: For an NVT ensemble (thermostat) and for an NPT isotropic ensemble (barostat). The current method can be detected with the command `integrate set` without any parameters.

The NVT integrator is set without parameters (the temperature can be set with the thermostat). For the NPT ensemble, the parameters that can be added are:

- $p_{ext}$ The external pressure as float variable. This parameter is required.
- *piston* The mass of the applied piston as float variable. This parameter is required.
- $x$:$y$:$z$ Three integers to set the box geometry for non-cubic boxes. This parameter is optional.
- `-cubic_box` If this optional parameter is added, a cubic box is assumed.

## 7.2. `time_integration`: Runtime of the integration loop

*TCL Syntax*

(1) `time_integration`
(2) `time_integration` *steps*

*Description*

This command runs the integration as would the `integrate` command and returns the wall runtime in seconds.

## 7.3. `minimize_energy`: Run steepest descent minimization

*TCL Syntax*

(1) `minimize_energy` $f_{\max}$ *steps gamma maxdisplacement*

*Description*

*Python Syntax* (30)

```
espressomd.minimize_energy.init(
  f_max = ⟨double⟩,
  gamma = ⟨double⟩,
  max_steps = ⟨double⟩,
  max_displacement = ⟨double⟩)
espressomd.minimize_energy.minimize()
System.minimize_energy.init(
  f_max = ⟨double⟩,
  gamma = ⟨double⟩,
  max_steps = ⟨double⟩,
  max_displacement = ⟨double⟩)
System.minimize_energy.minimize()
```

In Python the minimize_energy functionality can be imported from `espressomd` as class `MinimizeEnergy`. Alternatively it is already part of the System class object and can be called from there (second variant).

This command runs a steepest descent energy minimization on the system. Please note that the behaviour is undefined if either a thermostat, Maggs electrostatics or Lattice-Boltzmann is activated. It runs a simple steepest descent algorithm:

Iterate

$$p_i = p_i + \min(gamma \times F_i, maxdisplacement),$$

while the maximal force is bigger than $f_{\max}$ or for at most *steps* times. The energy is relaxed by *gamma*, while the change per coordinate per step is limited to *maxdisplacement*. The combination of *gamma* and *maxdisplacement* can be used to get an poor man's adaptive update. Rotational degrees of freedom are treated similarly: each particle is rotated around an axis parallel to the torque acting on the particle. Please be aware of the fact that this needs not to converge to a local minimum in periodic boundary conditions. Translational and rotational coordinates that are fixed using the "fix" command or the ROTATION_PER_PARTICLE feature are not altered.

## 7.4. `tune_skin`: Tune the skin

*TCL Syntax*

(1) `tune_skin` *min max tol steps*

*Description*

Determines the fastest skin between *min* and *max* with tolerance *tol* by bisection. The integration time is determined by timing *steps* intergration. You should chose *steps* big engough so that multiple verlet updates occure even for the *max* skin, otherwise the timings are not meaningful. Please be aware that this command runs actual integrations and propagates the system. In a typical MD simulation it should be used after warmup and equilibration, in the same conditions where sampling is done.

## 7.5. `change_volume`: Changing the box volume

*TCL Syntax*

(1) `change_volume` $V_{\text{new}}$
(2) `change_volume` $L_{\text{new}}$ ( `x` | `y` | `z` | `xyz` )

*Description*

Changes the volume of either a cubic simulation box to the new volume $V_{\text{new}}$ or its given x-/y-/z-/xyz-extension to the new box-length $L_{\text{new}}$, and isotropically adjusts the particles coordinates as well. The function returns the new volume of the deformed simulation box.

## 7.6. `rotate_system`: Rotating the system around its center of mass

*TCL Syntax*

(1) `rotate_system` *phi theta alpha*

*Description*

Rotates the particle coordinates around the system's center of mass. This only makes sense for non-periodic boundaries, but no check is performed. In addition to the particle positions, the command also rotates the particles themselves, if the ROTATION feature is activated. Hence, dipole moments as well as virtual sites based on the VS_RELATIVE method will also be affected. The command only works on a single cpu.

The rotation axis is given by the parameters *phi* and *theta* as

$$\vec{a} = (\sin\theta\cos\phi; \sin\theta\sin\phi; \cos\theta), \tag{7.1}$$

and *alpha* denotes the rotation angle.

## 7.7. `lees_edwards_offset`: Applying shear between periodic images

*TCL Syntax*

(1) `lees_edwards_offset` $offset_{\text{new}}$

Required features: `LEES_EDWARDS`

*Description*

Lees-Edwards Periodic Boundary Conditions are used to impose a shear flow of speed $\dot{\gamma}$ on the system relative to its periodic images by moving the PBC wrap such that: $v_x_{unfolded} = v_x_{folded} + \dot{\gamma} y_{img}$ (where $v_x_{unfolded}$ is the $x$-component of the velocity of an image particle outside the main simulation box, and $y_{img}$ is the count of PBC boundaries crossed in the $y$-direction). The absolute value of the shear offset is set using this command; with the shear flow rate $\dot{\gamma}$ then determined internally as the difference between successive offsets. A typical usage would be to integrate by 1 MD timestep and then to increase the offset to a new value using this command; this usage pattern is intended to allow for arbitrary shear flow time profiles, such as an oscillatory shear. A common calculation to make using Lees-Edwards boundary conditions is to find the shear viscosity (or kinematic viscosity) by plotting shear stress (or shear stress/density) against the applied strain for different values of constant $\dot{\gamma}$.

Lees-Edwards differs from the NEMD approach (see 6.3 on page 109) in that the shear imposed is homogenous across the system (but only on average: symmetry breaking effects are not ruled out) rather than reversing direction with a periodicity of the box length. Accordingly the transport properties measured using Lees-Edwards are likely to

be different to (and arguably more physical than) those measured using NEMD or those from equilibrium simulations by a Green-Kubo type approach.

When the shear flow rate $\dot{\gamma}$ is non-zero, the Langevin thermostat will treat $v_x_{folded}$ as being relative to a flow field which changes smoothly from $-\dot{\gamma}/2$ at the bottom of the periodic box to $\dot{\gamma}/2$ at the top. This 'laminar' thermostatting is provided mostly because it gives quite convenient equilibration of a flowing system. In order to correctly observe transport properties, symmetry-breaking or entropy production in relation to shear flow is probably better to use the DPD thermostat (see 6.2.3 on page 105) once the initial heat-up has been carried out. The DPD thermostat removes kinetic energy from the system based on a frictional term defined relative to a local reference frame of a given particle-pair, without enforcing any specific flow pattern *a priori*. At high rates of dissipation, this can however lead to an artefactual shear-banding type effect at the periodic boundaries, such that the bulk fluid is nearly stationary. This effect is removed using the modification proposed to the DPD thermostat by Chatterjee [13] to allow treatment of systems with high dissipation rates, which is applied automatically if `LEES_EDWARDS` is compiled in. Chatterjee's modification is just to skip calculation of DPD forces (both dissipative and random) for particle pairs which cross a boundary in $y$.

The function returns the old value of the offset.

If `LEES_EDWARDS` is compiled in, then coordinates are folded into the primary simulation box as the integration progresses, to prevent a numerical overflow.

## 7.8. Stopping particles

Use the following functions, also see Section 6.8:

- `kill_particle_motion`: halts all particles in the current simulation, setting their velocities to zero, as well as their angular momentum if the feature ROTATION has been compiled in.

- `kill_particle_forces`: sets all forces on the particles to zero, as well as all torques if the feature ROTATION has been compiled in.

## 7.9. `velocities`: Setting the velocities

*TCL Syntax*

| `velocities` $v_{\max}$ [`start` *pid*] [`count` $N$]

*Description*

Sets the velocities of the particles with particle IDs between *pid* and $pid+N$ to a random vector with a length less than $v_{\max}$, and returns the absolute value of the total velocity assigned. By default, all particles are affected.

## 7.10. Fixing the particle sorting

*TCL Syntax*

```
|sort_particles
```

*Description*
Resorts the particles, making sure that

- the domain decomposition is strictly fullfilled, *i.e.* each particle is on the processor and in the cell that its position belongs to
- the particles within each cell are ordered with ascending identity.

Both conditions together form a unique particle ordering. This is important when doing checkpointing, because this makes sure that random numbers are applied in a specific order. Therefore, after writing or reading a checkpoint, you should call `sort_particles`.

## 7.11. Parallel tempering

*TCL Syntax*

```
parallel_tempering::main -rounds N -swap swap -perform perform
      [-init init] [-values {T_i}] [-connect master] [-port port]
      [-load j_node] [-resrate N_reset] [-info info]
```

*Description*
This command can be used to run a parallel tempering simulation. Since the simulation routines and the calculation of the swap probabilities are provided by the user, the method is not limited to sampling in the temperature space. However, we assume in the following that the sampled values are temperatures, and call them accordingly. It is possible to use multiple processors via TCP/IP networking, but the number of processors can be smaller than the number of temperatures.

*Arguments*

- *swap* specifies the name of the routine calculating the swap probability for a system. The routine has to accept three parameters: the *id* of the system to evaluate, and two temperatures $T_1$ and $T_2$. The routine should return a list containing the energy of the system at temperatures $T_1$ and $T_2$, respectively.
- *perform* specifies the name of the routine performing the simulation between two swap tries. The routine has to accept two parameters: the *id* of the system to propagate and the temperature $T$ at which to run it. Return values are ignored.
- *init* specifies the name of a routine initializing a system. This routine can for example create the particles, perform some intial equilibration or open output files. The routine has to accept two parameters: the *id* of the system to initialize and its initial temperature $T$. Return values are ignored.

- $R$ specifies the number of swap trial rounds; in each round, neighboring temperatures are tried for swapping alternatingly, i.e. with four temperatures, The first swap trial round tries to swap $1 \leftrightarrow 2$ and $3 \leftrightarrow 4$, the second round $2 \leftrightarrow 3$, and so on.
- *master* the name of the host on which the parallel_tempering master node is running.
- *port* the TCP/IP port on which the parallel_tempering master should listen. This defaults to 12000.
- $j_{\text{node}}$ specifies how many systems to run per ESPResSo-instance. If this is more than 1, it is the user's responsibility to manage the storage of configurations, see below for examples. This defaults to 1.
- $R_{\text{reset}}$ specifies after how many swap trial rounds to reset the counters for the acceptance rate statistics. This defaults to 10.
- *info* specifies which output the parallel tempering code should produce:

  `none` parallel tempering will be totally quiet, except for fatal errors

  `comm` information on client activities, such as connecting, is printed to stderr

  `all` print lots of information on swap energies and probabilities to stdout. This is useful for debugging and quickly checking the acceptance rates.

  This defaults to `all`.

### Introduction

The basic idea of parallel tempering is to run $N$ simulations with configurations $C_i$ in parallel at different temperatures $T_1 < T_2 < \ldots < T_N$, and exchange configurations between neighboring temperatures. This is done according to the Boltzmann rule, *i.e.* the swap probability for two configurations A and B at two different parameters $T_1$ and $T_2$ is given by

$$\min\left(1, \exp - \left[\beta(T_2)U_{\mathrm{A}}(T_2) + \beta(T_1)U_{\mathrm{B}}(T_1) - \beta(T_1)U_{\mathrm{A}}(T_1) - \beta(T_2)U_{\mathrm{B}}(T_2)\right]\right), \qquad (7.2)$$

where $U_C(T)$ denotes the potential energy of configuration $C$ at parameter $T$ and $\beta(T)$ the corresponding inverse temperature. If $T$ is the temperature, $U_C$ is indepedent of $T$, and $\beta(T) = 1/(k_B T)$. In this case, the swap probability reduces to the textbook result

$$\min(1, \exp - \left[\left(1/T_2 - 1/T_1\right)\left(U_{\mathrm{A}} - U_{\mathrm{B}}\right)/k_B\right]. \qquad (7.3)$$

However, $T$ can also be chosen to be any other parameter, for example the Bjerrum length, *i.e.* the the strength of the electrostatic interaction. In this case, $\beta(T) = \beta$ is a constant, but the energy $U_C(T)$ of a configuration $C$ depends on $T$, and one needs the full expression (7.2). ESPResSo always uses this expression.

In practice, one does not swap configurations, but temperatures, simply because exchanging temperatures requires much less communication than exchanging the properties of all particles.

Th ESPResSo implementation of parallel tempering repeatedly propagates all configurations $C_i$ and tries to swap neighboring temperatures. After the first propagation, the routine attempts to swap temperatures $T_1$ and $T_2$, $T_3$ and $T_4$, and so on. After the second propagation, swaps are attempted between temperatures $T_2$ and $T_3$, $T_4$ and $T_5$, and so on. For the propagation, parallel tempering relies on a user routine; typically, one will simply propagate the configuration by a few 100 MD time steps.

### Details on usage and an example

The parallel tempering code has to be loaded explicitely by `source "scripts/parallel_tempering.tcl"` from the Espresso directory. To make use of the parallel tempering tool, one needs to implement three methods: the propagation, the energy calculation and an initialization routine for a configuration. A typical initialization routine will look roughly like this:

```
proc init {id temp} {
  # create output files for temperature temp
  set f [open "out-$temp.dat" w]; close $f
  init_particle_positions
  thermostat langevin $temp 1.0
  equilibration_integration
  global config
  set config($id) "{[part]} [setmd time]"
}
```

The last two lines are only necessary if each instance of ESPResSo handles more than one configuration, *e.g.* if you have 300 temperatures, but only 10 ESPResSo processes (*i.e.*`-load 30`). In this case, all user provided routines need to save and restore the configurations. Saving the time is not necessary because the simulation tine across swaps is not meaningful anyways; it is however convenient for investigating the (temperature-)history of individual configurations.

A typical propagation routine accordingly looks like this

```
proc perform {id temp} {
  global config
  particle delete
  foreach p [lindex $config($id) 0] { eval part $p }
  setmd time [lindex $config($id) 1]
  thermostat langevin $temp 1.0
  set f [open "out-$temp.dat" a];
  integrate 1000
  puts $f "[setmd time] [analyze energy]"
  close $f
  set config($id) "{[part]} [setmd time]"
}
```

Again, the saving and storing of the current particle properties in the config array are only necessary if there is more than one configuration per process. In practice, one will rescale the velocities at the beginning of perform to match the current temperature, otherwise the thermostat needs a short time to equilibrate. The energies necessary to determine the swap probablility are calculated like this:

```
proc swap {id temp1 temp2} {
  global config
  particle delete
  foreach p $config($id) { eval part $p }
  set epot [expr [analyze energy total] - [analyze energy kinetic]]
  return "[expr $epot/$temp1] [expr $epot/$temp2]"
}
```

Note that only the potential energy is taken into account. The temperature enters only indirectly through the inverse temperature prefactor, see Eqn. (7.2).

The simulation is then started as follows. One of the processes runs the command

```
for {set T 0} {$T < 3} {set T [expr $T + 0.01]} {
    lappend temperatures $T }
parallel_tempering::main -load 30 -values $temperatures -rounds 1000 \
    -init init -swap swap -perform perform
```

This command turns the ESPResSo instance executing it into the master part of the parallel tempering simulation. It waits until a sufficient number of clients has connected. This are additional ESPResSo instances, which are identical to the master script, except that they execute

```
parallel_tempering::main -connect $host -load 30 \
    -init init -swap swap -perform perform
```

Here, `host` is a variable containing the TCP/IP hostname of the computer running the master process. Note that the master process waits until enough processes have connected to start the simulation. In the example, there are 300 temperatures, and each process, including the master process, will deal with 30 of them. Therefore, 1 master and 9 slave processes are required. For a typical queueing system, a starting routine could look like this:

```
master=
for h in $HOSTS; do
  if [ "$master" == "" ]; then
    ssh $h "cd run; ./pt_test.tcl"
    master=$h;
  else
    ssh $h "cd run; ./pt_test.tcl -connect $host"
  fi
done
```

where `pt_test.tcl` passes the `-connect` option on to `parallel_tempering::main`.

### Sharing data

*TCL Syntax*

`parallel_tempering::set_shareddata` *data*

*Description*
can be used at any time *by the master process* to specify additional data that is available on all processes of the parallel_tempering simulation. The data is accessible from all processes as `parallel_tempering::shareddata`.

## 7.12. Metadynamics

*TCL Syntax*

(1) `metadynamics`
(2) `metadynamics set off`
(3) `metadynamics set distance` $pid_1$ $pid_2$ $d_{\min}$ $d_{\max}$ $b_{\text{height}}$ $b_{\text{width}}$ $f_{\text{bound}}$ $d_{\text{bins}}$ *numrelaxationsteps*
(4) `metadynamics set relative_z` $pid_1$ $pid_2$ $z_{\min}$ $z_{\max}$ $b_{\text{height}}$ $b_{\text{width}}$ $f_{\text{bound}}$ $z_{\text{bins}}$ *numrelaxationsteps*
(5) `metadynamics print_stat current_coord`
(6) `metadynamics print_stat coord_values`
(7) `metadynamics print_stat profile`
(8) `metadynamics print_stat force`
(9) `metadynamics load_stat` *profile_list* *force_list*
`Required features: METADYNAMICS`

*Description*
Performs metadynamics sampling. Metadynamics is an efficient scheme to calculate the potential of mean force of a system as a function of a given reaction coordinate from a canonical simulation. The user first chooses a reaction coordinate (*e.g.* `distance`) between two particles ($pid_1$ and $pid_2$). As the system samples values along this reaction coordinate (here the distance between $pid_1$ and $pid_2$), an iterative biased force pulls the system away from the values of the reaction coordinate most sampled. Ultimately, the system is driven in such a way that it self-diffuses along the reaction coordinate between the two boundaries (here $d_{\min}$ and $d_{\max}$). The potential of mean force (or free energy profile) can be extracted by reading the `profile`.

*Arguments*

- $pid_1$ ID of the first particle involved in the metadynamics scheme.
- $pid_2$ ID of the second particle involved in the metadynamics scheme.
- $d_{\min}$ , $z_{\min}$ : minimum value of the reaction coordinate. While $d_{\min}$ must be positive (it's a distance), $z_{\min}$ can be negative since it's the relative height of $pid_1$ with respect to $pid_2$.

- $d_{\mathrm{max}}$ , $z_{\mathrm{max}}$ : maximum value of the reaction coordinate.
- $b_{\mathrm{height}}$ height of the bias function.
- $b_{\mathrm{width}}$ width of the bias function.
- $f_{\mathrm{bound}}$ strength of the ramping force at the boundaries of the reaction coordinate interval.
- $d_{\mathrm{bins}}$ , $z_{\mathrm{bins}}$ : number of bins of the reaction coordinate. This is only used for the numerical evaluation of the bias function.
- *numrelaxationsteps* number of relaxation steps before setting a new hill.
- *profile_list* Tcl list of a previous metadynamics profile.
- *force_list* Tcl list of a previous metadynamics force.

### Details on usage

Variant (1) returns the status of the metadynamics routine. Variant (2) turns metadynamics off (default value). Variant (3) sets a metadynamics scheme with the reaction coordinate `distance`, which corresponds to the distance between any two particles of the system (*e.g.* calculate the potential of mean force of the end-to-end distance of a polymer). Variant (4) sets a metadynamics scheme with the reaction coordinate `relative_z`: relative height (*i.e.* z coordinate) of particle *pid1* with respect to *pid2* (*e.g.* calculate the potential of mean force of inserting one particle *pid1* through an interface with center of mass *pid2*). Variant (5) prints the current value of the reaction coordinate. Variant (6) prints a list of the binned values of the reaction coordinate (*e.g.* $d_{\mathrm{bins}}$ values between $d_{\mathrm{min}}$ and $d_{\mathrm{max}}$). Variant (7) prints the current potential of mean force for all values of the reaction coordinate considered. Variant (8) prints the current force (norm rather than vector) for all values of the reaction coordinate considered. Variant (9) loads a previous metadynamics sampling by reading a Tcl list of the potential of mean force and applied force. This is especially useful to restart a simulation.

Note that the metadynamics scheme works seamlessly with the VIRTUAL_SITES feature, allowing to define centers of mass of groups of particles as end points of the reaction coordinate. One can therefore measure the potential of mean force of the distance between a particle and a *molecule* or *interface*.

The metadynamics scheme has (as of now) only been implemented for one processor: MPI usage is *not* supported. However, one can speed up sampling by communicating the `profile` and `force` between independent simulations (denoted *walkers*). The `print_stat` and `load_stat` can be used to input/output metadynamics information between walkers at regular intervals. Warning: the information extracted from `print_stat` contains the entire history of the simulation, while only the *last* increment of sampling should be communicated between walkers in order to avoid counting the same samples multiple times.

#### Details on implementation

As of now, only two reaction coordinates have been implemented: `distance` and `relative_z`. Many different reaction coordinates can be set up, and it is rather easy to implement new ones. See the code in `metadynamics.{h,c}` for further details.

The bias functions that are applied to the potential of mean force and the biased force are not gaussian function (as in many metadynamics codes) but so-called Lucy functions. See [43] for more details. These avoid the calculation of exponentials.

## 7.13. `integrate_sd`: Running a stokesian dynamics simulation

*TCL Syntax*

(1) `integrate_sd` *steps*

*Description*

ESPResSo uses in the stokesian dynamics algorithm the euler integrater for the equations of motion. The motion are overdamped. The velocities of the particles are related to the displacement in the last timestep. This are, at least if the system is thermalized, however not usefull, as the displacements don't scale linear with the timestep. The command `integrate_sd` with an integer *steps* as parameter integrates the system for *steps* time steps. This is implemented using CUDA, so CUDA has to be available in the system.

Currently there is no parallel implementation of this integrator, so all particles have to be in a single process.

### 7.13.1. Setting up the system

Before running a stokesian dynamics simulation, you have to set a view constants:

`sd_radius` The hydrodynamic particle radius of the (sperical) particles. Only one particle size is supported.

`sd_viscosity` The viscisity of the fluid. Remember this is only a scaling of the timestep, so you can set it without problems to one.

`sd_seed` (int[2]) seed of the Stokes Dynamics random number generator. As the generator used for the SD runes on the GPU (cuRAND) it has its own seed and own state.

`sd_random_state` (int[2]) offset of the random number generator. Together with the seed, the state of the random number generator is well defined.

`sd_precision_random` (double) precision used for the approximation of the square root of the mobility. Sometimes higher accuracy can speedup the simulation.

Make sure to only use the stokesian dynamics thermostat. If you made a warmup integration without SD, set it with:

```
thermostat off
thermostat sd $temp
```

### 7.13.2. Periodicity

The Code uses the ewald-summation (as derived in [9]) of the Rotne-Prager tensor to be usefull in periodic boundary conditions. To disable the peridoicity of the hydrodynamic, use the feature `SD_NOT_PERIODIC`.

The ewald summation is required if the system is periodic, as otherwise an cutoff is introduced, which can lead to negative eigenvalues of the mobility. This leads to a break down of the code, as it is not possible to thermalize such a system.

### 7.13.3. Floatingpoint precision

It is possible to switch between double and float as datatype in Stokesian Dynamics. As GPUs have more single than double precision floating point units, the single precision is faster. However this can lead to problems, if the eigenvalues are not precise enough and get negative. Therfore the default is double precision. If you want to use single precision, use the feature `SD_USE_FLOAT`.

### 7.13.4. Farfield only

The mobility matrix consists of two contribution, the farfield and the nearfield lubrication correction. If the nearfield is neglegible, than the feature `SD_FF_ONLY` can be used. This should speedup the simulation, as the nearfield doesn't need to be inverted. Additional the thermal displacements can be directly calculated.

## 7.14. Multi-timestepping

*TCL Syntax*

```
setmd smaller_time_step 0.001
      part i smaller_timestep 1
Required features:  MULTI_TIMESTEP
```

*Description*

The multi-timestepping integrator allows to run two concurrent integration time steps within a simulation, associating beads with either the large *time_step* or the other *smaller_time_step*. Setting *smaller_time_step* to a positive value turns on the multi-timestepping algorithm. The ratio *time_step*/*smaller_time_step must* be an integer. Beads are by default associated with *time_step*, corresponding to the particle property *smaller_timestep* 0. Setting *smaller_timestep* to 1 associate the particle to the *smaller_time_step* integration. The integrator can be used in the NVE ensemble, as well as with the Langevin thermostat and the modified Andersen barostat for NVT and NPT simulations, respectively. See [10] for more details.

# 8. Analysis in Tcl

Add a comment to the UG that the analysis routines in ESPResSo with Python currently require the user to 'from espressomd import analyze.'

ESPResSo has two fundamentally different classes of observables for analyzing the systems. On the one hand, some observables are computed from the Tcl level. In that case, the observable is measured in the moment that the corresponding Tcl function is called, and the results are returned to the Tcl script. In general, observables in this class should only be computed after a large number of timesteps, as switching forth and back between the C- and the Tcl-level is costly. This chapter describes all observables in this class.

On the other hand, some observables are computed and stored in the C-core of ESPResSo during a call to the function `integrate`, while they are set up and their results are collected from the Tcl level. These observables are more complex to implement and offer less flexibility, while the are significantly faster and more memory efficient, and they can be set up to be computed every few timesteps. The observables in this class are described in chapter 9.

The class of Tcl-level analysis functions is mainly controlled via the `analyze` command. It has two main uses: Calculation of observables (`analyze` *observable*) and definition and analysis of topologies in the system (`analyze` *topologycommand*). In addition, ESPResSo offers the command `uwerr` (see section 8.4 for computing statistical errors in time series.

## 8.1. Available observables

The command `analyze` provides online-calculation of local and global observables.

### 8.1.1. Minimal distances between particles

*TCL Syntax*

(1) `analyze mindist` [*type_list_a type_list_b*]
(2) `analyze distto` *pid*
(3) `analyze distto` *x y z*

*Description*

Variant (1) returns the minimal distance between two particles in the system. If the type-lists are given, then the minimal distance between particles of only those types is determined.

`distto` returns the minimal distance of all particles to particle *pid* (variant (2)), or to the coordinates ($x$, $y$, $z$) (Variant (3)).

### 8.1.2. Particles in the neighbourhood

*TCL Syntax*

(1) `analyze nbhood` *pid r_catch*
(2) `analyze nbhood` *x y z* $r_c atch$

*Description*

Returns a Tcl-list of the particle ids of all particles within a given radius *r_catch* around the position of the particle with number *pid* in variant (1) or around the spatial coordinate ($x$, $y$, $z$) in variant (2).

### 8.1.3. Particle distribution

*Python Syntax* (31)

```
espressomd.analyze.Analysis(system).distribution(
  system = ⟨system⟩,
  type_list_a = ⟨list of ints⟩,
  type_list_b = ⟨list of ints⟩,
  r_min = ⟨float⟩,
  r_max = ⟨float⟩,
  r_bins = ⟨int⟩,
  log_flag = ⟨int⟩,
  int_flag = ⟨int⟩)
```

*TCL Syntax*

`analyze distribution` *part_type_list_a part_type_list_b*
[*rmin* [*rmax* [*rbins* [*log_flag* [*int_flag*]]]]]

*Description*

Returns its parameters and the distance distribution of particles (probability of finding a particle of type $a$ at a certain distance around a particle of type $b$, disregarding the fact that a spherical shell of a larger radius covers a larger volume) with types specified in *part_type_list_a* around particles with types specified in *part_type_list_b* with distances between *rmin* and *rmax*, binned into *rbins* bins. The bins are either equidistant (if $log_flag = 0$) or logarithmically equidistant (if $log_flag \geq 1$). If an integrated distribution is required, use $int_flag = 1$. The distance is defined as the *minimal* distance between a particle of one group to any of the other group.

*Output format*

The output corresponds to the blockfile format (see section 10.5 on page 173):

{ *parameters* }
{
  { $r$ $dist(r)$ }
  ⋮
}

### 8.1.4. Radial density map

*TCL Syntax*

```
analyze radial_density_map xbins ybins xrange yrange
      [axisofrotation centerofrotation beadtypelist [thetabins]]
```

*Description*

Returns the radial density of particles around a given axis. Parameters are:

- *xbins* histogram bins in x direction.
- *ybins* histogram bins in y direction.
- *xrange* range for analysis in x direction.
- *yrange* range for analysis in y direction.
- *axisofrotation* rotate around given axis. (x, y, or z)
- *centerofrotation* rotate around given point.
- *beadtypelist* only analyze beads of given types.
- *thetabins* histogram bins in angle theta.

*Caveat*

This command does not do what you might expect. Here is an overview of the currently identified properties.

1. *xbins* is the number of bins along the axis of rotation.
2. *ybins* is the number of bins in the radial direction.
3. The centre point (*centerofrotation*) of the cylinder is located in the lower cap, i.e., *xrange* is the height of the cylinder with respect to this centre point.
4. The bins are distributed along *axisofrotation* starting from 0 (*centerofrotation*).
5. The *thetabins* seem to average with respect to the centre of mass of the particles in the individual bins rather than with respect to the central axis, which one would think is natural.

### 8.1.5. Cylindrical Average

*Python Syntax* (32)

```
espressomd.analyze.Analysis(system).cylindrical_average(
  system = ⟨system⟩,
  center = ⟨3 floats⟩,
  direction = ⟨3 floats⟩,
  length = ⟨float⟩,
  radius = ⟨float⟩,
  bins_axial = ⟨int⟩,
  bins_radial = ⟨int⟩,
  types = ⟨list of ints⟩)
```

*TCL Syntax*

```
analyze cylindrical_average center direction length radius bins_axial
        bins_radial [types]
```

*Description*

The command returns a list of lists. The outer list contains all data combined whereas each inner list contains one line. Each lines stores a different combination of the radial and axial index. The output might look something like this

```
{ { 0 0 0.05 -0.25 0.0314159 0 0 0 0 0 0 }
  { 0 1 0.05 0.25 0.0314159 31.831 1.41421 1 0 0 0 }
  ... }
```

In this case two different particle types were present. The columns of the respective lines are coded like this

| index_radial | index_axial | pos_radial | pos_axial | binvolume | density | v_radial | v_axial | density | v_radial | v_axial |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0.05 | -0.25 | 0.0314159 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0.05 | 0.25 | 0.0314159 | 31.831 | 1.41421 | 1 | 0 | 0 | 0 |

As one can see the columns `density`, `v_radial`, and `v_axial` appear twice. The order of appearance corresponds two the order of the types in the argument *types*. For example if *types* was set to `{0 1}` then the first triple is associated to type 0 and the second triple to type 1.

After knowing what the output looks like we might want to have more information on how to input data.

- *center* is a double list containing the coordinates of the centre point of the cylinder.
- *direction* is a double list containing a (not necessarily normalised) vector.
- *length* is the total length of the cylinder.
- *radius* is the radius of the cylinder.
- *bins_axial* is the number of bins along the *direction* vector.

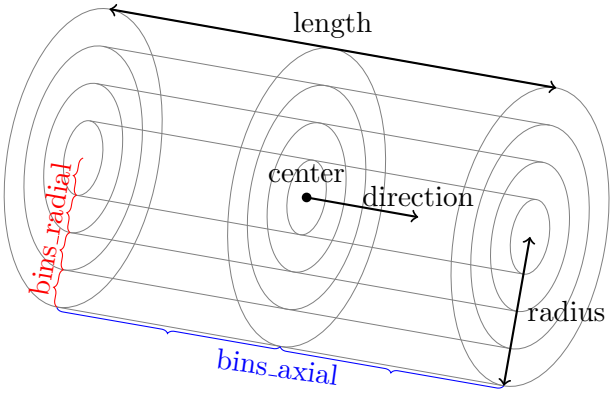



Figure 8.1.: Illustration of the parameters for the `analyze cylindrical_average` command.

- *bins_radial* is the number of bins in radial direction.
- *types* is an int list of the type IDs.

Because all of this text is super abstract we additionally drew a picture of what these variables actually mean, see figure 8.1.

### 8.1.6. Modes

*TCL Syntax*

```
analyze modes2d
```

*Description*

Analyzes the modes of a configuration. Requires that a grid is set and that the system contains more than two particles. Output are four numbers in the order:

$$ht_{RE} \quad ht_{IM} \quad \theta_{RE} \quad \theta_{IM}$$

### 8.1.7. Lipid orientation

*TCL Syntax*

```
(1) analyze get_lipid_orients
(2) analyze lipid_orient_order
```

*Description*

Document the usage!

### 8.1.8. Bilayers

*TCL Syntax*

```
(1) analyze bilayer_set
(2) analyze bilayer_density_profile
```

*Description*

Document the usage!

### 8.1.9. GPB

*TCL Syntax*

```
analyze cell_gpb Manningparameter outercellradius innercellradius
      [accuracy [numberofinteractions]]
```

*Description*

Document the usage and what it is!

### 8.1.10. Get folded positions

*TCL Syntax*

```
analyze get_folded_positions [-molecule] [shift x y z]
```

*Description*

Outputs the folded positions of particles. Without any parameters, the positions of all particles are given, folded to the box length. The optional parameter `-molecule` ensures that molecules (particle groups) are kept intact. The optional shift parameters can be used to shift the not separated molecules if needed.

### 8.1.11. Vkappa

*Python Syntax* (33)

```
espressomd.analyze.Analysis(system).Vkappa(
  system = ⟨system⟩,
  mode = ⟨string⟩,
  Vk1 = ⟨float⟩,
  Vk2 = ⟨float⟩,
  avk = ⟨float⟩)
```

*TCL Syntax*

```
analyze Vkappa [(  reset | read | set V_κ,1  V_κ,2  avk  ) ]
```

*Description*

Calculates the compressibility $V \times \kappa_T$ through the Volume fluctuations $V \times \kappa_T = \beta\left(\langle V^2\rangle - \langle V\rangle^2\right)$ [37]. Given no arguments this function calculates and returns the current value of the running average for the volume fluctuations. The argument `reset`

clears the currently stored values. With `read` the cumulative mean volume, cumulative mean squared volume and how many samples were used can be retrieved. Likewise the option `set` enables you to set those.

### 8.1.12. Radial distribution function

*Python Syntax* (34)

```
espressomd.analyze.Analysis(system).rdf(
  system = ⟨system⟩,
  rdf_type = ⟨string⟩,
  type_list_a = ⟨list of ints⟩,
  type_list_b = ⟨list of ints⟩,
  r_min = ⟨float (0.0)⟩,
  r_max = ⟨float⟩,
  r_bins = ⟨int (100)⟩ n_conf = ⟨int⟩)
```

*TCL Syntax*

`analyze ( rdf | <rdf> )` *part_type_list_a part_type_list_b* [*rmin rmax rbins*]

*Description*

Returns its parameters and the radial distribution function (rdf) of particles with types specified in *part_type_list_a* around particles with types specified in *part_type_list_b*. The range is given by *rmin* and *rmax* and is divided into *rbins* equidistant bins.

*Output format*

The output corresponds to the blockfile format (see section 10.5 on page 173):

{ *parameters* }
{
  { *r rdf*(*r*) }
  ⋮
}

### 8.1.13. Structure factor

*Python Syntax* (35)

```
espressomd.analyze.Analysis(system).structure_factor(
  system = ⟨system⟩,
  sf_types = ⟨list of ints⟩,
  sf_order = ⟨int⟩)
```

*TCL Syntax*

`analyze structurefactor` *types order*

*Description*

Returns the spherically averaged structure factor $S(q)$ of particles specified in *types*. $S(q)$ is calculated for all possible wave vectors, $\frac{2\pi}{L} <= q <= \frac{2\pi}{L} order$. Do not choose parameter *order* too large, because the number of calculations grows as $order^3$.

*Output format*

The output corresponds to the blockfile format (see section 10.5 on page 173):

$\{\ q_value\ \ S(q)_value\ \}$
$\vdots$

### 8.1.14. Van-Hove autocorrelation function $G(r,t)$

*TCL Syntax*

`analyze vanhove` *type rmin rmax rbins* [*tmax*]

*Description*

Returns the van Hove auto correlation function $G(r,t)$ and the mean square displacement $msd(t)$ for particles of type *ptype* for the configurations stored in the array configs. This tool assumes that the configurations stored with `analyze append` (see section 8.3 on page 150) are stored at equidistant time intervals. $G(r,t)$ is calculated for each multiple of this time intervals. For each time t the distribution of particle displacements is calculated according to the specification given by *rmin*, *rmax* and *rbins*. Optional argument *tmax* defines the maximum value of $t$ for which $G(r,t)$ is calculated. If it is omitted or set to zero, maximum possible value is used. If the particles perform a random walk (*i.e.* a normal diffusion process) $G(r,t)/r^2$ is a Gaussian distribution for all times. Deviations of this behavior hint on another diffusion process or on the fact that your system has not reached the diffusive regime. In this case it is also very questionable to calculate a diffusion constant from the mean square displacement via the Stokes-Einstein relation.

*Output format*

The output corresponds to the blockfile format (see section 10.5 on page 173):

$\{$ `msd` $\{\ msd(0)\ \ msd(1)\ \ \ldots\ \}\ \}$
$\{$ `vanhove` $\{\ \{\ G(0,0)\ \ G(1,0)\ \ \ldots\ \}$
$\{\ G(0,1)\ \ G(1,1)\ \ \ldots\ \}$
$\vdots$
$\}$
$\}$

The $G(r,t)$ are normalized such that the integral over space always yields 1.

### 8.1.15. Center of mass

*TCL Syntax*

```
analyze centermass part_type
```

*Description*

Returns the center of mass of particles of the given type.

### 8.1.16. Moment of inertia matrix

*Python Syntax* (36)

```
espressomd.analyze.Analysis(system).momentofinertiamatrix(
  system = ⟨system⟩,
  p_type = ⟨int⟩)
```

*TCL Syntax*

```
(1) analyze momentofinertiamatrix  typeid
(2) analyze find_principal_axis typeid
```

*Description*

Variant (1) returns the moment of inertia matrix for particles of given type *typeid*. The output is a list of all the elements of the 3x3 matrix. Variant (2) returns the eigenvalues and eigenvectors of the matrix.

### 8.1.17. Gyration tensor

*Python Syntax* (37)

```
espressomd.analyze.Analysis(system).gyration_tensor(
  system = ⟨system⟩,
  p_type = ⟨int⟩)
```

*TCL Syntax*

```
analyze gyration_tensor [typeid]
```

*Description*

Analyze the gyration tensor of particles of a given type *typeid*, or of all particles in the system if no type is given. Returns a Tcl-list containing the squared radius of gyration, three shape descriptors (asphericity, acylindricity, and relative shape anisotropy), eigenvalues of the gyration tensor and their corresponding eigenvectors. The eigenvalues are sorted in descending order.

### 8.1.18. Aggregation

*TCL Syntax*

```
analyze aggregation dist_criteria s_mol_id f_mol_id
      [min_contact [charge_criteria]]
```

*Description*
Returns the aggregate size distribution for the molecules in the molecule id range *s_mol_id* to *f_mol_id*. If any monomers in two different molecules are closer than *dist_criteria* they are considered to be in the same aggregate. One can use the optional *min_contact* parameter to specify a minimum number of contacts such that only molecules having at least *min_contact* contacts will be considered to be in the same aggregate. The second optional parameter *charge_criteria* enables one to consider aggregation state of only oppositely charged particles.

### 8.1.19. Identifying pearl-necklace structures

*TCL Syntax*

`analyze necklace` *pearl_threshold back_dist space_dist first length*

*Description*
Algorithm for identifying pearl necklace structures for polyelectrolytes in poor solvent [39]. The first three parameters are tuning parameters for the algorithm: *pearl_threshold* is the minimal number of monomers in a pearl. *back_dist* is the number of monomers along the chain backbone which are excluded from the space distance criterion to form clusters. *space_dist* is the distance between two monomers up to which they are considered to belong to the same clusters. The three parameters may be connected by scaling arguments. Make sure that your results are only weakly dependent on the exact choice of your parameters. For the algorithm the coordinates stored in partCfg are used. The chain itself is defined by the identity first of its first monomer and the chain length length. Attention: This function is very specific to the problem and might not give useful results for other cases with similar structures.

### 8.1.20. Finding holes

*TCL Syntax*

`analyze holes` $typeid_{\text{probe}}$ *mesh_size*

Required features: `LENNARD_JONES`

*Description*
Function for the calculation of the unoccupied volume (often also called free volume) in a system. Details can be found in Schmitz and Muller-Plathe [52]. It identifies free space in the simulation box via a mesh based cluster algorithm. Free space is defined via a probe particle and its interactions with other particles which have to be defined through LJ interactions with the other existing particle types via the inter command before calling this routine. A point of the mesh is counted as free space if the distance of the point is larger than LJ_cut+LJ_offset to any particle as defined by the LJ interaction parameters between the probe particle type and other particle types. How to use this function: Define interactions between all (or the ones you are interested in) particle types in your system and a fictitious particle type. Practically one uses the van der

Waals radius of the particles plus the size of the probe you want to use as the Lennard Jones cutoff. The mesh spacing is the box length divided by the $mesh_size$.

*Output format*

{ *n_holes mean_hole_size max_hole_size free_volume_fraction*
  { *sizes* }
  { *surfaces* }
  { *element_lists* }
}

A hole is defined as a continuous cluster of mesh elements that belong to the unoccupied volume. Since the function is quite rudimentary it gives back the whole information suitable for further processing on the script level. *sizes* and *surfaces* are given in number of mesh points, which means you have to calculate the actual size via the corresponding volume or surface elements yourself. The complete information is given in the element_lists for each hole. The element numbers give the position of a mesh point in the linear representation of the 3D grid (coordinates are in the order x, y, z). Attention: the algorithm assumes a cubic box. Surface results have not been tested. .

I think there is still a bug in there (Hanjo)

### 8.1.21. Temperature of the LB fluid

*TCL Syntax*

```
analyze fluid temp [1] or 2 or 3
```

Required features: [1] `LB` [2] `LB_GPU` [3] `ELECTROKINETICS`

*Description*

This command returns the temperature of the lattice-Boltzmann (LB) fluid, see Chapter 12, by averaging over the fluid nodes. In case `LB_BOUNDARIES` or `LB_BOUNDARIES_GPU` are compiled in and boundaries are defined, only the available fluid volume is taken into account.

### 8.1.22. Momentum of the System

*Python Syntax* (38)

```
espressomd.analyze.Analysis(system).calc_re(
  system = ⟨system⟩,
  include_particles = ⟨bool⟩,
  include_lbfluid = ⟨bool⟩)
```

*TCL Syntax*

```
analyze momentum [( particles | lbfluid )]
```

*Description*

This command returns the total linear momentum of the particles and the lattice-Boltzmann (LB) fluid, if one exists. Giving the optional parameters either causes the

command to ignore the contribution of LB or of the particles.

### 8.1.23. Energies

*TCL Syntax*

(1) `analyze energy`
(2) `analyze energy ( total | kinetic | coulomb | magnetic )`
(3) `analyze energy bonded` *bondid*
(4) `analyze energy nonbonded` *typeid1* *typeid2*

*Description*

Describe the different energies components returned by the different commands!

Returns the energies of the system. Variant (1) returns all the contributions to the total energy. Variant (2) returns the numerical value of the total energy or its kinetic or Coulomb or magnetic contributions only. Variants (3) and (4) return the energy contributions of the bonded resp. non-bonded interactions.

*Output format (variant* (1)*)*

{ `energy` *value* } { `kinetic` *value* } { `interaction` *value* } ...

### 8.1.24. Pressure

*TCL Syntax*

(1) `analyze pressure`
(2) `analyze pressure total`
(3) `analyze pressure ( totals | ideal | coulomb | tot_nonbonded_inter | tot_nonbonded_intra | vs_relative )`
(4) `analyze pressure bonded` *bondid*
(5) `analyze pressure nonbonded` *typeid1* *typeid2*
(6) `analyze pressure nonbonded_intra` [*typeid*]
(7) `analyze pressure nonbonded_inter` [*typeid*]

*Description*

Computes the pressure and its contributions in the system. Variant (1) returns all the contributions to the total pressure. Variant (2) will return the total pressure only. Variants (3), (4) and (5) return the corresponding contributions to the total pressure.

**Warning:** Pressure works only with certain interactions and features. Read in detail before use!

Document arguments nb_inter, nb_intra, tot_nb_inter and tot_nb_intra

The pressure is calculated (if there are no electrostatic interactions) by

$$p = \frac{2E_{kinetic}}{Vf} + \frac{\sum_{j>i} F_{ij} r_{ij}}{3V} \tag{8.1}$$

where $f = 3$ is the number of translational degrees of freedom of each particle, $V$ is the volume of the system, $E_{kinetic}$ is the kinetic energy, $F_{ij}$ the force between particles i and

j, and $r_{ij}$ is the distance between them. The kinetic energy divided by the degrees of freedom is

$$\frac{2E_{kinetic}}{f} = \frac{1}{3}\sum_i m_i v_i^2. \tag{8.2}$$

Note that Equation 8.1 can only be applied to pair potentials and central forces. Description of how contributions from other interactions are calculated is beyond the scope of this manual. Three body potentials are implemented following the procedure in Ref. [61]. A different formula is used to calculate contribution from electrostatic interactions in P3M. For electrostatic interactions, the $k$-space contribution is not well tested, so use with caution! Anything outside that is currently not implemented. Four-body dihedral potentials are not included. In case of rigid body rotation, virial contribution from torques is not included. The pressure contribution for rigid bodies constructed by means of the VIRTUAL_SITES_RELATIVE mechanism is included. On the other hand, the pressure contribution for rigid bonds is not included. All other constraints of any kind are not currently accounted for in the pressure calculations. The pressure is no longer correct, e.g., when particles are confined to a plane.

Description of how electrostatic contribution to Pressure is calculated

The command is implemented in parallel.

*Output format (variant* (1)*)*

```
{ { pressure total_pressure }
   { ideal ideal_gas_pressure }
   { { bond_type pressure }
      ⋮
   }
   { { nonbonded_type pressure }
      ⋮
   }
   { coulomb pressure }
}
```

specifying the pressure, the ideal gas pressure, the contributions from bonded interactions, the contributions from non-bonded interactions and the electrostatic contributions.

### 8.1.25. Stress Tensor

*TCL Syntax*

```
(1) analyze stress_tensor
(2) analyze stress_tensor total
(3) analyze stress_tensor ( totals | ideal | coulomb |
        tot_nonbonded_inter | tot_nonbonded_intra )
(4) analyze stress_tensor bonded bond_type
(5) analyze stress_tensor nonbonded typeid1 typeid2
(6) analyze stress_tensor nonbonded_intra [typeid]
(7) analyze stress_tensor nonbonded_inter [typeid]
```

*Description*
Computes the stress tensor of the system. The various options are equivalent to those described by `analyze pressure` in 8.1.24 on page 142. It is called a stress tensor but the sign convention follows that of a pressure tensor.

**Warning:** Stress tensor works only with certain interactions and features. Same restrictions as in the case of Pressure are applicable (see section 8.1.24).

The stress tensor is calculated by

$$p^{(kl)} = \frac{\sum_i m_i v_i^{(k)} v_i^{(l)}}{V} + \frac{\sum_{j>i} F_{ij}^{(k)} r_{ij}^{(l)}}{V} \tag{8.3}$$

where the notation is the same as for `analyze pressure` in 8.1.24 on page 142 and the superscripts $k$ and $l$ correspond to the components in the tensors and vectors.

Note that the angular velocities of the particles are not included in the calculation of the stress tensor.

The command is implemented in parallel.

*Output format (variant* (1)*)*

{ { `pressure` *total_pressure_tensor* }
  { `ideal` *ideal_gas_pressure_tensor* }
  { { *bond_type* *pressure_tensor* }
    ⋮
  }
  { { *nonbonded_type* *pressure_tensor* }
    ⋮
  }
  { `coulomb` *pressure_tensor* }
}

specifying the pressure tensor, the ideal gas pressure tensor, the contributions from bonded interactions, the contributions from non-bonded interactions and the electrostatic contributions.

## 8.1.26. Local Stress Tensor

*TCL Syntax*

`analyze local_stress_tensor` *periodic_x* *periodic_y* *periodic_z* *range_start_x* *range_start_y* *range_start_z* *range_x* *range_y* *range_z* *bins_x* *bins_y* *bins_z*

*Description*
Computes local stress tensors in the system. A cuboid is defined starting at the coordinate (*range_start_x*,*range_start_y*,*range_start_z*) and going to the coordinate (*range_start_x*+*range_x*,

*range_start_y*+*range_y*, *range_start_z*+*range_z*). This cuboid in divided into *bins_x* bins in the x direction, *bins_y* bins in the y direction and *bins_z* bins in the z direction such that the total number of bins is *bins_x***bins_y***bins_z*. For each of these bins a stress tensor is calculated using the Irving Kirkwood method. That is, a given interaction contributes towards the stress tensor in a bin proportional to the fraction of the line connecting the two particles that is within the bin.

If the P3M and MMM1D electrostatic methods are used, these interactions are not included in the local stress tensor. The DH and RF methods, in contrast, are included. Concerning bonded interactions only two body interactions (FENE, Harmonic) are included (angular and dihedral are not). For all electrostatic interactions only the real space part is included.

Care should be taken when using constraints of any kind, since these are not accounted for in the local stress tensor calculations.

The command is implemented in parallel.

*Output format (variant* (1)*)*

{ { `LocalStressTensor` }
  { { *x_bin* *y_bin* *z_bin* } { *pressure_tensor* } }
    ⋮
}

specifying the local pressure tensor in each bin.

## 8.2. Analyzing groups of particles (molecules)

*TCL Syntax*

(1) `analyze set chains` [*chain_start* *n_chains* *chain_length*]
(2) `analyze set topo_part_sync`
(3) `analyze set`

*Description*

The above set of functions is designed to facilitate analysis of molecules. Molecules are expected to be a group of particles comprising a contiguous range of particle IDs. Each molecule is a set of consecutively numbered particles and all molecules are supposed to consist of the same number of particles. Some functions in this group require that the particles constituting a molecule are connected into linear chains (particle $n$ is connected to $n+1$ and so on) while others are applicable to molecules of whatever topology.

The `analyze set` command defines the structure of the current system to be used with some of the analysis functions.

Variant (1) defines a set of *n_chains* chains of equal length *chain_length* which start with the particle with particle number *chain_start* and are consecutively numbered (*i.e.* the last particle in that topology has number $chain_start + n_chains * chain_length - 1$).

Variant (2) synchronizes topology and particle data, assigning *mol_id* values to particles.

Variant (3) will return the chains currently stored.

### 8.2.1. Chains

All analysis functions in this section require the topology of the chains to be set correctly. The topology can be provided upon calling. This (re-)sets the structure info permanently, *i.e.* it is only required once.

#### End-to-end distance

*Python Syntax* (39)

```
espressomd.analyze.Analysis(system).calc_re(
  system = ⟨system⟩,
  chain_start = ⟨int⟩,
  n_chains = ⟨int⟩,
  chain_length = ⟨int⟩)
```

*TCL Syntax*

```
analyze ( re | <re> ) [chain_start n_chains chain_length]
```

*Description*
Returns the quadratic end-to-end-distance and its root averaged over all chains. If `<re>` is used, the distance is averaged over all stored configurations (see section 8.3 on page 150).

*Output format*
{ *re error_of_re re2 error_of_re2* }

#### Radius of gyration

*Python Syntax* (40)

```
espressomd.analyze.Analysis(system).calc_rg(
  system = ⟨system⟩,
  chain_start = ⟨int⟩,
  n_chains = ⟨int⟩,
  chain_length = ⟨int⟩)
```

*TCL Syntax*

```
analyze ( rg | <rg> ) [chain_start n_chains chain_length]
```

*Description*
Returns the radius of gyration averaged over all chains. It is a radius of a sphere, which would have the same moment of inertia as the molecule, defined as

$$R_{\mathrm{G}}^2 = \frac{1}{N}\sum_{i=1}^{N}(\vec{r}_i - \vec{r}_{\mathrm{cm}})^2 \ , \tag{8.4}$$

where $\vec{r}_i$ are position vectors of individual particles constituting a molecule and $\vec{r}_{\mathrm{cm}}$ is the position vector of its centre of mass. The sum runs over all $N$ particles comprising the molecule. For more information see any polymer science book, e.g. [50]. If `<rg>` is used, the radius of gyration is averaged over all stored configurations (see section 8.3 on page 150).

*Output format*
{ *rg* *error_of_rg* *rg2* *error_of_rg2* }

### Hydrodynamic radius

*Python Syntax* (41)

```
espressomd.analyze.Analysis(system).calc_rh(
  system = ⟨system⟩,
  chain_start = ⟨int⟩,
  n_chains = ⟨int⟩,
  chain_length = ⟨int⟩)
```

*TCL Syntax*

```
analyze ( rh | <rh> ) [chain_start n_chains chain_length]
```

*Description*
Returns the hydrodynamic radius averaged over all chains. If `<rh>` is used, the hydrodynamic radius is averaged over all stored configurations (see section 8.3 on page 150). The following formula is used for the computation:

$$\frac{1}{R_{\mathrm{H}}} = \frac{2}{N^2} \sum_{i=1}^{N} \sum_{j=i}^{N} \frac{1}{|\vec{r}_i - \vec{r}_j|} \,, \tag{8.5}$$

The above-mentioned formula is only valid under certain assumptions. For more information, see Chapter 4 and equation 4.102 in [21].

*Output format*
{ *rh* *error_of_rh* }

### Internal distances

*TCL Syntax*

```
analyze ( internal_dist | <internal_dist> ) [chain_start n_chains chain_length]
```

*Description*
Returns the averaged internal distances within the chains (over all pairs of particles). If `<internal_dist>` is used, the values are averaged over all stored configurations (see section 8.3 on page 150).

*Output format*
{ *idf*(*0*) *idf*(*1*) ... *idf*(*chain_length* − *1*) }

The index corresponds to the number of beads between the two monomers considered (0 = next neighbours, 1 = one monomer in between, ...).

### Internal distances II (specific monomer)

*TCL Syntax*

```
analyze ( bond_dist | <bond_dist> ) [index index]
        [chain_start n_chains chain_length]
```

*Description*

In contrast to `analyze internal_dist`, it does not average over the whole chain, but rather takes the chain monomer at position *index* (default: 0, *i.e.* the first monomer on the chain) to be the reference point to which all internal distances are calculated. If `<bond_dist>` is used, the values will be averaged over all stored configurations (see section 8.3 on page 150).

*Output format*

{ $bdf(0)$ $bdf(1)$ ... $bdf(chain_length - 1 - index)$ }

### Bond lengths

*TCL Syntax*

```
analyze ( bond_l | <bond_l> ) [chain_start n_chains chain_length]
```

*Description*

Analyzes the bond lengths of the chains in the system. Returns its average, the standard deviation, the maximum and the minimum. If you want to look only at specific chains, use the optional arguments, *i.e.* $chain_start = 2 * MPC$ and $n_chains = 1$ to only include the third chain's monomers. If `<bond_l>` is used, the value will be averaged over all stored configurations (see section 8.3 on page 150). This function assumes linear chain topology and does not check if the bonds really exist!

*Output format*

{ *mean stddev max min* }

### Form factor

*TCL Syntax*

```
analyze ( formfactor | <formfactor> ) qmin qmax qbins
        [chain_start n_chains chain_length]
```

*Description*

Check this!

Computes the spherically averaged form factor of a single chain, which is defined by

$$S(q) = \frac{1}{chain_length} \sum_{i,j=1}^{chain_length} \frac{\sin(qr_{ij})}{qr_{ij}} \tag{8.6}$$

of a single chain, averaged over all chains for $qbin + 1$ logarithmically spaced q-vectors $qmin, \ldots, qmax$ where $qmin > 0$ and $qmax > qmin$. If `<formfactor>` is used, the form factor will be averaged over all stored configurations (see section 8.3 on the following page).

*Output format*

{
  { $q$ $S(q)$ }
  ⋮
}

with $q \in \{qmin, \ldots, qmax\}$.

### Chain radial distribution function

*Python Syntax* (42)

```
espressomd.analyze.Analysis(system).rdfchain(
  system = ⟨system⟩,
  r_min = ⟨float⟩,
  r_max = ⟨float⟩,
  r_bins = ⟨float⟩,
  chain_start = ⟨float⟩,
  n_chains = ⟨int⟩,
  chain_length = ⟨float⟩)
```

*TCL Syntax*

```
analyze rdfchain rmin rmax rbins [chain_start n_chains chain_length]
```

*Description*

Returns three radial distribution functions (rdf) for the chains. The first rdf is calculated for monomers belonging to different chains, the second rdf is for the centers of mass of the chains and the third one is the distribution of the closest distances between the chains (*i.e.* the shortest monomer-monomer distances). The distance range is given by *rmin* and *rmax* and it is divided into *rbins* equidistant bins.

*Output format*

{
  {$r$ $rdf1(r)$ $rdf2(r)$ $rdf3(r)$ }
  ⋮
}

#### Mean square displacement of chains

*TCL Syntax*

(1) `analyze ( <g1>| <g2>| <g3> )` [*chain_start n_chains chain_length*]
(2) `analyze g123 [-init]` [*chain_start n_chains chain_length*]

*Description*
Variant (1) returns

- the mean-square displacement of the beads in the chain (`<g1>`)
- the mean-square displacement of the beads relative to the center of mass of the chain (`<g2>`)
- or the motion of the center of mass (`<g3>`)

averaged over all stored configurations (see section 8.3). At short time scales, `g1` and `g2` coincide, since the motion of the center of mass is much slower. At large timescales `g1` and `g3` coincide and correspond to the center of mass motion, while `g2` levels off. `g2` and `g3` together correspond to `g1`. For details, see Grest and Kremer [28].

Variant (2) returns all of these observables for the current configuration, as compared to the reference configuration. The reference configuration is set, when the option `-init` is used.

*Output format (variant* (1)*)*
{ $gi(0 * dt)$ $gi(1 * dt)$ ... }

*Output format (variant* (2)*)*
{ $g1(t)$ $g2(t)$ $g3(t)$ }

## 8.3. Storing configurations

Some observables (*i.e.* non-static ones) require knowledge of the particles' positions at more than one or two times. Therefore, it is possible to store configurations for later analysis. Using this mechanism, the program is also able to work quasi-offline by successively reading in previously saved configurations and storing them to perform any analysis desired afterwards.

Note that the time at which configurations were taken is not stored. The most observables that work with the set of stored configurations do expect that the configurations are taken at equidistant timesteps.

Note also, that the stored configurations can be written to a file and read from it via the `blockfile` command (see section 10.5 on page 173).

### 8.3.1. Storing and removing configurations

*TCL Syntax*

(1) `analyze append`
(2) `analyze remove` [*index*]
(3) `analyze replace` *index*
(4) `analyze push` [*size*]
(5) `analyze configs` *config*

*Description*

Variant (1) appends the current configuration to the set of stored configurations. Variant (2) removes the *index*th stored configuration, or all, if *index* is not specified. Variant (3) will replace the *index*th configuration with the current configuration.

Variant (4) will append the current configuration to the set of stored configuration and remove configurations from the beginning of the set until the number of stored configurations is equal to *size*. If *size* is not specified, only the first configuration in the set is removed.

Variants (1) to (4) return the number of currently stored configurations.

Variant (5) will append the configuration *config* to the set of stored configurations. *config* has to define coordinates for all configurations in the format:

{*x1 y1 z1 x2 y2 z2 ...* }

### 8.3.2. Getting the stored configurations

*TCL Syntax*

(1) `analyze configs`
(2) `analyze stored`

*Description*

Variant (1) returns all stored configurations, while variant (2) returns only the number of stored configurations.

*Output format (variant* (1)*)*

{
  {*x1 y1 z1 x2 y2 z2 ...* }
  ⋮
}

## 8.4. `uwerr`: Computing statistical errors in time series

*TCL Syntax*

(1) `uwerr` *data nrep col* [*s_tau*] [`plot`]
(2) `uwerr` *data nrep f* [*s_tau* [*f_args*]] [`plot`]

*Description*

Calculates the mean value, the error and the error of the error for an arbitrary numerical time series according to Wolff [65].

*Arguments*

- *data* is a matrix filled with the primary estimates $a_\alpha^{i,r}$ from $R$ replica with $N_1, N_2, \ldots, N_R$ measurements each.

$$data = \begin{pmatrix} a_1^{1,1} & a_2^{1,1} & a_3^{1,1} & \cdots \\ a_1^{2,1} & a_2^{2,1} & a_3^{2,1} & \cdots \\ \vdots & \vdots & \vdots & \vdots \\ a_1^{N_1,1} & a_2^{N_1,1} & a_3^{N_1,1} & \cdots \\ a_1^{1,2} & a_2^{1,2} & a_3^{1,2} & \cdots \\ \vdots & \vdots & \vdots & \vdots \\ a_1^{N_R,R} & a_2^{N_R,R} & a_3^{N_R,R} & \cdots \end{pmatrix}$$

- *nrep* is a vector whose elements specify the length of the individual replica.

$$nrep = (N_1, N_2, \ldots, N_R)$$

- $f$ is a user defined Tcl function returning a double with first argument a vector which has as many entries as data has columns. If $f$ is given instead of the column, the corresponding derived quantity is analyzed.
- *f_args* are further arguments to $f$.
- *s_tau* is the estimate $S = \tau/\tau_{\text{int}}$ as explained in section (3.3) of [65]. The default is 1.5 and it is never taken larger than $\min_{r=1}^{R} N_r/2$.
- [`plot`] If plot is specified, you will get the plots of $\Gamma/\Gamma(0)$ and $\tau_{int}$ vs. $W$. The data and gnuplot script is written to the current directory.

*Output format*

*mean error error_of_error act*
*error_of_act* $[Q]$

where *act* denotes the integrated autocorrelation time, and $Q$ denotes a *quality measure*, *i.e.* the probability to find a $\chi^2$ fit of the replica estimates.

The function returns an error message if the windowing failed or if the error in one of the replica is to large.

# 9. Analysis in the core

Analysis in the core is a new concept introduced in ESPResSo since version 3.1. It was motivated by the fact, that sometimes it is desirable that the analysis functions do more than just return a value to the scripting interface. For some observables it is desirable to be sampled every few integrations steps. In addition, it should be possible to pass the observable values to other functions which compute history-dependent quantities, such as correlation functions. All this should be done without the need to interrupt the integration by passing the control to the script level and back, which produces a significant overhead when performed too often.

Some observables in the core have their corresponding counterparts in the Tcl observables of the `analyze` command described in Chapter 8. However, only the core-observables can be used on the fly with the toolbox of the correlator and on the fly analysis of time series. Similarly, some special cases of using the correlator have their redundant counterparts in the analysis in Tcl (Chapter 8), but the correlator provides a general and versatile toolbox which can be used with any implemented core-observables. The only trick to bridge the gap between Tcl based analysis and core analysis is the tclcommand observable that allows use the return value of arbitrary Tcl functions (also self-written) as input for the core analysis. See more below.

## 9.1. Observables

### 9.1.1. Introduction

The first step of the core analysis is to tell ESPResSo to create an observable. An observable in the sense of the core analysis can be considered as a rule how to compute a certain set of numbers from a given state of the system or a role how to collect data from other observables. Any observable is represented as a single array of double values. Any more complex shape (tensor, complex number, ...) must be compatible to this prerequisite. Every observable however documents the storage order.

Creating an observable means just allocating the corresponding memory, assigning a function to compute the observable value and reserving an *id* which will be used to refer to the observable. In addition to the possibility to print the observable value (return the observable value to the script interface), the *id* of a core-observable can be passed to another analysis function. The observable value is computed from the current state of the system at the moment when it is needed, *i.e.* when requested explicitly by the user calling the `observable print` function or when requested automatically by some other analysis function. Updating is an orthogonal concept: Observables that collect

data over time (e.g. the average observable) need to be updated regularly, even though their current value is not of interest.

Not all observables are implemented in parallel. When performing a parallel computation, too frequent updates to observables which are not implemented in parallel may produce a significant slowdown.

### 9.1.2. Creating an observable

To create a new observable, use

*TCL Syntax*

`observable new` *name* [*parameters+*]

*Description*

Upon this call, ESPResSo allocates the necessary amount of memory and returns an integer *id* which will be used later to refer to the observable. The parameter *name* and further arguments have to correspond to one of the observables described below.

#### Available observables

Currently the following observables are implemented. Particle specifications (see section 9.1.7 below) define a group of particles, from which the observable should be calculated. They are generic to all observables and are described after the list of observables.

Missing descriptions of parameters of several observables

Here are the observables, that only depend on the current state of the simulation system:

- `particle_positions` *particle_specifications*
  Positions of the particles, in the format $x_1,\ y_1,\ z_1,\ x_2,\ y_2,\ z_2,\ \ldots\ x_n,\ y_n,\ z_n$. The particles are ordered ascending according to their ids.

- `particle_velocities` *particle_specifications*
  Velocities of the particles, in the format $v_1^x,\ v_1^y,\ v_1^z,\ v_2^x,\ v_2^y,\ v_2^z,\ \ldots\ v_n^x,\ v_n^y,\ v_n^z$. The particles are ordered ascending according to their ids.

- `particle_body_velocities` *particle_specifications*
  Velocities of the particles in the body frame, in the format $v_1^x,\ v_1^y,\ v_1^z,\ v_2^x,\ v_2^y,\ v_2^z,\ \ldots\ v_n^x,\ v_n^y,\ v_n^z$. The particles are ordered ascending according to their ids. This command only produces a meaningful result when `ROTATIONS` is compiled in.

- `particle_forces` *particle_specifications*
  Forces on the particles, in the format $f_1^x,\ f_1^y,\ f_1^z,\ f_2^x,\ f_2^y,\ f_2^z,\ \ldots\ f_n^x,\ f_n^y,\ f_n^z$. The particles are ordered ascending according to their ids.

- `particle_angular_momentum` *particle_specifications*
  Angular momenta (omega) of the particles, in the format
  $\omega_1^x,\ \omega_1^y,\ \omega_1^z,\ \omega_2^x,\ \omega_2^y,\ \omega_2^z,\ \ldots\ \omega_n^x,\ \omega_n^y,\ \omega_n^z$. The particles are ordered ascending according to their ids and the angular velocity/momentum is specified in the laboratory frame.

- `particle_body_angular_momentum` *particle_specifications*
  Angular momenta (omega) of the particles, in the format
  $\omega_1^x,\ \omega_1^y,\ \omega_1^z,\ \omega_2^x,\ \omega_2^y,\ \omega_2^z,\ \ldots\ \omega_n^x,\ \omega_n^y,\ \omega_n^z$. The particles are ordered ascending according to their ids and the angular velocity/momentum is specified in the body (co-rotating) frame.

- `com_position` *particle_specifications* [blocked *size*]
  Position of the centre of mass. If `blocked` *size* is specified, the particles are subdivided into blocks of size *size* and the centre of mass position is calculated for each block separately.

- `com_velocity` *particle_specifications* [blocked *size*]
  Velocity of the centre of mass. If `blocked` *size* is specified, the particles are subdivided into blocks of size *size* and the centre of mass velocity is calculated for each block separately.

- `com_force` *particle_specifications* [blocked *size*]
  Total force on the specified particles. If `blocked` *size* is specified, the particles are subdivided into blocks of size *size* and the total force is calculated for each block separately.

- `stress_tensor`
  The stress tensor. It only works with all particles. It is returned as a 9-dimensional array:
  $\{\ \sigma_{xx},\ \sigma_{xy},\ \sigma_{xz},\ \sigma_{yx},\ \sigma_{yy},\ \sigma_{yz},\ \sigma_{zx},\ \sigma_{zy},\ \sigma_{zz}\ \}$

- `stress_tensor_acf_obs`
  The observable for computation of the Stress tensor autocorrelation function. Similarly to the stress tensor, it only works with all particles. It is returned as a 6-dimensional array:
  $\{\ \sigma_{xy},\ \sigma_{yz},\ \sigma_{zx},\ (\sigma_{xx}-\sigma_{yy}),\ (\sigma_{xx}-\sigma_{zz}),\ (\sigma_{yy}-\sigma_{zz})\ \}$
  where $\sigma_{ij}$ are the components of the stress tensor.

  any suggestion for a more suitable name?

- `particle_currents` *particle_specifications*
  Electric currents due to individual particles. For a particle $i$: $j_i^x = q_i v_i^x/\Delta t$ where $\Delta t$ is the simulation time step. Required feature: ELECTROSTATICS

- `currents` *particle_specifications*
  Electric currents averaged over all particles: $j^x = \sum_i q_i v_i^x/\Delta t$ where $\Delta t$ is the simulation time step. Required feature: ELECTROSTATICS

- `dipole_moment` *particle_specifications*
  The dipole moment of the specified group of particles: $\mu^x = \sum_i q_i r_i^x$ Required feature: ELECTROSTATICS

- `com_dipole_moment` *particle_specifications*
  Total dipole moment of the specified group of particles: $\mu^x = \sum_i d_i^x$ where $d_i^x$ is an intrinsic dipole moment of the individual $i$-th particle. Required feature: DIPOLES

- `interacts_with` *particle_specifications1 particle_specifications2 cutoff*
  For each particle belonging to *particle_specifications1* the observable is unity if a neighbour of a type from *particle_specifications2* is found within the distance defined by the *cutoff*. If no such neighbour is found, the observable is zero. The observable has one dimension per each particle of *particle_specifications*$_1$

- `density_profile` *particle_specifications profile_specifications*
  Compute the density profile within the specified cube. For profile specifications, see section 9.1.8.

- `force_density_profile` *particle_specifications profile_specifications*
  Compute the force density profile within the specified cube. For profile specifications, see section 9.1.8.

- `lb_velocity_profile` *particle_specifications profile_specifications*
  Compute the Lattice-Boltzmann velocity profile within the specified cube. For profile specifications, see section 9.1.8.

- `flux_density_profile` *particle_specifications profile_specifications*
  Compute the flux density within the specified cube. For profile specifications, see section 9.1.8.

- `radial_density_profile` Compute the density profile in cylindrical coordinates. For profile specifications, see section 9.1.8.

- `radial_flux_density_profile`
  Compute the flux density profile in cylindrical coordinates. For profile specifications, see section 9.1.8.

- `lb_radial_velocity_profile`
  Compute the Lattice-Boltzmann velocity profile in cylindrical coordinates. For profile specifications, see section 9.1.8.

- `rdf` *type_listtype_list*[*r_min*[*r_max*[*n_bins*]]]
  Compute the radial distribution function, see 8.1.12.

- `structure_factor` *order*
  Compute the structure factor. Remember it scales as order$^3$, see 8.1.13.

- `radial_density_distribution` *type* < *type* > *minr* < *minr* >\
  *maxr* < *maxr* > *rbins* < *rbins* > (*start_point* < *X* >< *Y* >< *Z* >\
  *end_point* < *X* >< *Y* >< *Z* > |*id_start_point* < *id* > *id_end_point* < *id* >)
  Computes the radial density distribution for particles of the given type around the axis given either as fixed positions or as particle ids. The binning is done between `minr` and `maxr` with `rbins`.

- `spatial_polymer_property` (*ids* < *id_list* > |*type* < *type* >)*N* < *Npoly* >
  Calculates the mean charge along `Npoly` weak polyelectrolytes of the *same* length. The particles can be specified as a list of particle ids or through the particle type.

- `persistence_length` *ids* < *id_list* > *max_d* < *max_d* > *cut_off* < *cut_off* >
  Calculates the persistence length of the given polymer specified through the `id_list`. `max_d` is the maximum distance (in terms of particles) for which the correlation is computed. With `cut_off` the number of particles at the polymer ends that are ignored for the calculation, can be specified.

- `polymer_pair_correlation` *ids* < *id_list* > *maxr* < *maxr* > *minr* < *minr* > *k* < *k* >\
  *rbins* < *rbins* > *N* < *Npoly* > *poly_len* < *poly_len* >
  Computes the pair correlation of particles on a polymer chain given through the `id_list`, here `k` is the distance (in terms of particles) between the mononers on the chain for which the pair correlation is computed. This is averaged over the whole chain and all the given chains. The number of polymers for which this distribution is computed has to be given as `Npoly` and their length through `poly_len`. The distribution is calculated for distances between `minr` and `maxr` with `rbins`.

The tclcommand observable is a helpful tool, that allows to make the analysis framework much more versatile, by allowing the evaluation of arbitrary tcl commands.

- `tclcommand` *dimQ command*
  An arbitrary Tcl function that returns a list of floating point numbers of fixed size *dimQ* can be specified. Although its execution might be slow, it allows to prototype new observables without a lot of trouble. Many existing analysis commands can be made to cooperate with the core analysis that way.

The following commands allow to collect data automatically over time once their autoupdate feature is enabled.

- `average` *ref*
  The running average of the reference observable with id *ref*. It can be resetted by `observable` *no* `reset`

### 9.1.3. Printing an observable

*TCL Syntax*

| `observable` *id* `print` [`no_calculation`] [`formatted`]

*Description*

Prints the value of the observable with a given *id*. If the observable refers to the current state of the system, its value is updated before printing, except if `no_calculation` is given.

Formatted printing is not fully supported yet.

### 9.1.4. Checkpointing observables

*TCL Syntax*

`observable` *id* `write_checkpoint` *filename* [`binary`]

*Description*

Writes the current state of the observable to `filename`. If `binary` is given then it does so in binary form. This is useful to save the state of statful observables as the average. This checkpoing mechanism is not portable, rereading is only guaranteed to work on the same system with the same Espresso binary.

*TCL Syntax*

`observable` *id* `read_checkpoint` *filename* [`binary`]

*Description*

Reads the state of the observable from a checkpoint file. The observable has to be configured beforehand in the script in exactly the same way as it was when the checkpoint was written. That means that the configuration of the observable – including possible particle and profile specifications – is not saved in the file but has to be provided by the user. Please be aware that the value of observables that refer to the current state of the system are overwritten by the `print` command except if the option `no_calculation` is given.

### 9.1.5. Passing an observable to an analysis function

Currently the only analysis function which uses the core observables is the correlator (section 9.2).

### 9.1.6. Deleting an observable to an analysis function

*TCL Syntax*

`observable` *id* `delete`

*Description*

Deletes the observable, *i.e.* frees the allocated memory and makes the *id* free for a new observable.

Does not work yet

### 9.1.7. Particle specifications

You can specify from which particles the observable should be computed in one of the following ways. In all cases, particle specifications refer to the current state of espresso. Any later changes to particles (additions, deletions, changes of types) will not be automatically reflected in the observable.

- `all`
  Requests observable calculation based on all particles in the system.

- `types` *type_list*
  Restricts observable calculation to a given particle type(s). The type list is a tcl list of existing particle types.

- `id` *id_list*
  Restricts observable calculation to a given list of particle id(s). The id list is a tcl list of existing particle ids.

### 9.1.8. Profile specifications

Profiles are specified by giving the spacial area that is to be profiled and the number of bins in each spacial direction. The area to be analyzed is characterized by *minx*/*maxx* *miny*/*maxy* and *minz*/*maxz*. The defaults correspond to the box size when the observable is created. The bin size in each direction defaults to 1, and can be change with the parameter *xbins*/*ybins*/*zbins*. Changing one, two or three of them to a value $> 1$ with thus create a one-, two- or three-dimensional map of the desired quantity. The full syntax thus reads as:

*TCL Syntax*

```
observable new needs_profile_specs [other_parameters] [ minx minx ]
      [ maxx maxx ] [ miny miny ] [ maxy maxy ] [ minz minz ]
      [ maxz maxz ] [ xbins xbins ] [ ybins ybins ] [ zbins zbins ]
```

*Description*

Radial profiles allow to do the same as usual profiles, except the coordinate system is a cylindrical one and the binning is done in the cylindrical coordinates (defined with the axis in z-direction). This is very helpful if the symmetry of the system is cylindrical. The spacial are is characterized by a center (default to the center of the box) a maximum radial position *maxr* (defaults to the smaller value of the box lengths in x and y directions) and a minimum and maximum value of $z$. It is possible to also resolve different polar angles, thus using it as a full 3D mapping tool, but this will only rarely be used. The full syntax is:

*TCL Syntax*

```
observable new needs_radial_profile_specs [other_parameters]
      [ center <cx> <cy> <cx> ] [ maxr maxr ] [ minz minz ]
      [ maxz maxz ] [ rbins rbins ] [ phibins phibins ] [ zbins zbins ]
```

*Description*

## 9.2. Correlations

### 9.2.1. Introduction

Time correlation functions are ubiquitous in statistical mechanics and molecular simulations when dynamical properties of many-body systems are concerned. A prominent example is the velocity autocorrelation function, $\langle \mathbf{v}(t) \cdot \mathbf{v}(t+\tau) \rangle$ which is used in the Green-Kubo relations. In general, time correlation functions are of the form

$$C(\tau) = \langle A(t) \otimes B(t+\tau) \rangle \ , \tag{9.1}$$

where $t$ is time, $\tau$ is the lag time (time difference) between the measurements of (vector) observables $A$ and $B$, and $\otimes$ is an operator which produces the vector quantity $C$ from $A$ and $B$. The ensemble average $\langle \cdot \rangle$ is taken over all time origins $t$. Correlation functions describing dynamics of large and complex molecules such as polymers span many orders of magnitude, ranging from MD time step up to the total simulation time.

ESPResSo uses a fast correlation algorithm (see section 9.2.6) which enables efficient computation of correlation functions spanning many orders of magnitude in the lag time.

The generic correlation interface of ESPResSo may process either observables defined in the kernel, or data which it reads from an external file or values entered through the scripting interface. Thus, apart from data processing on the fly, it can also be used as an efficient correlator for stored data. In all cases it produces a matrix of $n+2$ columns. The first two columns are the values of lag times $\tau$ and the number of samples taken for a particular value of $\tau$. The remaining ones are the elements of the $n$-dimensional vector $C(\tau)$.

Processing data from Tcl input or from input files is not fully supported yet.

The `uwerr` command for computing averages and error estimates of a time series of observables relies on estimates of autocorrelation functions and the respective autocorrelation times. The correlator provides the same functionality as a by-product of computing the correlation function (see section 9.2.5.

An example of the usage of observables and correlations is provided in the script `correlation.tcl` in the samples directory.

### 9.2.2. Creating a correlation

Correlation first has to be defined by saying which observables are to be correlated, what should be the correlation operation, sampling frequency, etc. When a correlation is defined, its id is returned which is used further to do other operations with the correlation. The correlation can be either updated automatically on the fly without direct user intervention, or by an explicit user call for an update.

*TCL Syntax*

```
correlation new obs1 id1 [obs2 id2] corr_operation
      operation dt dt tau_max tau_max [tau_lin tau_lin]
      [compress1 name [compress2 name] ]
```

*Description*

Defines a new correlation and returns an integer *id* which has been assigned to it. Its further arguments are described below.

*Arguments*

- `obs1` and `obs2`
  are ids of the observables A and B that are to correlated. The ids have to refer to existing observables which have been previously defined by the `observable` command. Some observables are already implemented, and others can be easily added. This can be done with very limited ESPResSo knowledge just by following the implementations that are already in. If `obs2` is omitted, autocorrelation of `obs1` is calculated by default.

- `corr_operation`
  The operation that is performed on $A(t)$ and $B(t+\tau)$ to obtain $C(\tau)$. The following operations are currently is available:

  - `scalar_product`
    Scalar product of $A$ and $B$, *i.e.* $C = \sum_i A_i B_i$
  - `componentwise_product`
    Comnponentwise product of $A$ and $B$, *i.e.* $C_i = A_i B_i$
  - `square_distance_componentwise`
    Each component of the correlation vector is the square of the difference between the corresponding components of the observables, *i.e.* $C_i = (A_i - B_i)^2$. Example: when $A$ is `particle_positions`, it produces the mean square displacement (for each component separately).
  - `tensor_product`
    Tensor product of $A$ and $B$, *i.e.* $C_{i \cdot l_B + j} = A_i B_j$, with $l_B$ the length of $B$.
  - `complex_conjugate_product`

    Complex conjugate product must be defined.
  - `fcs_acf` $w_x$ $w_y$ $w_z$
    Fluorescence Correlation Spectroscopy (FCS) autocorrelation function, *i.e.*

$$G_i(\tau) = \frac{1}{N} \left\langle \exp\left( -\frac{\Delta x_i^2(\tau)}{w_x^2} - \frac{\Delta y_i^2(\tau)}{w_y^2} - \frac{\Delta z_i^2(\tau)}{w_z^2} \right) \right\rangle , \tag{9.2}$$

    where $\Delta x_i^2(\tau) = \big(x_i(0) - x_i(\tau)\big)^2$ is the square discplacement of particle $i$ in the $x$ direction, and $w_x$ is the beam waist of the intensity profile of the exciting laser beam,

$$W(x,y,z) = I_0 \exp\left( -\frac{2x^2}{w_x^2} - \frac{2y^2}{w_y^2} - \frac{2z^2}{w_z^2} \right) . \tag{9.3}$$

    Equation (9.2) is a generalization of the formula presented by Höfling *et al.* [31]. For more information, see references therein. Per each 3 dimensions of the observable, one dimension of the correlation output is produced.

If `fcs_acf` is used with other observables than `particle_positions`, the physical meaning of the result is unclear.

- `dt`
  The time interval of sampling data points. When autoupdate is used, *dt* has to be a multiple of timestep. It is also used to produce time axis in real units. *Warning: if dt is close to the timestep, autoupdate is strongly recommended. Otherwise cpu time is wasted on passing the control between the script and kernel.*

- `tau_max`
  This is the maximum value of $\tau$ for which the correlation should be computed. *Warning: Unless you are using the multiple tau correlator, choosing tau_max of more than 100dt will result in a huge computational overhead. In a multiple tau correlator with reasonable parameters, tau_max can span the entire simulation without too much additional cpu time.*

- `tau_lin`
  The number of data-points for which the results are linearly spaced in tau. This is a parameter of the multiple tau correlator. If you want to use it, make sure that you know how it works. By default, it is set equal to *tau_max* which results in the trivial linear correlator. By setting *tau_lin* $<$ *tau_max* the multiple tau correlator is switched on. In many cases, *tau_lin*=16 is a good choice but this may strongly depend on the observables you are correlating. For more information, we recommend to read Ref. [48] or to perform your own tests.

- `compress1` and `compress2`
  Are functions used to compress the data when going to the next level of the multiple tau correlator. Different compression functions for different observables can be specified if desired, otherwise the same function is used for both. Default is `discard` which takes one of the observable values and discards the other one. This is safe for all observables but produces poor statistics in the tail. For some observables, `linear` compression can be used which makes an average of two neighbouring values but produces systematic errors. Depending on the observable, the systematic error can be anything between harmless and disastrous. For more information, we recommend to read Ref. [48] or to perform your own tests.

### 9.2.3. Inquiring about already existing correlations

*TCL Syntax*

(1) `correlation`
(2) `correlation n_corr`

*Description*

Maybe not all parameters are printed.

Variant (1) returns a tcl list of the defined correlations including their parameters. Variant (2) returns the number of currently defined correlations.

### 9.2.4. Collecting time series data for the correlation

*TCL Syntax*

(1) `correlation` *id* `autoupdate { start | stop}`
(2) `correlation` *id* `update`
(3) `correlation` *id* `finalize`

*Description*

Variant (1) is the recommended way of updating the correlations. By specifying `start` or `stop` it starts or stops automatically updating the correlation estimates. The automatic updates are done within the integration loop without further user intervention. The update frequency is adjusted based on the value of *dt* provided when defining the correlation. Note that autoupdate has to be started after setting the sim-time (e.g. after *setmdtime0*).

Variant (2) is an explicit call for an instantaneous update of the correlation estimates, using the current system state. It is only possible to use (2) if the correlation is not being autoupdated. However, it is possible to use it after autoupdate has been stopped. When updating by an explicit call, ESPResSo does not check if the lag time between two updates corresponds the value of *dt* specified when creating the correlation.

Variant (3) correlates all data from history which are left in the buffers. Once this has been done, the history is lost and no further updates are possible. When a new observable value is passed to a correlation, level 0 of the compression buffers of the multiple tau correlator (see section 9.2.6 for details) is updated immediately. Higher levels are updated only when the lower level buffers are filled and there is a need to push some values one level up. When the updating is stopped, a number of observable values have not reached the higher level, especially when *tau_max* is comparable to the total simulation time and if there are many compression levels. In such case, variant (3) is very useful. If *tau_max* is much shorter, it does not have a big effect.

### 9.2.5. Printing out the correlation and related quantities

*TCL Syntax*

(1) `correlation` *id* `write_to_file` *filename*
(2) `correlation` *id* `print`
(3a) `correlation` *id* `print [ average1 | variance1 | correlation_time ]`
(3b) `correlation` *id* `print [ average_errorbars ]`

*Description*

Variant (1) writes the current status of the correlation estimate to the specified filename. If the file exists, its contents will be overwritten.

*Output format*

The output looks as follows:

```
tau1 n_samples C1 C2 ... Cn
tau2 n_samples C1 C2 ... Cn
```

Where each line corresponds to a given value of `tau`, `n_samples` is the number of samples which contributed to the correlation at this level and $C_i$ are the individual components of the correlation.

Variant (2) returns the current status of the correlation estimate as a Tcl variable.

*Output format*
The output looks as follows:

```
tau1 n_samples C1 C2 ... Cn
tau2 n_samples C1 C2 ... Cn
```

Variants (3a) and (3b) return the corresponding estimate of the statistical property as a Tcl variable.
`average1` prints the average of observable1.
`variance1` prints the variance of observable1.
`correlation_time` prints the estimate of the correlation time.
`average_errorbars` prints the estimate of the error of the average based on the method according to [65] (same as used by the `uwerr` command).

### 9.2.6. The correlation algorithm: multiple tau correlator

Here we briefly describe the multiple tau correlator which is implemented in ESPResSo. For a more detailed description and discussion of its behaviour with respect to statistical and systematic errors, please read the cited literature. This type of correlator has been in use for years in the analysis of dynamic light scattering [51]. About a decade later it found its way to the Fluorescence Correlation Spectroscopy (FCS) [40]. The book of Frenkel and Smit [26] describes its application for the special case of the velocity autocorrelation function.

Let us consider a set of $N$ observable values as schematically shown in figure 9.1, where a value of index $i$ was measured in time $i\delta t$. We are interested in computing the correlation function according to equation (9.1) for a range lag times $\tau = (i-j)\delta t$ between the measurements $i$ and $j$. To simplify the notation, we further drop $\delta t$ when referring to observables and lag times.

The trivial implementation takes all possible pairs of values corresponding to lag times $\tau \in [\tau_{\min} : \tau_{\max}]$. Without loss of generality, let us further consider $\tau_{\min} = 0$. The computational effort for such an algorithm scales as $\mathcal{O}(\tau_{\max}^2)$. As a rule of thumb, this is feasible if $\tau_{\max} < 10^3$. The multiple tau correlator provides a solution to compute the correlation functions for arbitrary range of the lag times by coarse-graining the high $\tau$ values. It applies the naive algorithm to a relatively small range of lag times $\tau \in [0 : p-1]$. This we refer to as compression level 0. To compute the correlations for lag times $\tau \in [p : 2(p-1)]$, the original data are first coarse-grained, so that $m$ values of the original data are compressed to produce a single data point in the higher compression level. Thus the lag time between the neighbouring values in the higher compression level increases by a factor of $m$, while the number of stored values decreases by the same factor and the number of correlation operations at this level reduces by a factor of $m^2$.

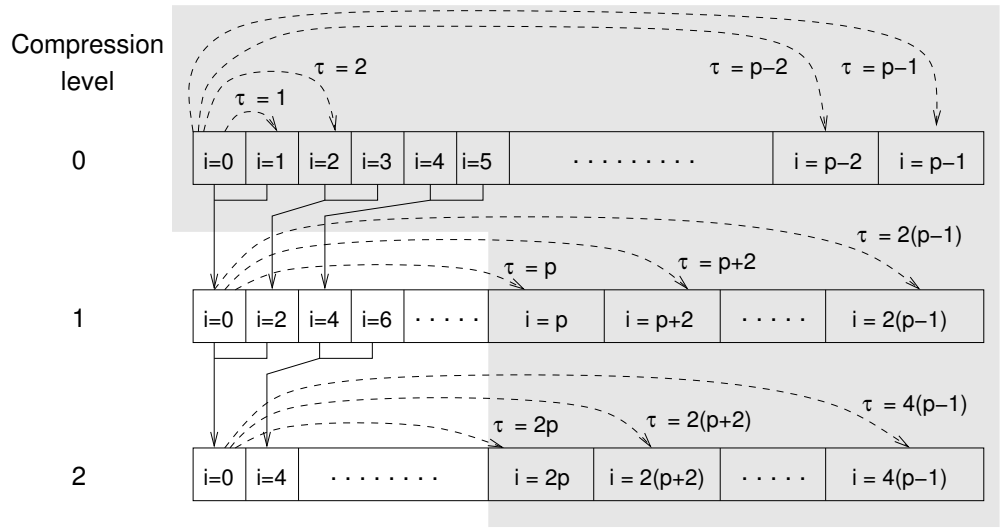



Figure 9.1.: Schematic representation of buffers in the correlator.

Correlations for lag times $\tau \in [2p : 4(p-1)]$ are computed at compression level 2, which is created in an analogous manner from level 1. This can continue hierarchically up to an arbitrary level for which enough data is available. Due to the hierarchical reduction of the data, the algorithm scales as $\mathcal{O}\big(p^2 \log(\tau_{\max})\big)$. Thus an additional order of magnitude in $\tau_{\max}$ costs just a constant extra effort.

The speedup is gained at the expense of statistical accuracy. The loss of accuracy occurs at the compression step. In principle one can use any value of $m$ and $p$ to tune the algorithm performance. However, it turns out that using a high $m$ dilutes the data at high $\tau$. Therefore $m = 2$ is hard-coded in the ESPResSo correlator and cannot be modified by user. The value of $p$ remains an adjustable parameter which can be modified by user by setting `tau_lin` when defining a correlation. In general, one should choose $p \gg m$ to avoid loss of statistical accuracy. Choosing $p = 16$ seems to be safe but it may depend on the properties of the analyzed correlation functions. A detailed analysis has been performed in Ref. [48].

The choice of the compression function also influences the statistical accuracy and can even lead to systematic errors. The default compression function is `discard2` which discards the second for the compressed values and pushes the first one to the higher level. This is robust and can be applied universally to any combination of observables and correlation operation. On the other hand, it reduces the statistical accuracy as the compression level increases. In many cases, the `average` compression operation can be applied, which averages the two neighbouring values and the average then enters the higher level, preserving almost the full statistical accuracy of the original data. In general, if averaging can be safely used or not, depends on the properties of the difference

$$\frac{1}{2}(A_i \otimes B_{i+p} + A_{i+1} \otimes B_{i+p+1}) - \frac{1}{2}(A_i + A_{i+1}) \otimes \frac{1}{2}(B_{i+p} + B_{i+p+1}) \qquad (9.4)$$

For example in the case of velocity autocorrelation function, the above-mentioned difference has a small value and a random sign, *i.e.* different contributions cancel each other. On the other hand, in the of the case of mean square displacement the difference is always positive, resulting in a non-negligible systematic error. A more general discussion is presented in Ref. [48].

### 9.2.7. Checkpointing the correlator

It is possible to checkpoint the correlator. Thereby the data is written directly to a file. It may be usefull to write to a binary file, as this preserves the full bit-value of the variables, whereas the text file representation has a lower accuracy.

*TCL Syntax*

(1) `correlation` *id* `write_checkpoint_binary` *filename*
(2) `correlation` *id* `write_checkpoint_ascii` *filename*

*Description*

In order to load a checkpoint, the correlator has to be initialized. Therefore the observable(s) have to be created. Make sure that the correlator is exactly initilized as it was when the checkpoint was created. If this is not fullfilled, and e.g. the size of an observable has changed, loading the checkpoint failes.

*TCL Syntax*

(1) `correlation` *id* `read_checkpoint_binary` *filename*
(2) `correlation` *id* `read_checkpoint_ascii` *filename*

*Description*

Depending on whether the checkpoint was written as binary or as text, the corresponding variant for reading the checkpoint has to be used.

An simple example for checkpointing:

```
set pp  [observable new particle_positions all]
set cor1 [correlation new obs1 $pp corr_operation square_distance_componentwise \
dt 0.01 tau_max 1000 tau_lin 16]
integrate 1000
correlation $cor1 write_checkpoint_binary "cor1.bin"
```

And then to continue the simulation:

```
set pp  [observable new particle_positions all]
set cor1 [correlation new obs1 $pp corr_operation square_distance_componentwise \
dt 0.01 tau_max 1000 tau_lin 16]
correlation $cor1 read_checkpoint_binary "cor1.bin"
```

# 10. Input / Output

## 10.1. No generic checkpointing!

One of the most asked-for feature that seems to be missing in ESPResSo is *checkpointing*, *i.e.* a simple way to tell ESPResSo to store and restore the current state of the simulation, and to be able to write this state to or read it from a file. This would be most useful to be able to restart a simulation from a specific point in time.

Unfortunately, it is impossible to provide a simple command (*e.g.* `checkpoint`), out of two reasons. The main reason is that ESPResSo has no way to determine what information constitutes the actual state of the simulation. On the one hand, ESPResSo scripts sometimes use Tcl-variables that contain essential information about a simulation, *e.g.* the stored values of an observable that was computed in previous time steps, counters, etc. These would have to be contained in a checkpoint. However, not all Tcl-variables are of interest. For example, Tcl has a number of automatically set variables that contain information about the hostname, the machine type, etc. These variables should most probably *not* be included the simulation state. ESPResSo has no way to distinguish between these variables. On the other hand, the ESPResSo core has a number of internal variables, *e.g.* the particle coordinates. While most of these are probably good candidates for being included into a checkpoint, this is not necessarily so. For example, when you have particles in your system that have fixed coordinates, should these be stored in a checkpoint, or not? If the system contains mostly fixed particles and only very few moving particles, this would increase the memory size of a checkpoint needlessly. And what about the interactions in the system, or the bonds? Should these be stored in a checkpoint, or are they generated by the script?

Another problem with a generic checkpoint would be the control flow of the script. In principle, the checkpoint would have to store where in the script the checkpointing function was called to be able to return there. All this is even further complicated by the fact that ESPResSo is running in parallel.

Instead, in ESPResSo, the user has to specify what information needs to be saved to a file to be able to restore the simulation state. The `blockfile` and `writemd` commands help you to do that. `blockfile` writes text files. When floating point numbers are stored in such files (*e.g.* the particle positions), there is only a limited precision. Therefore, it is not possible to bitwise reproduce a simulation state using this function. When you need bitwise reproducibility, you will have to use the command `writemd`, which stores positions, forces and velocities in binary format. Note that there is no command to write other MD parameters like time step or interactions in binary format. You should restore these using exactly the same Tcl command that you used to create them.

Finally, there is one more complication: random forces are computed in the order

the particles are stored in memory. This order usually differs after reading a blockfile back, since the particles are stored in consecutive identity order. In memory, they are usually not in a specific order. Therefore, you need to use `sort_particles` after writing a blockfile that you want to use for checkpointing, so that the particles are resorted to the same consecutive order. Note that this does not change physics, just the order the random numbers are applied.

When using an LB fluid, you need to also write out the fluid nodes, see the `lbfluid` command for further details.

## 10.2. Writing H5MD-files

For large amounts of data it's a good idea to store it in the hdf5 (H5MD is based on hdf5) file format (see `https://www.hdfgroup.org/` for details). ESPResSo currently supports some basic functions for writing simulation data to H5MD files.

### 10.2.1. `h5md_init`: Initialize a H5MD data file

*TCL Syntax*

| `h5md_init` */where/to/store/data/file* [*list_of_p_ids*]

*Description*

The `h5md_init` command initializes the H5MD file and creates groups and datasets according to the conventions of H5MD (see `http://nongnu.org/h5md/`). The init command assumes that the filename ends with the suffix ".h5". If there is no filename present matching with the given, the filename will be given the suffix "_1.h5". In order to safely start from a checkpoint the init command automatically copies the old file to a new file with incremented suffix index (e.g. "_2.h5" for the first start from a checkpoint). The optional argument *list_of_p_ids* can be provided. In this case observables are written for these particle ids only.

### 10.2.2. `h5md_write_positions`: Write particle positions to H5MD file

*TCL Syntax*

| `h5md_write_positions` [`folded`]

*Description*

Writes the positions of all particles to the dataset which was previously initialized by `h5md_init`.

### 10.2.3. `h5md_write_velocities`: Write particle velocities to H5MD file

*TCL Syntax*

| `h5md_write_velocities`

*Description*
Writes the velocities of all particles to the dataset which was previously initialized by `h5md_init`.

*TCL Syntax*

`|h5md_write_forces`

*Description*
Writes the forces of all particles to the dataset which was previously initialized by `h5md_init`.

### 10.2.4. `h5md_observable1D_init`: Initialize a user-defined H5MD observable

*TCL Syntax*

`|h5md_observable1D_init` *name_of_observable*

*Description*
Initializes an observable dataset of a zero dimensional, but timedependent, observable with given name in the H5MD file previously initialized by `h5md_init`.

### 10.2.5. `h5md_observable1D_write`: Write to a user-defined H5MD dataset

*TCL Syntax*

`|h5md_observable1D_write` *name_of_observable value*

*Description*
Writes `value` to a user-defined observable dataset with given name.

*TCL Syntax*

`|h5md_observable2D_init` *name_of_observable*

*Description*
Initializes an observable dataset of a one dimensional, but timedependent, observable with given *property* of particles in Espresso in the H5MD file previously initialized by `h5md_init`.

### 10.2.6. `h5md_observable2D_write`: Write to a user-defined H5MD dataset

*TCL Syntax*

`|h5md_observable2D_write` *name_of_observable value*

*Description*
Writes `value` to a user-defined observable dataset with given name.

### 10.2.7. `h5md_close`: Close H5MD file

*TCL Syntax*

```
| h5md_close
```

*Description*
Closes the H5MD data file.

## 10.3. Writing H5MD-files with Python

To write data in a hdf5-file, first an object of the class `h5md` has to be created and linked to the respective hdf5-file. This may, for example, look like:

```
from espressomd import h5md
system = espressomd.System()
h5=h5md.h5md("File.h5",system)
```

The `h5md` class contains the sub-class `write_to_h5`, which contains all write commands such as `userdefined`. The sample python script "h5md_WriteFile.py" shows the usage of writing a hdf5-file.

### 10.3.1. `userdefined`: Write an arbitrary value

*Python Syntax*

```
| h5.write_to_h5.userdefined(value,groupname,datasetname,datatype,
        datasetindex,chunks)
```

*Description*
The command `userdefined` writes the value *value* of type *datatype* in the hdf5-file. The storage path is given by *groupname* and the dataset is called *datasetname*. Fragmenting the dataset to smaller units can increase the access speed to the hdf5-file. The dataset can be chunked to smaller dataset sub units with size *chunks*. When not defining the chunk size explicitly, it will be set to automatic.

*Arguments*

- *value* The value to be written.
- *groupname* Name of the group, which contains the dataset.
- *datasetname* Name of the dataset, which contains the values.
- *datatype* The value's data type.
- *datasetindex* Position of *value* within the dataset.
- *chunks* Size of dataset sub units.

### 10.3.2. `particle_property`: Write particle properties

*Python Syntax*

`|h5.write_to_h5.particle_property(`*timestep*`,`*groupname*`,`*datasetname*`,`*chunks*`)`

*Description*
This command writes particle properties, such as the position, in the hdf5-file for all particles at once. The command `particle_property` is a short cut for the various particle properties such as `pos` (position) and `f` (force). These values are stored at the assigned time-step *timestep*. When the values are time-independent set $timestep = -1$. When leaving out the variables *groupname* and *datasetname*, the predefined values of *groupname* and *datasetname* are used in accordance to the H5MD-format.

*Arguments*

- *timestep* Time index in the hdf5-file.
- *groupname* Name of the group, which contains the dataset.
- *datasetname* Name of the dataset, which contains the values.
- *chunks* Size of dataset sub units.

### 10.3.3. `observables`: Write observables

*Python Syntax*

`|h5.write_to_h5.observable(`*timestep*`,`*groupname*`,`*datasetname*`,`*chunks*`)`

*Description*
With this command the various ESPResSo observables can be written. The command `observable` is a short cut for the various observables such as `analyze_energy`.

*Arguments*

- *timestep* Time index in the hdf5-file.
- *groupname* Name of the group, which contains the dataset.
- *datasetname* Name of the dataset, which contains the values.
- *chunks* Size of dataset sub units.

### 10.3.4. `VMD`: Write VMD

*Python Syntax*

`|h5.write_to_h5.VMD(`*datasetname*`,`*value*`)`

*Description*
The molecular visualization program VMD can read hdf5-files, when the H5MD-format is applied. To use VMD only the particle positions are needed. For coloring issues VMD

can use further datasets. These datasets *datasetname* (e.g. 'indexOfSpecies'), with the associated values *value*, are written via the `VMD` command. The datasets are stored in the h5-group "parameters/vmd_structure/". All possible dataset are shown in the example file.

*Arguments*

- *datasetname* Name of the dataset, which contains the values.
- *value* The value to be written.

## 10.4. Reading H5MD-files with Python

To read data from a hdf5-file, again an object of the class `h5md` has to be created and linked to the respective hdf5-file. The `h5md` class contains the sub-class `read_from_h5`, which contains all read commands such as `userdefined`. The sample python script "h5md_ReadFile.py" shows the usage of reading from a hdf5-file.

### 10.4.1. `userdefined`: Read an arbitrary value

*Python Syntax*

`|h5.read_from_h5.userdefined(`*groupname*`,`*datasetname*`,`*datasetindex*`)`

*Description*

The command `userdefined` reads and returns a value from the hdf5-file positioned at *datasetindex*.

*Arguments*

- *groupname* Name of the group, which contains the dataset.
- *datasetname* Name of the dataset, which contains the values.
- *datasetindex* Position of *value* within the dataset.

### 10.4.2. `particle_property`: Read particle properties

*Python Syntax*

`|h5.read_from_h5.particle_property(`*timestep*`,`*groupname*`,`*datasetname*`)`

*Description*

This command reads a particle property at time-step *timestep* from the hdf5-file and passes the values to the respective ESPResSo variables. The command `particle_property` is again a short cut for the various particle property commands such as `pos` and `f`.

*Arguments*

- *timestep* Time index in the hdf5-file.
- *groupname* Name of the group, which contains the dataset.

• *datasetname* Name of the dataset, which contains the values.

## 10.5. `blockfile`: Using the structured file format

ESPResSo uses a standardized ASCII block format to write structured files for analysis or storage. Basically the file consists of blocks in curled braces, which have a single word title and some data. The data itself may consist again of such blocks. An example is:

```
{file {Demonstration of the block format}
{variable epsilon {_dval_ 1} }
{variable p3m_mesh_offset {_dval_ 5.0000000000e-01
   5.0000000000e-01 5.0000000000e-01 } }
{variable node_grid {_ival_ 2 2 2 } }
{end}
```

Whitespace will be ignored within the format (space, tab and return).

The keyword variable should be used to indicate that a variable definition follows in the form *name data*. *data* itself is a block with title `_ival_` or `_dval_` denoting integer resp. double values, which then follow in a whitespace separated list.

Such blocks can be read in and written either from ESPResSo-scripts (see in the following subsections), or from your own C-code using the C-Interface (see section **??**).

### 10.5.1. Writing ESPResSo's global variables

*TCL Syntax*

(1) `blockfile` *channel* `write variable` {*varname1 varname2 ...*}
(2) `blockfile` *channel* `write variable all`

*Description*

Variant (1) writes the global variables *varname1 varname2 ...* (which are known to the `setmd` command (see section 6.1 on page 100) to *channel*. Variant (2) will write all known global variables.

Note, that when the block is read, all variables with names listed in the Tcl variable `blockfile_variable_blacklist` are ignored.

### 10.5.2. Writing Tcl variables

*TCL Syntax*

(1) `blockfile` *channel* `write tclvariable` { *varname1 varname2 ...*}
(2) `blockfile` *channel* `write tclvariable all`
(3) `blockfile` *channel* `write tclvariable reallyall`

*Description*

These commands will write Tcl global variables to *channel*. Global variables are those declared in the top scope of the Tcl script, or those that were explicitly declared

global. When reading the block, all variables with names listed in the Tcl variable `blockfile_tclvariable_blacklist` are ignored.

Variant (1) writes the Tcl global variables *varname1*, *varname2*, ... to *channel*. Variant (2) will write all Tcl variables to the file, with the exception of the internally predefined globals from Tcl (`tcl_version`, `argv`, `argv0`, `argc`, `tcl_interactive`, `auto_oldpath`, `errorCode`, `auto_path`, `errorInfo`, `auto_index`, `env`, `tcl_pkgPath`, `tcl_patchLevel`, `tcl_libPath`, `tcl_library` and `tcl_platform`). Variant (3) will even write those.

### 10.5.3. Writing particles, bonds and interactions

*TCL Syntax*

(1) `blockfile` *channel* `write particles` *what* ( *range* | `all` )
(2) `blockfile` *channel* `write bonds` *range*
(3) `blockfile` *channel* `write interactions`

*Description*

Variant (1) writes particle information in a standardized format to *channel*. *what* can be any list of parameters that can be specified in `part` $part_i d$ `print`, except for `bonds`. Note that `id` and `pos` will automatically be added if missing. *range* is a Tcl list of ranges which particles to write. A range is defined as *start-stop*, where *start* and *stop* are particle identities. *stop* can also be the string "end", denoting the highest used particle identity. Thus `" 0-5 10-end"` are all particles with the exception of particles 6-9. The keyword `all` denotes all known particles, *i.e.* is eqivalent to `"0-end"`).

Variant (2) writes the bond information in a standardized format to *channel*. The involved particles and bond types must exist and be valid.

Variant (3) writes the interactions in a standardized format to *channel*.

### 10.5.4. Writing the random number generator states

*TCL Syntax*

(1) `blockfile` *channel* `write random`
(3) `blockfile` *channel* `write seed`

*Description*

Variant (1) write the full information on the current states of the random number generators (see sections 11.2.1 on page 192) on any node to *channel*. Using this information, it is possible to recover the exact states of the generators.

Variant (3) write only the seed(s) which were used to initialize the random number generators. Note that this information is not sufficient to restore the full state of a random number generator, because the internal state might contain more information.

### 10.5.5. Writing all stored configurations

*TCL Syntax*

`blockfile` *channel* `write configs`

*Description*

This command writes all configurations currently stored for off-line analysis (see section 8.3 on page 150) to *channel*.

### 10.5.6. Writing arbitrary blocks

*TCL Syntax*

(1) `blockfile` *channel* `write start` *tag*
(2) `blockfile` *channel* `write end`
(3) `blockfile` *channel* `write` *tag* [*arg*]...

*Description*

*channel* has to be a Tcl channel. Variant (1) starts a block and gives it the title *tag*, variant (2) ends the block. Between two calls to the command, arbitrary data can be written to the channel. When variant (3) is used, the function `blockfile_write_`*tag* is called with all of the commands arguments. This function should then write the data.

*Example*

```
set file [open "data.dat" w]
blockfile $file write start "mydata"
puts $file "{This is my data!}"
blockfile $file write end
```

will write

```
{mydata {This is my data!}}
```

to the file `data.dat`.

### 10.5.7. Reading blocks

*TCL Syntax*

(1) `blockfile` *channel* `read start`
(2) `blockfile` *channel* `read toend`
(3) `blockfile` *channel* `read auto`
(4) `blockfile` *channel* `read` ( `particles` | `interactions` | `bonds` | `variable` | `seed` | `random` | `bitrandom` | `configs` )

*Description*

Variants (1) and (2) are the low-level block-reading commands. Variant (1) reads the start part of a block and returns the block title, while variant (2) reads the block data and returns it.

Variants (3) and (4) read whole blocks. Variant (3) reads in one block and does the following:

1. if a procedure `blockfile_read_auto_`*tag* exists, this procedure takes over (*tag* is the first expression in the block). For most block types, at least all mentioned above, *i.e.* `particles`, `interactions`, `bonds`, `seed`, `random`, `bitrandom`, `configs`, and `variable`, the corresponding procedure will overwrite the current information with the information from the block.

2. if the procedure does not exist, it returns

   { *usertag rest_of_block* }

3. if the file is at the end, it returns `eof`

Variant (4) checks for a block with tag *block* and then again executes the corresponding `blockfile_read_auto_`*tag*, if it exists.

In the contrary that means that for a new blocktype you will normally implement two procedures:

`|blockfile_write_`*tag channel write tag arg...*
which writes the block including the header and enclosing braces and

`|blockfile_read_auto_`*tag channel read auto*
which reads the block data and the closing brace. The parameters *write*, *read*, *tag* and *auto* are regular parameters which will always have the specified value. They occur just for technical reasons.

In a nutshell: The blockfile command is provided for saving and restoring the current state of ESPResSo, *e.g.* for creating and using checkpoints. Hence you can transfer all accessible information from files to ESPResSo and vice versa.

```
set out [open "|gzip -c - > checkpoint.block.gz" "w"]
blockfile $out write variable all
blockfile $out write interactions
blockfile $out write random
blockfile $out write bitrandom
blockfile $out write particles "id pos type q v f" all
blockfile $out write bonds all
blockfile $out write configs
close $out
```

This example writes all global variables, all interactions, the full current state of the random number generator, all information (*i.e.* id, position, type-number, charge, velocity, forces, bonds) of all particles, and all stored particle configurations to the file `checkpoint.block.gz` which is compressed on-the-fly. If you want to be able to read in the information using ESPResSo, note that interactions must be stored before particles before bonding information, as for the bonds to be set all particles and all interactions must already be known to ESPResSo.

```
set in [open "|gzip -cd checkpoint.block.gz" "r"]
while { [blockfile $in read auto] != "eof" } {}
close $in
```

This is basically all you need to restore the information in the blockfile, overwriting the current settings in ESPResSo.

## 10.6. Writing and reading binary files

Binary files are written using the command

*TCL Syntax*

`|writemd` *channel* [`posx|posy|posz|vx|vy|vz|fx|fy|fz`]`...`

*Description*

This will write out particle data to the Tcl channel *channel* for all particles in binary format. Apart from the mandatory particle id, only limited information can be stored. The coordinates (`posx`, `posy` and `posz`), velocities (`vx`, `vy` and `vz`) and forces (`fx`, `fy` and `fz`). Other information should be stored in a blockfile or reconstructed differently. Note that since both `blockfile` and `writemd` are using a Tcl channel, it is actually possible to mix them, so that you can write a single checkpoint file. However, the `blockfile read auto` mechanism cannot handle the binary section, thus you need to read this section manually. Reading of binary particle data happens through

*TCL Syntax*

`|readmd` *channel*

*Description*

For the exact format of the written binary sequence, see `src/tcl/binary_file_tcl.cpp`.

## 10.7. MPI-IO

When using ESPResSo with MPI, blockfiles and writemd have the disadvantage, that the master node does *all* the output. This is done by sequentially communicating all particle data to the master node. MPI-IO offers the possibility to write out particle data in parallel using binary IO. To output variables and other non-array information, use normal blockfiles (section 10.5).

To dump data using MPI-IO, use the following syntax:

*TCL Syntax*

`|mpiio` *filename* [`read|write`] [`pos|v|bond|type`]`...`

*Description*

This command writes data to several files using *filename* as common filename prefix. Beware, that *filename* must not be a Tcl channel but a string which must not contain colons. The data can be positions (`pos`), velocities (`v`), particle types (`type`) and particle

bonds (`bond`) or any combination of these. The particle ids are always dumped. For safety reasons, MPI-IO will not overwrite existing files, so if the command fails and prints `MPI_ERR_IO` make sure the files are non-existent.

The files produced by this command are (assumed *filename* is "1"):

**1.head** Internal information (Dumped fields, bond partner num); always produced

**1.pref** Internal information (Exscan results of nlocalparts); always produced

**1.ids** Particle ids; always produced

**1.type** Particle types; optional

**1.pos** Particle positions; optional

**1.vel** Particle velocities; optional

**1.bond** Bond information; optional

**1.boff** Internal bond prefix information; optional, necessary to read 1.bond

Currently, these files have to be read by exactly the same number of MPI processes that was used to dump them, otherwise an error is signalled. Also, the same type of machine (endianess, byte order) has to be used. Otherwise only garbage will be read. The read command replaces the particles, i.e. all previous existent particles will be *deleted*.

There is a python script (`tools/mpiio2blockfile.py`) which converts MPI-IO snapshots to regular ESPResSo blockfiles.

## 10.8. Writing VTF files

The formats VTF (**V**TF **T**rajectory **F**ormat), VSF (**V**TF **S**tructure **F**ormat) and VCF (**V**TF **C**oordinate **F**ormat) are formats for the visualization software VMD[33][1]. They are intended to be human-readable and easy to produce automatically and modify.

The format distinguishes between *structure blocks* that contain the topological information of the system (*i.e.* the system size, particle names, types, radii and bonding information, amongst others), while *coordinate blocks* (a.k.a. as *timestep blocks*) contain the coordinates for the particles at a single timestep. For a visualization with VMD, one structure block and at least one coordinate block is required.

Files in the VSF format contain a single structure block, files in the VCF format contain at least one coordinate block, while files in the VTF format contain a single structure block first and an arbitrary number of coordinate blocks afterwards, thus allowing to store all information for a whole simulation in a single file. For more details on the format, refer to the homepage of the format[2].

[1] `http://www.ks.uiuc.edu/Research/vmd/`

[2] `https://github.com/olenz/vtfplugin/wiki/VTF-format`

Creating files in these formats from within ESPResSo is supported by the commands `writevsf` and `writevcf`, that write a structure respectively a coordinate block to the given Tcl channel. To create a VTF file, first use `writevsf` at the beginning of the simulation, and then `writevcf` after each timestep to generate a trajectory of the whole simulation.

The structure definitions in the VTF/VSF formats are incremental, *i.e.* a user can easily add further structure lines to the VTF/VSF file after a structure block has been written to specify further particle properties for visualization.

Note that the ids of the particles in ESPResSo and VMD may differ. VMD requires the particle ids to be enumerated continuously without any holes, while this is not required in ESPResSo. When using `writevsf` and `writevcf`, the ESPResSo particle ids are automatically translated into VMD particle ids. The function `vtfpid` allows the user to get the VMD particle id for a given ESPResSo particle id.

Also note, that these formats can not be used to write trajectories where the number of particles or their types varies between the timesteps. This is a restriction of VMD itself, not of the format.

### 10.8.1. `writevsf`: Writing the topology

*TCL Syntax*

`writevsf` *channelId* [( `short` | `verbose` )] [`radius` ( *radii* | `auto` )] [`typedesc` *typedesc*]

*Description*

Writes a structure block describing the system's structure to the channel given by *channelId*. *channelId* must be an identifier for an open channel such as the return value of an invocation of `open`. The output of this command can be used for a standalone VSF file, or at the beginning of a VTF file that contains a trajectory of a whole simulation.

*Arguments*

- [( `short` | `verbose` )] Specify, whether the output is in a human-readable, but somewhat longer format (`verbose`), or in a more compact form (`short`). The default is `verbose`.
- [`radius` ( *radii* | `auto` )] Specify the VDW radii of the atoms. *radii* is either `auto`, or a Tcl-list describing the radii of the different particle types. When the keyword `auto` is used and a Lennard-Jones interaction between two particles of the given type is defined, the radius is set to be $\frac{\sigma_{LJ}}{2}$ plus the LJ shift. Otherwise, the radius 0.5 is substituted. The default is `auto`.

  *Example*

  ```
  writevsf $file radius {0 2.0 1 auto 2 1.0}
  ```

- [`typedesc` *typedesc*] *typedesc* is a Tcl-list giving additional VTF atom-keywords to specify additional VMD characteristics of the atoms of the given type. If no

description is given for a certain particle type, it defaults to `name` *`name`* `type` *`type`*, where *name* is an atom name and *type* is the type id.

*Example*

```
writevsf $file typedesc {0 "name colloid" 1 "name pe"}
```

### 10.8.2. `writevcf`: Writing the coordinates

*TCL Syntax*

```
writevcf channelId [( short | verbose )] [( folded | absolute )]
      [pids ( pids | all )] [userdata userdata]
```

*Description*

Writes a coordinate (or timestep) block that contains all coordinates of the system's particles to the channel given by *channelId*. *channelId* must be an identifier for an open channel such as the return value of an invocation of `open`.

*Arguments*

- [( `short` | `verbose` )] Specify, whether the output is in a human-readable, but somewhat longer format (`verbose`), or in a more compact form (`short`). The default is `verbose`.
- [( `folded` | `absolute` )] Specify whether the particle positions are written in absolute coordinates (`absolute`) or folded into the central image of a periodic system (`folded`). The default is `absolute`.
- [`pids` ( *pids* | `all` )] Specify the coordinates of which particles should be written. If `all` is used, all coordinates will be written (in the ordered timestep format). Otherwise, *pids* has to be a Tcl-list specifying the pids of the particles. The default is `all`.

  *Example*

  ```
  pids {0 23 42}
  ```

- [`userdata` *userdata*] Specify arbitrary user data for the particles. *userdata* has to be a Tcl list containing the user data for every particle. The user data is appended to the coordinate line and can be read into VMD via the VMD plugin `VTFTools`. The default is to provide no userdata.

  *Example*

  ```
  userdata {"red" "blue" "green"}
  ```

### 10.8.3. `vtfpid`: Translating ESPResSo particles ids to VMD particle ids

*TCL Syntax*

```
vtfpid pid
```

*Description*

If *pid* is the id of a particle as used in ESPResSo, this command returns the atom id used in the VTF, VSF or VCF formats.

## 10.9. `writevtk`: Particle Visualization in paraview

This feature allows to export the particle positions in a paraview [3] compatible VTK file. Paraview is a powerful and easy to use open-source visualization program for scientific data. Since ESPResSo can export the lattice-Boltzmann velocity field 12.8 in the VTK format as well and paraview allows to visualize particles with glyphs and vector fields with stream lines, glyphs, contourplots, etc., one can use it so completely visualize a coupled lattice-Boltzmann MD simulation. It can also create videos without much effort if one exports data of individual time steps into separate files with filenames including a running index (`data_0.vtk`, `data_1.vtk`, ...).

*TCL Syntax*

| `writevtk` *filename* [( `all` | *types* )]

*Description*

*Arguments*

- *filename* Name of the file to export the particle positions into.
- [( `all` | *type* )] Specifies a list of particle types which should be exported. The default is `all`. Alternatively, a list of type number can be given. Exporting the positions of all particles but in separate files might make sense if one wants to distinguish the different particle types in the visualization (i.e. by color or size). To export a type `1` use something along `writevtk "test.tcl" "1"`. To export types `1`, `5`, `7`, which are not to be distinguished in the visualization, use `writevtk "test.tcl" "7 1 5"`. The order in the list is arbitrary, but duplicates are *not* ignored!

## 10.10. Reading and Writing PDB/PSF files

The PDB (Brookhaven Protein DataBase) format is a widely used format for describing atomistic configurations. PSF is a format that is used to describe the topology of a PDB file.

When visualizing your system with VMD, it is recommended to use the VTF format instead (see section 10.8), as it was specifically designed for visualizations with VMD. In contrast to the PDB/PSF formats, in VTF files it is possible to specify the VDW radii of the particles, to have a varying simulation box size, etc.

[3] `http://www.paraview.org/`

### 10.10.1. `writepsf`: Writing the topology

*TCL Syntax*

`writepsf` *file* [`-molecule`] $N_P$ $MPC$ $N_C I$ $N_p S$ $N_n S$

*Description*

Writes the current topology to the file *file* (here, *file* is not a channel, since additional information cannot be written anyway). $N_P$, $MPC$ and so on are parameters describing a system consisting of equally long charged polymers, counterions and salt. This information is used to set the residue name and can be used to color the atoms in VMD. If you specify `-molecule`, the residue name is taken from the molecule identity of the particle. Of course different kinds of topologies can also be handled by modified versions of `writepsf`.

### 10.10.2. `writepdb`: Writing the coordinates

*TCL Syntax*

(1) `writepdb` *file*
(2) `writepdbfoldchains` *file* *chain_start* *n_chains* *chain_length* *box_l*
(3) `writepdbfoldtopo` *file* *shift*

*Description*

Variant (1) writes the corresponding particle data.

Variant (2) writes folded particle data where the folding is performed on chain centers of mass rather than single particles. In order to fold in this way the chain topology and box length must be specified. Note that this method is outdated. Use variant (3) instead.

Variant (3) writes folded particle data where the folding is performed on chain centers of mass rather than single particles. This method uses the internal box length and topology information from espresso. If you wish to shift particles prior to folding then supply the optional shift information. *shift* should be a three member tcl list consisting of x, y, and z shifts respectively and each number should be a floating point (ie with decimal point).

### 10.10.3. `readpdb`: Reading the coordinates and interactions

*TCL Syntax*

`readpdb pdb_file` *pdbfile* `type` *type* `first_id` *firstid*
[ `itp_file` *itpfile* `first_type` *fisttype*]
[`lj_with` *othertype* *epsilon* *sigma* [1]] [`lj_rel_cutoff` *cutoff* [1] ]
[`fit_to_box`]

Required features: [1] `LENNARD_JONES`

*Description*

Reads the positions and possibly charges, types and Lennard-Jones interactions from the file *pdbfile* and a corresponding Gromacs topology file *itpfile*. The topology file must contain the `atoms` and `atomtypes` sections, it may be necessary to use the Gromacs preprocessor to obtain a complete file from a system configuration and a force field.

Any offset of the particle positions if removed, such that the lower left corner bounding box of the particles is in the origin. If `fit_to_box` is given, the box size if increased to hold the particles if necessary. If it is not set and the particles do not fit into the box, the behavior is undefined.

*type* sets the particle type for the added particles. If there is a topology file give that contains a types for the particles, the particles get types by the order in the topology file plus *firstype*. If the corresponding type in the topology file has a charge, it is used, otherwise the particle charge defaults to zero.

The particles get consecutive id's in the order of the pdb file, starting at *firstid*. Please be aware that existing particles get overwritten by values from the file.

The `lj_with` section produces Lennard-Jones interactions between the type *othertype* and the types defined by the topology file. The interaction parameters are calculated as $\epsilon_{\text{othertype},j} = \sqrt{\epsilon_{\text{othertype}}\epsilon_j}$ and $\sigma_{\text{othertype},j} = \frac{1}{2}(\sigma_{\text{othertype}} + \sigma_j)$, where $j$ runs over the atomtypes defined in the topology file. This corresponds to the combination rule 2 of Gromacs. There may be multiple such sections. The cutoff is determined by *cutoff* as cutoff $\times \sigma_{ij}$ in a relative fashion. The potential is shifted so that it vanishes at the cutoff. The command returns the number of particles that were successfully added.

Reading bonded interactions and dihedrals is currently not supported.

## 10.11. Online-visualisation with VMD

IMD (Interactive Molecular Dynamics) is the protocol that VMD uses to communicate with a simulation. Tcl_md implements this protocol to allow online visual analysis of running simulations.

In IMD, the simulation acts as a data server. That means that a simulation can provide the possibility of connecting VMD, but VMD need not be connected all the time. You can watch the simulation just from time to time.

In the following the setup and usage of IMD is described.

### 10.11.1. `imd`: Using IMD in the script

*TCL Syntax*

(1) `imd connect` [*port*]
(2) `imd positions` [( `-unfolded` |`-fold_chains` )]
(3) `imd listen` *seconds*
(4) `imd disconnect`

*Description*

In your simulation, the IMD connection is setup up using variant (1), where *port* is an arbitrary port number (which usually has to be between 1024 and 65000). By default, ESPResSo will try to open port 10000, but the port may be in use already by another ESPResSo simulation. In that case it is a good idea to just try another port.

While the simulation is running, variant (2) can be used to transfer the current coordinates to VMD, if it is connected. If not, nothing happens and the command just consumes a small amount of CPU time. Note, that before you can transfer coordinates to VMD, VMD needs to be aware of the structure of the system. For that, you first need to load a corresponding structure file (PSF or VSF) into VMD. Also note, that the command `prepare_vmd_connection` (see section 10.11.3) can be used to automatically set up the VMD connection and transfer the structure file.

By specifying `-unfolded`, the unfolded coordinates of the particles will transferred, while `-fold_chains` will fold chains according to their centers of mass and retains bonding connectivity. Note that this requires the chain structure to be specified first using the analyze command.

Variant (3) can be used to let the simulation wait for *seconds* seconds or until IMD has connected, before the script is continued. This is normally only useful in demo scripts, if you want to see all frames of the simulation.

Variant (4) will terminate the IMD session. This is normally not only nice but also the operating system will not free the port for some time, so that without disconnecting for some 10 seconds you will not be able to reuse the port.

### 10.11.2. Using IMD in VMD

The PDB/PSF files created by ESPResSo via the command `writepsf` and `writepdb` can be loaded into VMD. This should bring up an initial configuration.

Then you can use the VMD console to execute the command

`imd connect` *host port*

where *host* is the host running the simulation and *port* is the port it listens to. Note that VMD crashes, if you do that without loading a structure file before. For more information on how to use VMD to extract more information or hide parts of configuration, see the VMD Quick Help.

### 10.11.3. Automatically setting up a VMD connection

*TCL Syntax*

(1) `prepare_vmd_connection` *filename* [`start`] [`wait` *wait*] [`localhost`] [`constraints`] `...`

(2) `prepare_vmd_connection` [*filename* [*wait* [*start* [*constraints*]]]]

*Description*

To reduce the effort involved in setting up the IMD connection, starting VMD and loading the structure file, ESPResSo provides the command `prepare_vmd_connection`. It

writes out the required vsf structure description file to *filename*.vsf (default for *filename* is `vmd`), doing some nice stuff such as coloring the molecules, bonds and counterions appropriately, rotating your viewpoint, and connecting your system to the visualization server.

If the option [constraints] is given, then the command will create graphics primitives in VMD that represent some of the spatial constraints (sphere, rhomboid and cylinder at present).

If [start] is given, the command will automatically try to start VMD and connect it to the ESPResSo simulation. Otherwise it only writes the VMD setup script *filename*`.vmd_start.script`. You can use this script later to connect to the ESPResSo simulation by running either

```
vmd -e vmd_start.script
```

or by running

```
source "vmd_start.script"
```

at VMD's Tcl console. If you choose to not start VMD automatically, `prepare_vmd_connection` puts the hostname into the VMD script, so that you can start it from any computer. However, some more recent Linux distributions block any incoming transfer even from the computer itself, if it does not come from localhost. If you encounter problems to connect to VMD on the very same computer, try the [localhost] option, which will enforce to use the hostname `localhost`. Note that the [start] option implies the [localhost] option, since VMD is necessarily started from the same computer.

If the option [wait] is provided, then the command waits for at most *wait* seconds for VMD to connect. Since VMD usually takes a while to start, it is usually a good idea to combine the [start] option with a waiting time of 100, so a bit less than a minute.

All remaining parameters are passed to the `writevsf` that is used to setup the system, so that you can specify the sizes of particles etc.

`prepare_vmd_connection` also supports an older, deprecated syntax (variant 2) with limited functionality. This syntax uses fixed position parameters and boolean values for [start] and [constraints], as described above.

## 10.12. Error handling

Errors in the parameters are detected as early as possible, and hopefully self-explanatory error messages returned without any changes to the data in the internal data of ESPResSo. This include errors such as setting nonexistent properties of particles or simply misspelled commands. These errors are returned as standard Tcl errors and can be caught on the Tcl level via

```
catch {script} err
```

When run noninteractively, Tcl will return a nice stack backtrace which allows to quickly find the line causing the error.

However, some errors can only be detected after changing the internal structures, so that ESPResSo is left in a state such that integration is not possible without massive fixes by the users. Especially errors occuring on nodes other than the primary node fall under this condition, for example a broken bond or illegal parameter combinations.

I do not understand this. How does the error look?

For error conditions such as the examples given above, a Tcl error message of the form

*tcl_error* `background 0` *error_a error_b* `1` *error_c*

is returned. Following possibly a normal Tcl error message, after the background keyword all severe errors are listed node by node, preceeded by the node number. A special error is `<consent>`, which means that one of the slave nodes found exactly the same errors as the master node. This happens mainly during the initialization of the integrate, *e.g.* if the time step is not set. In this case the error message will be

```
background_errors 0 {time_step not set} 1 <consent>
```

In each case, the current action was not fulfilled, and possibly other parts of the internal data also had to be changed to allow ESPResSo to continue, so you should really know what you do if you try and catch these errors.

# 11. Auxilliary commands

Missing commands: Probably all from `scripts/auxiliary.tc`

## 11.1. Finding particles and bonds

### 11.1.1. `countBonds`

*TCL Syntax*

| `countBonds` *particle_list*

*Description*

Returns a Tcl-list of the complete topology described by *particle_list*, which must have the same format as the output of the command `part` (see section 4.1 on page 32).

The output list contains only the particle id and the corresponding bonding information, thus it looks like *e.g.*

```
{106 {0 107}} {107 {0 106} {0 108}} {108 {0 107} {0 109}} ...
{210 {0 209} {0 211}} {211 {0 210}} 212 213 ...
```

for a single chain of 106 monomers between particle 106 and 211, with additional loose particles 212, 213, ... (*e.g.* counter-ions). Note, that the `part` command stores any bonds only with the particle of lower particle number, which is why `[part 109]` would only return `... bonds 0 110`, therefore not revealing the bond between particle 109 and (the preceding) particle 108, while `countBonds` would return all bonds particle 109 participates in.

### 11.1.2. `findPropPos`

*TCL Syntax*

| `findPropPos` *particle_property_list property*

*Description*

Returns the index of *property* within *particle_property_list*, which is expected to have the same format as `[part` *particle_id*`]`. If *property* is not found, `-1` is returned.

This function is useful to access certain properties of particles without hard-wiring their index-position, which might change in future releases of `part`.

*Example*

```
[lindex [part $i] [findPropPos [part $i] type]]
```

This returns the particle type id of particle $i$ without fixing where exactly that information has to be in the output of `[part $i]`.

### 11.1.3. `findBondPos`

*TCL Syntax*

| `findBondPos` *particle$_p$roperty$_l$ist*

*Description*
Returns the index of the bonds within *particle$_p$roperty$_l$ist*, which is expected to have the same format as `[part` *particle_number*`]`; hence its output is the same as `[findPropPos` *particle$_p$roperty$_l$ist* `bonds]`. If the particle does not have any bonds, `-1` is returned.

### 11.1.4. `timeStamp`

*TCL Syntax*

| `timeStamp` *path prefix postfix suffix*

*Description*
Modifies the filename contained within *path* to be preceded by a *prefix* and having *postfix* before the *suffix*; *e.g.*

```
timeStamp ./scripts/config.gz DH863 001 gz
```

returns `./scripts/DH863_config001.gz`. If *postfix* is $-1$, the current date is used in the format `%y%m%d`. This would results in `./scripts/DH863_config021022.gz` on October 22nd, 2002.

## 11.2. Additional Tcl math-functions

The following procedures are found in scripts/ABHmath.tcl.

- CONSTANTS
  - `PI`

    returns $\pi$ with 16 digits precision.
  - `KBOLTZ`

    Returns Boltzmann constant in Joule/Kelvin
  - `ECHARGE`

    Returns elementary charge in Coulomb
  - `NAVOGADRO`

    Returns Avogadro number
  - `SPEEDOFLIGHT`

Returns speed of light in meter/second

- `EPSILON0`

  Returns dielectric constant of vaccum in Coulomb$\hat{2}$/(Joule meter)

- `ATOMICMASS`

  Returns the atomic mass unit u in kilogramms

- MATHEMATICAL FUNCTIONS
  - `sqr <arg>`

    returns the square of *arg*.

  - `min <arg1> <arg2>`

    returns the minimum of *arg1* and *arg2*.

  - `max <arg1> <arg2>`

    returns the maximum of *arg1* and *arg2*.

  - `sign <arg>`

    returns the signum-function of *arg*, namely +1 for *arg* $> 0$, -1 for $< 0$, and =0 otherwise.

- RANDOM FUNCTIONS
  - `gauss_random`

    returns random numbers which have a Gaussian distribution

  - `dist_random <dist> [max]`

    returns random numbers in the interval $[0, 1]$ which have a distribution according to the distribution function p(x) *dist* which has to be given as a tcl list containing equally spaced values of p(x). If p(x) contains values larger than 1 (default value of max) the maximum or any number larger than that has to be given *max*. This routine basically takes the function p(x) and places it into a rectangular area ([0,1],[0,max]). Then it uses to random numbers to specify a point in this area and checks wether it resides in the area under p(x). Attention: Since this is written in tcl it is probably not the fastest way to do this!

  - `vec_random [len]`

    returns a random vector of length *len* (uniform distribution on a sphere) This is done by chosing 3 uniformly distributed random numbers $[-1, 1]$ If the length of the resulting vector is <= 1.0 the vector is taken and normalized to the desired length, otherwise the procedure is repeated until succes. On average the procedure needs 5.739 random numbers per vector. (This is probably not the most efficient way, but it works!) Ask your favorit mathematician for a proof!

- `phivec_random <v> <phi> [len]`

  return a random vector at angle *phi* with *v* and length *len*

- PARTICLE OPERATIONS

  Operations involving particle positions. The parameters *pi* can either denote the particle identity (then the particle position is extracted with the The part command command) or the particle position directly When the optional *box* parameter for minimum image conventions is omited the functions use the the `setmd box_l` command.

  - `bond_vec <p1> <p2>`

    Calculate bond vector pointing from particles *p2* to *p1* return = (*p1*.pos - *p2*.pos)

  - `bond_vec_min <p1> <p2> [box]`

    Calculate bond vector pointing from particles *p2* to *p1* return = MinimumImage(*p1*.pos - *p2*.pos)

  - `bond_length <p1> <p2>`

    Calculate bond length between particles *p1* and *p2*

  - `bond_length_min <p1> <p2> [box]`

    Calculate minimum image bond length between particles *p1* and *p2*

  - `bond_angle <p1> <p2> <p3> [type]`

    Calculate bond angle between particles *p1*, *p2* and *p3*. If *type* is "r" the return value is in radiant. If it is "d" the return value is in degree. The default for *type* is "r".

  - `bond_dihedral <p1> <p2> <p3> <p4> [type]`

    Calculate bond dihedral between particles *p1*, *p2*, *p3* and *p4* If *type* is "r" the return value is in radiant. If it is "d" the return value is in degree The default for *type* is "r".

  - `part_at_dist <p> <dist>`

    return position of a new particle at distance *dist* from *p* with random orientation

  - `part_at_angle <p1> <p2> <phi> [len]`

    return position of a new particle at distance *len* (default=1.0) from *p2* which builds a bond angle *phi* for (*p1*, *p2*, p-new)

  - `part_at_dihedral <p1> <p2> <p3> <theta> [phi] [len]`

    return position of a new particle at distance *len* (default=1.0) from *p3* which builds a bond angle *phi* (default=random) for (*p2*, *p3*, p-new) and a dihedral angle *theta* for (*p1*, *p2*, *p3*, p-new)

- INTERACTION RELATED

  Help functions related to interactions implemented in ESPResSo.

  - `calc_lj_shift <lj_sigma> <lj_cutoff>`

    returns the value needed to shift the Lennard Jones potential to zero at the cutoff.

- VECTOR OPERATIONS

  A vector $v$ is a tcl list of numbers with an arbitrary length Some functions are provided only for three dimensional vectors. corresponding functions contain 3d at the end of the name.

  - `veclen <v>`

    return the length of a vector

  - `veclensqr <v>`

    return the length of a vector squared

  - `vecadd <a> <b>`

    add vector $a$ to vector $b$: return = ($a$+$b$)

  - `vecsub <a> <b>`

    subtract vector $b$ from vector $a$: return = ($a$-$b$)

  - `vecscale <s> <v>`

    scale vector $v$ with factor $s$: return = ($s$*$v$)

  - `vecdot_product <a> <b>`

    calculate dot product of vectors $a$ and $b$: return = ($a$.$b$)

  - `veccross_product3d <a> <b>`

    calculate the cross product of vectors $a$ and $b$: return = ($a$ x $b$)

  - `vecnorm <v> [len]`

    normalize a vector to length *len* (default 1.0)

  - `unitvec <p1> <p2>`

    return unit vector pointing from position *p1* to position *p2*

  - `orthovec3d <v> [len]`

    return orthogonal vector to $v$ with length *len* (default 1.0) This vector does not have a random orientation in the plane perpendicular to $v$

  - `create_dihedral_vec <v1> <v2> <theta> [phi] [len]`

    create last vector of a dihedral (*v1*, *v2*, res) with dihedral angle *theta* and bond angle (*v2*, res) *phi* and length *len* (default 1.0). If *phi* is ommited or set to rnd then *phi* is assigned a random value between 0 and 2 Pi.

- TCL LIST OPERATIONS
  - `average <list>`

    Returns the avarage of the provided *list*
  - `list_add_value <list> <val>`

    Add *val* to each element of *list*
  - `flatten <list>`

    flattens a nested *list*
  - `list_contains <list> <val>`

    Checks wether *list* contains *val*. returns the number of occurences of *val* in *list*.
- REGRESSION
  - `LinRegression <l>`

    where $l$ is a listof pairs of points `{ {$x1 $y1} {$x2 $y2} ...}` . `LinRegression` returns the least-square linear fit $ax + b$ and the standard errors $\sigma_a$ and $\sigma_b$.
  - `LinRegressionWithSigma <l>`

    where $l$ is a list of lists of points in the form `{{$x1 $y1 $s1} {$x2 $y2 $s2} ...}` where `s` is the standard deviation of `y`. `LinRegressionWithSigma` returns the least-square linear fit $ax+b$, the standard errors $\sigma_a$ and $\sigma_b$, covariance $\mathrm{cov}(a, b)$ and $\chi$.

### 11.2.1. `t_random`

- Without further arguments (tcl only),

  `t_random`

  returns a random double between 0 and 1 using the standard C++ Mersenne twister random number generator. For python please use the numpy random number generator.
- `t_random int <n> (tcl only)`

  returns a random integer between 0 and n-1. For python please use the numpy random number generator.
- To obtain the seeds of the random number generator (C++ Mersenne Twister) which is used in the C core of Espresso use the command

  `t_random seed`

```
system = espressomd.System()
system.seed
```

which returns a list with the seeds of the random number generators on each of the 'n_nodes' nodes if they were set by the user, while

```
t_random seed <seed(0)> ... <seed(n_nodes-1)>
```

```
system = espressomd.System()
system.seed=[seed(0), ... seed(n_nodes-1)]
```

sets those seeds to the new values respectively, re-initialising the random number generators on each node. Note that this is automatically done on invoking Espresso, however due to that your simulation will always start with the same random sequence on any node *unless you seed your random number generator* at the beginning of the simulation.

## 11.3. Checking for features of ESPResSo

In an ESPResSo-Tcl-script, you can get information whether or not one or some of the features are compiled into the current program with help of the following Tcl-commands:

- `code_info`

  provides information on the version, compilation status and the debug status of the used code. It is highly recommended to store this information with your simulation data in order to maintain the reproducibility of your results. Exemplaric output:

  ```
  ESPRESSO: v1.5.Beta (Neelix), Last Change: 23.01.2004
  { Compilation status { PARTIAL_PERIODIC } { ELECTROSTATICS }
  { EXTERNAL_FORCES } { CONSTRAINTS } { TABULATED }
  { LENNARD_JONES } { BOND_ANGLE_COSINE } }
  { Debug status { MPI_CORE FORCE_CORE } }
  ```

- `has_feature <feature> ...`

  tests, if *feature* is compiled into the ESPResSo kernel. A list of possible features and their names can be found here.

- `require_feature <feature> ...`

  tests, if *feature* is feature is compiled into the ESPResSo kernel, will exit the script if it isn't and return the error code 42. A list of possible features and their names can be found here.

# 12. Lattice-Boltzmann

For an implicit treatment of a solvent, ESPResSo allows to couple the molecular dynamics simulation to a Lattice-Boltzmann fluid. The Lattice- Boltzmann-Method (LBM) is a fast, lattice based method that, in its "pure" form, allows to calculate fluid flow in different boundary conditions of arbitrarily complex geometries. Coupled to molecular dynamics, it allows for the computationally efficient inclusion of hydrodynamic interactions into the simulation. The implementation of boundary conditions for the LBM is a difficult task, a lot of research is still being conducted on this topic. The focus of the ESPResSo implementation of the LBM is, of course, the coupling to MD and therefore available geometries and boundary conditions are somewhat limited in comparison to "pure" codes.

Here we restrict the documentation to the interface. For a more detailed description of the method, please refer to the literature.

## 12.1. Setting up a LB fluid

Please cite [7] (BibTeX-key `espresso2` in file `doc/ug/citations.bib`) if you use the LB fluid and [49] (BibTeX-key `lbgpu` in file `doc/ug/citations.bib`) if you use the GPU implementation.

*TCL Syntax*

`lbfluid` [`gpu`] [2] [`agrid` *agrid*] [1 or 2] [`dens` *density* ] [1 or 2 or 3]
[`visc` *viscosity*] [1 or 2 or 3] [`tau` *lb_timestep*] [1 or 2]
[`bulk_viscosity` *bulk_viscosity*] [1 or 2 or 3] [`ext_force` $f_x$ $f_y$ $f_z$] [1 or 2 or 3]
[`friction` *gamma* ] [1 or 2 or 3] [`couple` *2pt/3pt* ] [2]
[`gamma_odd` *gamma_odd*] [1 or 2 or 3] [`gamma_even` *gamma_even*] [1 or 2 or 3]
[`mobility`] *mobilities* [3] [`sc_coupling`] *coupling_constants* [3]

Required features: [1] `LB` [2] `LB_GPU` [3] `SHANCHEN`

*Description*

The `lbfluid` command initializes the fluid with a given set of parameters. It is also possible to change parameters on the fly, but this will only rarely be done in practice. Before being able to use the LBM, it is necessary to set up a box of a desired size. The parameter `agrid` is used to set the lattice constant of the fluid, so the size of the box in every direction must be a multiple of *agrid*.

In ESPResSo the LB scheme and the MD scheme are not synchronized: In one LB time step typically several MD steps are performed. This allows to speed up the simulations

and is adjusted with the parameter *tau*, the LB timestep. The parameters *dens* and *visc* set up the density and (kinematic) viscosity of the LB fluid in (usual) MD units. Internally the LB implementation works with a different set of units: all lengths are expressed in *agrid*, all times in *tau* and so on. Therefore changing *agrid* and *tau*, might change the behaviour of the LB fluid, *e.g.* at boundaries, due to characteristics of the LBM itself. It should also be noted that the LB nodes are located at 0.5, 1.5, 2.5, etc (in terms of *agrid*). This has important implications for the location of hydrodynamic boundaries which are generally considered to be halfway between two nodes to first order. Currently it is not possible to precisely give a parameter set where reliable results are expected, but we are currently performing a study on that. Therefore the LBM should *not be used as a black box*, but only after a careful check of all parameters that were applied.

The parameter `ext_force` allows to apply an external body force density that is homogeneous over the fluid. It is again to be given in MD units. The parameter `bulk_viscosity` allows to tune the bulk viscosity of the fluid and is given in MD units. In the limit of low Mach (often also low Reynolds) number the results should be independent of the bulk viscosity up to a scaling factor. It is however known that the values of the viscosity does affect the quality of the implemented link-bounce-back method. `gamma_odd` and `gamma_even` are the relaxation parameters for the kinetic modes. Due to their somewhat obscure nature they are to be given directly in LB units.

Before running a simulation at least the following parameters must be set up: `agrid`, `dens`, `visc`, `tau`, `friction`. For the other parameters, the following are taken: *bulk_viscosity*=0, *gamma_odd*=0, *gamma_even*=0, $f_x = f_y = f_z = 0$.

If the feature `SHANCHEN` is activated, the Lattice Boltzmann code (so far GPU version only) is extended to a two-component Shan-Chen (SC) method. The `lbfluid` command requires in this case to supply two values, for the respective fluid components, to each of the options `dens`, `visc`, `bulk_viscosity`, `friction`, `gamma_odd` and `gamma_even`, when they are used, otherwise they are set to the default values. The three elements of the coupling matrix can be supplied with the option `sc_coupling`, and the mobility coefficient can be specified with the option `mobility`. By default no copuling is activated, and the relaxation parameter associated to the mobility is zero, corresponding to an infinite value for `mobility`. Additional details are given in 12.3 and 12.4.

*TCL Syntax*

`lbfluid print_interpolated_velocity` $x$ $y$ $z$

*Description*

This variant returns the velocity at point in countinous space. This can make it easier to calculate flow profiles independent of the lattice constant.

*TCL Syntax*

`lbfluid save_ascii_checkpoint` *filename*
`lbfluid save_binary_checkpoint` *filename*
`lbfluid load_ascii_checkpoint` *filename*
`lbfluid load_binary_checkpoint` *filename*

*Description*

The first two save commands save all of the LB fluid nodes' populations to *filename* in ascii or binary format respectively. The two load commands load the populations from *filename*. This is useful for restarting a simulation either on the same machine or a different machine. Some care should be taken when using the binary format as the format of doubles can depend on both the computer being used as well as the compiler. One thing that one needs to be aware of is that loading the checkpoint also requires the used to reuse the old forces. This is necessary since the coupling force between the paricles and the fluid has already been applied to the fluid. Failing to reuse the old forces breaks momentum conservation, which is in general a problem. It is particularly problematic for bulk simulations as the system as a whole acquires a drift of the center of mass, causing errors in the calculation of velocities and diffusion coefficients. The correct way to restart an LB simulation is to first load in the particles with the correct forces, and use "integrate *steps* reuse_forces" upon the first call to integrate. This causes the old forces to be reused and thus conserves momentum.

## 12.2. LB as a thermostat

*TCL Syntax*

`thermostat lb` [1 or 2 or 3] $T$

Required features: [1] `LB` [2] `LB_GPU` [3] `SHANCHEN`

*Description*

The LBM implementation in ESPResSo uses Ahlrichs and Dünweg's point coupling method to couple MD particles the LB fluid. This coupling consists in a frictional force and a random force:

$$\vec{F} = -\gamma\left(\vec{v} - \vec{u}\right) + \vec{F}_R.$$

The momentum acquired by the particles is then transferred back to the fluid using a linear interpolation scheme, to preserve total momentum. In the GPU implementation the force can alternatively be interpolated using a three point scheme which couples the particles to the nearest 27 LB nodes. This can be called using "lbfluid `couple` 3pt" and is described in Dünweg and Ladd by equation 301[22]. Note that the three point coupling scheme is incompatible with the Shan Chen Lattice Boltmann. The frictional force tends to decrease the relative velocity between the fluid and the particle whereas the random forces are chosen so large that the average kinetic energy per particle corresponds to the given temperature, according to a fluctuation dissipation theorem. No other thermostatting mechanism is necessary then. Please switch off any other thermostat before starting the LB thermostatting mechanism.

The LBM implementation provides a fully thermalized LB fluid, *i.e.* all nonconserved modes, including the pressure tensor, fluctuate correctly according to the given temperature and the relaxation parameters. All fluctuations can be switched off by setting the temperature to 0.

Regarind the unit of the temperature, please refer to Section 1.4.

## 12.3. The Shan Chen bicomponent fluid

Please cite [53] (BibTeX-key `sega13c` in file `doc/ug/citations.bib`) if you use the Shan Chen implementation described below.

The Lattice Boltzmann variant of Shan and Chan[54] is widely used as it is simple and yet very effective in reproducing the most important traits of multicomponent or multiphase fluids. The version of the Shan-Chen method implemented in ESPResSo is an extension to bi-component fluids of the multi-relaxation-times Lattice Boltzmann with fluctuations applied to all modes, that is already present in ESPResSo. It features, in addition, coupling with particles[53] and component-dependent particle interactions (see sections 12.4 and12.5).

The Shan-Chen fluid is set up using the `lbfluid` command, supplying two values (one per component) to the `dens` option. Optionally, two values can be set for each of the usual transport coefficients (shear and bulk viscosity), and for the ghost modes. It is possible to set a relaxation time also for the momentum modes, since they are not conserved quantities in the Shan-Chen method, by using the option `mobility`. The mobility transport coefficient expresses the propensity of the two components to mutually diffuse, and, differently from other transport coefficients, only one value is needed, as it carachterizes the mixture as a whole. When thermal fluctuations are switched on, a random noise is added, in addition, also to the momentum modes. Differently from the other modes, a correlated noise is added to the momentum ones, in order to preserve the *total* momentum.

The fluctuating hydrodynamic equations that are simulated using the Shan-Chen approach are

$$\rho\left(\frac{\partial}{\partial t}\vec{u}+(\vec{u}\cdot\vec{\nabla})\vec{u}\right)=-\vec{\nabla}p+\vec{\nabla}\cdot(\vec{\Pi}+\hat{\vec{\sigma}})+\sum_{\zeta}\vec{g}_{\zeta}, \tag{12.1}$$

$$\frac{\partial}{\partial t}\rho_{\zeta}+\vec{\nabla}\cdot(\rho_{\zeta}\vec{u})=\vec{\nabla}\cdot(\vec{D}_{\zeta}+\hat{\vec{\xi}}_{\zeta}), \tag{12.2}$$

$$\partial_t\rho+\vec{\nabla}\cdot(\rho\vec{u})=0, \tag{12.3}$$

where the index $\zeta = 1, 2$ specifies the component, $\vec{u}$ is the fluid (baricentric) velocity, $\rho = \sum_\zeta \rho_\zeta$ is the total density, and $p = \sum_\zeta p_\zeta = \sum_\zeta c_s^2\rho_\zeta$ is the internal pressure of the mixture ($c_s$ being the sound speed). Two fluctuating terms $\hat{\vec{\sigma}}$ and $\hat{\vec{\xi}}_\zeta$ are associated, respectivelu, to the diffusive current $\vec{D}_\zeta$ and to the viscous stress tensor $\vec{\Pi}$.

The coupling between the fluid components is realized by the force

$$\vec{g}_{\zeta}(\vec{r})=-\rho_{\zeta}(\vec{r})\sum_{\vec{r}'}\sum_{\zeta'}g_{\zeta\zeta'}\rho_{\zeta'}(\vec{r}')(\vec{r}'-\vec{r}), \tag{12.4}$$

that acts on the component $\zeta$ at node position $\vec{r}$, and depends on the densities on the neighboring nodes located at $\vec{r}'$. The width of the interfacial regions between two

components, that can be obtained with the Shan-Chen method is usually 5-10 lattice units. The coupling matrix $g_{\zeta\zeta'}$ is in general symmetric, so in the present implementation only three real values need to be specified with the option `sc_coupling`. The `lbfluid` command sets the density of the two components to the values specified by the option `dens`, and these can be modified with the `lbnode` command. Note that the number of active fluid components can be accessed through the global variable `lb_components`.

## 12.4. SC as a thermostat

The coupling of particle dynamics to the Shan-Chen fluid has been conceived as an extension of the Ahlrichs and Dünweg's point coupling, with the force acting on a particle given by

$$\vec{F} = -\frac{\sum_\zeta \gamma_\zeta \rho_\zeta(\vec{r})}{\sum_\zeta \rho_\zeta(\vec{r}_\zeta)} (\vec{v} - \vec{u}) + \vec{F}_R + \vec{F}^{ps}, \tag{12.5}$$

where $\zeta$ identifies the component, $\rho_\zeta(\vec{r})$ is a linear interpolation of the component density on the nodes surrounding the particle, $\gamma_\zeta$ is the component-dependent friction coefficient, $\vec{F}_R$ is the usual random force, and

$$\vec{F}^{\mathrm{ps}} = -\sum_\zeta \kappa_\zeta \nabla \rho_\zeta(\vec{r}). \tag{12.6}$$

This is an effective solvation force, that can drive the particle towards density maxima or minima of each component, depending on the sign of the constant $\kappa_\zeta$. Note that by setting the coupling constant to the same negative value for both components will, in absence of other forces, push the particle to the interfacial region.

In addition to the solvation force acting on particles, another one that acts on the fluid components is present, representing the solvation force of particles on the fluid.

$$\vec{F}^{\mathrm{fs}}_\zeta(\vec{r}) = -\lambda_\zeta \rho_\zeta(\vec{r}) \sum_i \sum_{\vec{r}'} \Theta \left[ \frac{(\vec{r}_i - \vec{r})}{|\vec{r}_i - \vec{r}|} \cdot \frac{(\vec{r}' - \vec{r})}{|\vec{r}' - \vec{r}|} \right] \frac{\vec{r}' - \vec{r}}{|\vec{r}' - \vec{r}|^2}, \tag{12.7}$$

where $\Theta(x) = 1$ if $0 < x < 1$, and 0 otherwise, the sum over lattice nodes is performed on the neighboring sites of $\vec{r}$ and the index $i$ runs over all particles. Note that a dependence on the particle index $i$ is assumed for $\kappa_\zeta$ and $\lambda_\zeta$. This force has the effect of raising or lowering (depending on the sign of the coupling constant $\lambda_\zeta$) the density in the eight nodes around a particle. The particle property `solvation` (Chap. 4) sets the coupling constants $\lambda_A$,$\kappa_A$,$\lambda_B$ and $\kappa_B$, where $A$ and $B$ denote the first and second fluid component, respectively. A complete description of the copuling scheme can be found in [53].

## 12.5. SC component-dependent interactions between particles

Often particle properties depend on the type of solvent in which they are. For example, a polymer chain swells in a good solvent, and collapses in a bad one. One of the possible

ways to model the good or bad solvent condition in coarse-grained models is to employ a WCA or a LJ (attractive) potential, respectively. If one wants to model the two components of the SC fluid as good/bad solvent, it is possible to do it using the `affinity` argument of the `inter` command. This non-bonded interaction type acts as a modifier to other interactions. So far only the Lennard-Jones interaction is changed by the `affinity`, so that it switches in a continuous way (after the potential minimum) from the full interaction to the WCA one. For more information see 5.1.2 and 5.2.3.

## 12.6. Reading and setting single lattice nodes

*TCL Syntax*

`lbnode` $x$ $y$ $z$ ( `print` | `set` ) *args*

Required features: [1] `LB` [2] `LB_GPU` [3] `SHANCHEN`

*Description*

The `lbnode` command allows to inspect (`print`) and modify (`set`) single LB nodes. Note that the indexing in every direction starts with 0. For both commands you have to specify what quantity should be printed or modified. Print allows the following arguments:

| | |
|---|---|
| `rho` | the density (one scalar[1,2] or two scalars[3]). |
| `u` | the fluid velocity (three floats: $u_x$, $u_y$, $u_z$) |
| `pi` | the fluid velocity (six floats: $\Pi_{xx}$, $\Pi_{xy}$, $\Pi_{yy}$, $\Pi_{xz}$, $\Pi_{yz}$, $\Pi_{zz}$) |
| `pi_neq` | the nonequilbrium part of the pressure tensor, components as above. |
| `pop` | the 19 population (check the order from the source code please). |
| `boundary` | the flag indicating whether the node is a fluid node (`boundary` $= 0$) or a boundary node (`boundary` $\neq 0$). Does not support `set`. Refer to the `lbboundary` command for this functionality. |

[1] `LB` or `LB_GPU`;
[2] `SHANCHEN`

Example: The line

```
puts [ lbnode 0 0 0 print u ]
```

prints the fluid velocity in node 0 0 0 to the screen. The command `set` allows to change the density or fluid velocity in a single node. Setting the other quantities can easily be implemented. Example:

```
puts [ lbnode 0 0 0 set u 0.01 0. 0.]
```

## 12.7. Removing total fluid momentum

*TCL Syntax*

`lbfluid remove_momentum`

Required features: [1] `LB` [2] `LB_GPU` [3] `SHANCHEN`

*Description*

In some cases, such as free energy profile calculations, it might be useful to prevent interface motion. This can be achieved using the command `lbfluid remove_momentum`, that removes the total momentum of the fluid.

## 12.8. Visualization

*TCL Syntax*

(1) `lbfluid print` [`vtk`] *property filename*

(2) `lbfluid print vtk velocity` [`bb1_x bb1_y bb1_z bb2_x bb2_y bb2_z`] *filename*

*Description*

The print parameter of the `lbfluid` command is a feature to simplify visualization. It allows for the export of the whole fluid field data into a file with name *filename* at once. Currently supported values for the parameter *property* are boundary and velocity when using `LB` or `LB_GPU` and density and velocity when using `SHANCHEN`. The additional option `vtk` enables export in the vtk format which is readable by visualization software such as paraview[1] or mayavi2[2]. Otherwise gnuplot readable data will be exported. If you plan to use paraview for visualization, note that also the particle positions can be exported in the VTK format 10.9. (2) allows you to only output part of the flow field by specifiying an axis aligned bounding box through the coordinates of two of its corners. This bounding box can be used to output a slice of the flow field. As an example, executing `lbfluid print vtk velocity 0 0 5 10 10 5 filename` will output the cross-section of the velocity field in a plane perpendicular to the $z$-axis at $z = 5$ (assuming the box size is 10 in the $x$- and $y$-direction). If the `SHANCHEN` bicomponent fluid is used, two filenames have to be supplied when exporting the density field, to save both components.

## 12.9. Setting up boundary conditions

*TCL Syntax*

(1) `lbboundary` *shape shape_args* [`velocity` *vx vy vz*]

(2) `lbboundary force` [$n_{\text{boundary}}$]

Required features: `LB_BOUNDARIES`

[1] http://www.paraview.org/

[2] http://code.enthought.com/projects/mayavi/

*Description*

If nothing else is specified, periodic boundary conditions are assumed for the LB fluid. Variant (1) allows to set up other (internal or external) boundaries.

The `lbboundary` command syntax is very close to the `constraint` syntax, as usually one wants the hydrodynamic boundary conditions to be shaped similarily to the MD boundaries. Currently the shapes mentioned above are available and their syntax exactly follows the syntax of the constraint command. For example

```
lbboundary wall dist 1.5 normal 1. 0. 0.
```

creates a planar boundary condition at distance 1.5 from the origin of the coordinate system where the half space $x > 1.5$ is treated as normal LB fluid, and the other half space is filled with boundary nodes.

Intersecting boundaries are in principle possible but must be treated with care. In the current, only partly satisfactory, all nodes that are within at least one boundary are treated as boundary nodes. Improving this is nontrivial, and suggestions are very welcome.

Currently, only the so called "link-bounce-back" algorithm for wall nodes is available. This creates a boundary that is located approximately midway between the lattice nodes, so in the above example this corresponds indeed to a boundary at $x = 1.5$. Note that the location of the boundary is unfortunately not entirely independent of the viscosity. This can *e.g.* be seen when using the sample script `poiseuille.tcl` with a high viscosity.

The bounce back boundary conditions allow to set velocity at a boundary to a nonzero value. This allows to create shear flow and boundaries moving relative to each other. This could be a fixed sphere in a channel moving at a finite speed – corresponding to the galilei-transform of a moving sphere in a fixed channel. The velocity boundary conditions are implemented according to [60] eq. 12.58. Using this implementation as a blueprint for the boundary treatment an implementation of the Ladd-Coupling should be relatively straightforward.

Variant (2) prints out the force on boundary number `n_boundary`.

## 12.10. Choosing between the GPU and CPU implementations

*TCL Syntax*

(1) `lbfluid cpu`
(2) `lbfluid gpu`
Required features: [1] `LB` [2] `LB_GPU`

*Description*

A very recent development is an implementation of the LBM for NVIDIA GPUs using the CUDA framework. On CUDA-supporting machines this can be activated by configuring with `configure --with-cuda=/path/to/cuda` and activating the feature `LB_GPU`. Within the ESPResSo-Tcl-script, the `lbfluid` command can be used to choose between

the CPU and GPU implementations of the Lattice-Boltzmann algorithm, for further information on CUDA support see section 6.5.

Variant (1) is the default and turns on the standard CPU implementation of the Lattice-Boltzmann fluid, while variant (2) turns on the GPU implementation, implying that all following LB-related commands are executed on the GPU.

Currently only a subset of the CPU commands are available for the GPU implementation. For boundary conditions analogous to the CPU implementation, the feature `LB_BOUNDARIES_GPU` has to be activated.

## 12.11. Electrohydrodynamics

*TCL Syntax*

`setmd mu_E` $\mu E_x$ $\mu E_y$ $\mu E_z$

Required features: `LB` `LB_ELECTROHYDRODYNAMICS`

*Description*

If the feature `LB_ELECTROHYDRODYNAMICS` is activated, the (non-GPU) Lattice Boltzmann Code can be used to implicitely model surrounding salt ions in an external electric field by having the charged particles create flow.

For that to work, you need to set the electrophoretic mobility (multiplied by the external $E$-field) $\mu E$ in all 3 dimensions for your system. The three given parameters are float values and should, for a meaningful system, be less than 1.0.

For more information on this method and how it works, read the publication [29].

# 13. Electrokinetics

The electrokinetics setup in ESPResSo allows for the description of electro-hydrodynamic systems on the level of ion density distributions coupled to a Lattice-Boltzmann (LB) fluid. The ion density distributions may also interact with explicit charged particles, which are interpolated on the LB grid. In the following paragraph we briefly explain the electrokinetic model implemented in ESPResSo, before we come to the description of the interface.

If you are interested in using the electrokinetic implementation in ESPResSo for scientific purposes, please contact G. Rempfer before you start your project.

## 13.1. Electrokinetic Equations

In the electrokinetics code we solve the following system of coupled continuity, diffusion-advection, Poisson, and Navier-Stokes equations:

$$\frac{\partial n_k}{\partial t} = -\nabla \cdot \vec{j}_k; \tag{13.1}$$

$$\vec{j}_k = -D_k \nabla n_k - \nu_k \, q_k n_k \, \nabla \Phi + n_k \vec{v}_{\mathrm{fl}}; \tag{13.2}$$

$$\Delta \Phi = -4\pi \, l_{\mathrm{B}} \, k_{\mathrm{B}} T \sum_k q_k n_k; \tag{13.3}$$

$$\begin{aligned} \left( \frac{\partial \vec{v}_{\mathrm{fl}}}{\partial t} + \vec{v}_{\mathrm{fl}} \cdot \vec{\nabla} \vec{v}_{\mathrm{fl}} \right) \rho_{\mathrm{fl}} &= -k_{\mathrm{B}} T \, \nabla \rho_{\mathrm{fl}} - q_k n_k \nabla \Phi \\ &\quad + \eta \vec{\Delta} \vec{v}_{\mathrm{fl}} + (\eta/3 + \eta_{\mathrm{b}}) \nabla (\nabla \cdot \vec{v}_{\mathrm{fl}}); \end{aligned} \tag{13.4}$$

$$\frac{\partial \rho_{\mathrm{fl}}}{\partial t} = -\nabla \cdot (\rho_{\mathrm{fl}} \vec{v}_{\mathrm{fl}}) \,, \tag{13.5}$$

which define relations between the following observables

- $n_k$ the number density of the particles of species $k$,
- $\vec{j}_k$ the number density flux of the particles of species $k$,
- $\Phi$ the electrostatic potential,
- $\rho_{\mathrm{fl}}$ the mass density of the fluid,
- $\vec{v}_{\mathrm{fl}}$ the advective velocity of the fluid,

and input parameters

| | |
|---|---|
| $D_k$ | the diffusion constant of species $k$, |
| $\nu_k$ | the mobility of species $k$, |
| $q_k$ | the charge of a single particle of species $k$, |
| $l_\mathrm{B}$ | the Bjerrum length, |
| $k_\mathrm{B}T$ | the thermal energy given by the product of Boltzmann's constant $k_\mathrm{B}$ and the temperature $T$, |
| $\eta$ | the dynamic viscosity of the fluid, |
| $\eta_\mathbf{b}$ | the bulk viscosity of the fluid. |

The temperature $T$, and diffusion constants $D_k$ and mobilities $\nu_k$ of individual species are linked through the Einstein-Smoluchowski relation $D_k/\nu_k = k_\mathrm{B}T$. The system of equations described in Eqs. (13.1)-(13.5), combining diffusion-advection, electrostatics, and hydrodynamics is conventionally referred to as the *Electrokinetic Equations*.

Complete in broad strokes the applicability of the electrokinetics model. Also mention the difference in temperatures between EK and LB species.

The electrokinetic equations have the following properties:

- On the coarse time and length scale of the model, the dynamics of the particle species can be described in terms of smooth density distributions and potentials as opposed to the microscale where highly localized densities cause singularities in the potential.

  In most situations, this restricts the application of the model to species of monovalent ions, since ions of higher valency typically show strong condensation and correlation effects – the localization of individual ions in local potential minima and the subsequent correlated motion with the charges causing this minima.

- Only the entropy of an ideal gas and electrostatic interactions are accounted for. In particular, there is no excluded volume.

  This restricts the application of the model to monovalent ions and moderate charge densities. At higher valencies or densities, overcharging and layering effects can occur, which lead to non-monotonic charge densities and potentials, that can not be covered by a mean-field model such as Poisson-Boltzmann or this one.

  Even in salt free systems containing only counter ions, the counter-ion densities close to highly charged objects can be overestimated when neglecting excluded volume effects. Decades of the application of Poisson-Boltzmann theory to systems of electrolytic solutions, however, show that those conditions are fulfilled for monovalent salt ions (such as sodium chloride or potassium chloride) at experimentally realizable concentrations.

- Electrodynamic and magnetic effects play no role. Electrolytic solutions fulfill those conditions as long as they don't contain magnetic particles.

- The diffusion coefficient is a scalar, which means there can not be any cross-diffusion. Additionally, the diffusive behavior has been deduced using a formalism

relying on the notion of a local equilibrium. The resulting diffusion equation, however, is known to be valid also far from equilibrium.

- The temperature is constant throughout the system.
- The density fluxes instantaneously relax to their local equilibrium values. Obviously one can not extract information about processes on length and time scales not covered by the model, such as dielectric spectra at frequencies, high enough that they correspond to times faster than the diffusive time scales of the charged species.

## 13.2. Setup

### 13.2.1. Initialization

*TCL Syntax*

`electrokinetics` [1 or 2 or 3] [`agrid` *agrid*] [`lb_density` *lb_density*] [`visc` *viscosity*] [`bulk_visc` *bulk_viscosity*] [`friction` *gamma* ] [`gamma_odd` *gamma_odd*] [`gamma_even` *gamma_even*] [`T` *T*] [`bjerrum_length` *bjerrum_length*] [`advection` *advection*] [`fluid-coupling` *fluid − coupling*] [`electrostatics_coupling`[4]]

Required features: [1] `ELECTROKINETICS` [2] `EK_BOUNDARIES` [3] `EK_REACTIONS` [4] `EK_ELECTROSTATIC_COUPLING`

*Description*

The `electrokinetics` command initializes the LB fluid with a given set of parameters, and it is very similar to the ESPResSo Lattice-Boltzmann `lbfluid` command in set-up. We therefore refer the reader to Chapter 12 for details on the implementation of LB in ESPResSo and describe only the major differences here.

The first major difference with the LB implementation is that the electrokinetics set-up is a Graphics Processing Unit (GPU) only implementation. There is no Central Processing Unit (CPU) version, and at this time there are no plans to make a CPU version available in the future. To use the electrokinetics features it is therefore imperative that your computer contains a CUDA capable GPU which is sufficiently modern.

To set up a proper LB fluid using the `electrokinetics` command one has to specify at least the following options: *agrid*, *lb_density*, *visc*, *friction*, $T$, and *bjerrum_length*. The other options can be used to modify the behavior of the LB fluid. Note that the `electrokinetics` command does not allow the user to set the time step parameter `tau` as is the case for the `lbfluid` command, this parameter is instead taken directly from the input of the `setmd t_step` command. The LB *mass density* is set independently from the electrokinetic *number densities*, since the LB fluid serves only as a medium through which hydrodynamic interactions are propagated, as will be explained further in the next paragraph. If no *lb_density* is specified, then our algorithm assumes *lb_density* = 1.0. The two 'new' parameters are $T$ the temperature at which the diffusive species

are simulated and *bjerrum_length* the Bjerrum length associated with the electrostatic properties of the medium. See the above description of the electrokinetic equations for an explanation of the introduction of a temperature, which does not come in directly via a thermostat that produces thermal fluctuations.

`advection` can be set to *on* or *off*. It controls whether there should be an advective contribution to the diffusive species' fluxes. Default is *on*.

`fluid-coupling` can be set to *friction* or *estatics*. This option determines the force term acting on the fluid. The former specifies the force term to be the sum of the species fluxes divided by their respective mobilities while the latter simply uses the electrostatic force density acting on all species. Note that this switching is only possible for the linkcentered stencil. For all other stencils, this choice is hardcoded. The default is *friction*.

`electrostatics_coupling` enables the action of the electrostatic potential due to the electrokinetics species and charged boundaries on the MD particles. The forces on the particles are calculated by interpolation from the electric field which is in turn calculated from the potential via finite differences. This only includes interactions between the species and boundaries and MD particles, not between MD particles and MD particles. To get complete electrostatic interactions a particles Coulomb method like Ewald or P3M has to be activate too.

### 13.2.2. Diffusive Species

*TCL Syntax*

`electrokinetics` [1 or 2 or 3] *species_number* [`density` *density*] [`D` *D*]
[`valency` *valency*] [`ext_force` $f_x$ $f_y$ $f_z$]
Required features: [1] `ELECTROKINETICS` [2] `EK_BOUNDARIES` [3] `EK_REACTIONS`

*Description*

The `electrokinetics` command followed by an integer *species_number* (in the range 0 to 10) and several options can be used to initialize the diffusive species. Here the options specify: the number density *density*, the diffusion coefficient $D$, the valency of the particles of that species *valency*, and an optional external (electric) force which is applied to the diffusive species. As mentioned before, the LB density is completely decoupled from the electrokinetic densities. This has the advantage that greater freedom can be achieved in matching the internal parameters to an experimental system. Moreover, it is possible to choose parameters for which the LB is more stable. The LB fluid must already be (partially) set up using the `electrokinetics` *agrid* ... command, before the diffusive species can be initialized. The variables *density*, $D$, and *valency* must be set to properly initialize the diffusive species; the *ext_force* is optional.

### 13.2.3. Boundaries

*TCL Syntax*

> `electrokinetics` [1 or 2 or 3] `boundary` [2] [`charge_density` *charge_density*] [`shape` *shape_args*]
>
> Required features: [1] `ELECTROKINETICS` [2] `EK_BOUNDARIES` [3] `EK_REACTIONS`

*Description*

The `boundary` command allows one to set up (internal or external) boundaries for the electrokinetics algorithm in much the same way as the `lbboundary` command is used for the LB fluid. The major difference with the LB command is given by the option *charge_density*, with which a boundary can be endowed with a volume charge density. To create a surface charge density, a combination of two oppositely charged boundaries, one inside the other, can be used. However, care should be taken to maintain the surface charge density when the value of *agrid* is changed. Currently, the following *shape*s are available: wall, sphere, cylinder, rhomboid, pore, stomatocyte, hollow_cone, and spherocylinder. We refer to the documentation of the `lbboundary` command (Chapter 12) for information on the options *shape_args* associated to these shapes. In order to properly set up the boundaries, the *charge_density* and relevant *shape_args* must be specified.

## 13.3. Output

### 13.3.1. Fields

*TCL Syntax*

> `electrokinetics` [1 or 2 or 3] `print` *property* [1 or 2] [`vtk`] *filename*
>
> Required features: [1] `ELECTROKINETICS` [2] `EK_BOUNDARIES` [3] `EK_REACTIONS`

*Description*

The print parameter of the `electrokinetics` command enables simple visualization of simulation data. A property of the fluid field can be exported into a file with name *filename* in one go. Currently, supported values of the parameter *property* are: *density*, *velocity*, *potential*, and *boundary*, which give the LB fluid density, the LB fluid velocity, the electrostatic potential, and the location and type of the boundaries, respectively. The boundaries can only be printed when the `EK_BOUNDARIES` is compiled in. The additional option `vtk` can be used to directly export in the vtk format. The vtk format is readable by visualization software such as paraview[1] and mayavi2[2]. If the [vtk] option is not specified, a gnuplot readable data file will be exported.

*TCL Syntax*

> `electrokinetics` [1 or 2 or 3] *species_number* `print` *property* [`vtk`] *filename*
>
> Required features: [1] `ELECTROKINETICS` [2] `EK_BOUNDARIES` [3] `EK_REACTIONS`

[1] http://www.paraview.org/
[2] http://code.enthought.com/projects/mayavi/

*Description*

This print statement is similar to the above command. It enables the export of diffusive species properties, namely: *density* and *flux*, which specify the number density and flux of species *species_number*, respectively.

### 13.3.2. Local Quantities

*TCL Syntax*

> `electrokinetics` [1 or 2 or 3] `node` *x y z* `velocity`
> Required features: [1] `ELECTROKINETICS` [2] `EK_BOUNDARIES` [3] `EK_REACTIONS`

*Description*

The `node` option of the `electrokinetics` command allows one to output the value of a quantity on a single LB node. The node is addressed using three integer values which run from 0 to *dim_x/agrid*, *dim_y/agrid*, and *dim_z/agrid*, respectively. Thus far, only the velocity of the LB fluid can be printed in the standard electrokinetics implementation. For other quantities the `lbnode` command may be used.

*TCL Syntax*

> `electrokinetics` [1 or 2 or 3] *species_number* `node` *x y z* `density`
> Required features: [1] `ELECTROKINETICS` [2] `EK_BOUNDARIES` [3] `EK_REACTIONS`

*Description*

This command can be used to output the number density of the *species_number*-th diffusive species on a single LB node.

## 13.4. Checkpointing

*TCL Syntax*

> (1) `electrokinetics` [1 or 2 or 3] `checkpoint save` *filename*
> (1) `electrokinetics` [1 or 2 or 3] `checkpoint load` *filename*
> Required features: [1] `ELECTROKINETICS` [2] `EK_BOUNDARIES` [3] `EK_REACTIONS`

*Description*

Variant (1) writes the species density fields as well as all necessary LB fields into two files called *filename.ek* and *filename.lb*. Variant (2) reads these files back and restores the fields. All algorithm parameters must be set via the simulation script, as they are not part of the checkpointed data.

The format of the checkpoint is binary and no special care is taken with respect to the specific binary layout of the machine.

## 13.5. Catalytic Reaction

### 13.5.1. Concept

The electrokinetics solver implemented in ESPResSo can be used to simulate a system, for which in addition to the electrokinetic equations, there is a (local) catalytic reaction which converts one species into another.

If you are interested in using this implementation in ESPResSo for scientific purposes, please contact J. de Graaf before you start your project.

Currently, a linear reaction is implemented which converts one species into two others, in order to model the catalytic decomposition of hydrogen peroxide in the presence of a platinum catalyst: $2H_2O_2 \rightarrow 2H_2O + O_2$. The decomposition of $H_2O_2$ is in reality more complicated than the linear reaction introduced here, since it is assumed to proceed via several intermediate complexed-states, but our model can be thought of as modeling the rate-limiting step. If we assume that there are three non-ionic species with number densities $n_k$, where $n_0 = [H_2O_2]$, $n_1 = [H_2O]$, and $n_2 = [O_2]$, then we can write the (electro)kinetic equations for this system as

$$\frac{\partial n_k}{\partial t} = -\nabla \cdot \vec{j}_k + f_k c n_k; \tag{13.6}$$

$$\vec{j}_k = -D_k \nabla n_k + n_k \vec{v}_{\mathrm{fl}}; \tag{13.7}$$

$$\left(\frac{\partial \vec{v}_{\mathrm{fl}}}{\partial t} + \vec{v}_{\mathrm{fl}} \cdot \vec{\nabla} \vec{v}_{\mathrm{fl}}\right) \rho_{\mathrm{fl}} = -k_{\mathrm{B}} T \sum_k \nabla n_k + \eta \vec{\Delta} \vec{v}_{\mathrm{fl}} + (\eta/3 + \eta_{\mathrm{b}}) \nabla (\nabla \cdot \vec{v}_{\mathrm{fl}}); \tag{13.8}$$

$$\frac{\partial \rho_{\mathrm{fl}}}{\partial t} = -\nabla \cdot (\rho_{\mathrm{fl}} \vec{v}_{\mathrm{fl}}), \tag{13.9}$$

which define relations between the following observables

- $n_k$ the number density of the particles of species $k$,
- $\vec{j}_k$ the number density flux of the particles of species $k$,
- $\rho_{\mathrm{fl}}$ the mass density of the fluid,
- $\vec{v}_{\mathrm{fl}}$ the advective velocity of the fluid,

and input parameters

- $D_k$ the diffusion constant of species $k$,
- $k_{\mathrm{B}}T$ the thermal energy given by the product of Boltzmann's constant $k_{\mathrm{B}}$ and the temperature $T$,
- $\eta$ the dynamic viscosity of the fluid,

$\eta_{\mathbf{b}}$ the bulk viscosity of the fluid,

$f_k$ the reaction constant $f_0 \equiv -1$, $f_1 = 1$ and $f_2 = 0.5$ for the above reaction,

$c$ the reaction rate.

In this set of equations we have fully decoupled the number densities and the fluid mass density. N.B. We have set the initial fluid mass density is not necessarily equal to the sum of the initial species number densities. This means that some care needs to be taken in the interpretation of the results obtained using this feature. In particular, the solution of the Navier-Stokes equation exclusively models the momentum transport through the (multicomponent) fluid, while the diffusive properties of the individual chemical species are handled by Eqs. (13.6) and (13.7).

It is important to note that to ensure mass conservation the reaction must satisfy:

$$\sum_k f_k m_k = 0, \tag{13.10}$$

where $m_k$ is the molecular mass of a reactive species. Unfortunately, the current electrokinetic implementation does not conserve mass flux locally. That is to say, the LB fluid is compressible and the sum of the fluxes of the three species is not equal to zero in the frame co-moving with the advective fluid velocity. It is therefore debatable whether it is necessary to impose Eq. (13.10), since the EK algorithm itself does not conserve mass density. However, we strived to be as accurate as possible and in future versions of the EK algorithm the lack of incompressiblity will be addressed.

The reaction is specified by the second term on the right-hand side of Eq. (13.6). It is important to note that this term can be set locally, as opposed to the other terms in the equation system Eqs. (13.6)-(13.9), in our implementation, as will become clear in the following. This has the advantage that catalytic surfaces may be modeled.

### 13.5.2. Initialization and Geometry Definition

*TCL Syntax*

`electrokinetics` [1 or 2 or 3] `reaction` [3] [`reactant_index` *reactant_index*]
[`product0_index` *product0_index*] [`product1_index` *product1_index*]
[`reactant_resrv_density` *reactant_resrv_density*]
[`product0_resrv_density` *product0_resrv_density*]
[`product1_resrv_density` *product1_resrv_density*]
[`reaction_rate` *reaction_rate*] [`mass_reactant` *mass_reactant*]
[`mass_product0` *mass_product0*] [`mass_product1` *mass_product1*]
[`reaction_fraction_pr_0` *reaction_fraction_pr_0*]
[`reaction_fraction_pr_1` *reaction_fraction_pr_1*]

Required features: [1] `ELECTROKINETICS` [2] `EK_BOUNDARIES` [3] `EK_REACTIONS`

*Description*

The `electrokinetics reaction` command is used to set up the catalytic reaction between three previously defined the diffusive species, of which the i identifiers are given by

*reactant_index*, *product0_index*, and *product1_index*, respectively. In the 1:2 reaction, these fulfill the role of the reactant and the two products, as indicated by the naming convention. For each species a reservoir (number) density must be set, given by the variables *reactant_resrv_density*, *product0_resrv_density*, and *product1_resrv_density*, respectively. These reservoir densities correspond to the initial number densities associated with the reactive species. The reservoir densities, in tandem with reservoir nodes, see below, can be used to keep the reaction from depleting all the reactant in the simulation box. The *reaction_rate* variable specifies the speed at which the reaction proceeds. The three masses (typically given in the atomic weight equivalent) are used to determine the total mass flux provided by the reaction, as described above, and are also used to check whether the reaction ratios that are given satisfy the chemical requirement of mass conservation. Finally, the parameters *reaction_fraction_pr_0* and *reaction_fraction_pr_1* specify what fractions of the product are generated when a given quantity of reactant is catalytically converted. To use a chemical reaction, all options for the `electrokinetics reaction` command must be specified.

*TCL Syntax*

`electrokinetics` [1 or 2 or 3] `reaction` [3] `region` [3] [`reaction_type` *reaction_type*] [`shape` *shape_args*]

Required features: [1] `ELECTROKINETICS` [2] `EK_BOUNDARIES` [3] `EK_REACTIONS`

*Description*

The `region` option of the `electrokinetics reaction` command allows one to set up regions in which the reaction takes place with the help of the constraints that are available to set up boundaries. The integer value *reaction_type* can be used to select the reaction: 0 no reaction takes place for this region, 1 the catalytic reaction takes place in this region, and 2 the region functions as a reservoir, wherein the species densities are reset to their initial (or reservoir) concentrations. The rest of the command follows the same format of the `electrokinetics boundary` command. Currently, the following *shape*s are available: box, wall, sphere, cylinder, rhomboid, pore, stomatocyte, hollow_cone, and spherocylinder. The box shape is a `region` specific command, which can be used to set the entire simulation box to a specific reaction value. To use the `electrokinetics reaction region` command, one must first set up a reaction, as described above. To successfully specify a region all the relevant arguments that go with the shape constraints must be provided.

### Parsing PDB Files

*TCL Syntax*

`electrokinetics` [1 or 2 or 3] `pdb-parse` [2] *pdb_filename* *itp_filename*

Required features: [1] `ELECTROKINETICS` [2] `EK_BOUNDARIES` [3] `EK_REACTIONS`

*Description*

The `electrokinetics pdb-parse` feature allows the user to parse simple PDB files, a file format introduced by the protein database to encode molecular structures. To-

gether with a topology file (here *itp_filename*) the structure gets interpolated to the `electrokinetics` grid. For the input you will need to prepare a PDB file with a `gromacs` force field to generate the topology file. Normally the PDB file extension is `.pdb`, the topology file extension is `.itp`. Obviously the PDB file is placed instead of *pdb_filename* and the topology file instead of *itp_filename*.

At the moment this fails badly, if you try to parse incorrectly formatted files. This will be fixed in the future.

### 13.5.3. Reaction-Specific Output

*TCL Syntax*

`electrokinetics` [1 or 2 or 3] `print` *property* [3] [`vtk`] *filename*

Required features: [1] `ELECTROKINETICS` [2] `EK_BOUNDARIES` [3] `EK_REACTIONS`

*Description*

The print parameter of the `electrokinetics` command can be used in combination with the `EK_REACTION` feature to give advanced output options. Currently, supported values of the parameter *property* are: *pressure* and *reaction_tags*, which give the location and type of the reactive regions and the ideal-gas pressure coming from the diffusive species, respectively. To use this command a reaction must be set up.

# 14. Object-in-fluid

Please cite [14] (BibTeX-key `cimrak` in file `doc/ug/citations.bib`) if you use the object-in-fluid implementation described below. For more details also see the documentation at http://cell-in-fluid.fri.uniza.sk/oif-documentation or contact the Cell-in-fluid Research Group at University of Žilina.

Simulations using ESPResSo work mostly with objects (molecules, atoms, polymers, colloids, crystals, ...) that are physically composed of points linked together with bonds. These objects are like skeletons, without inner or outer volume.

The idea behind this module, is to use ESPResSo for objects that do have inner volume, for example blood cells, magnetic beads, capsules, ... The boundary of an object is covered with triangular mesh. The vertices of the mesh are declared in ESPResSo as particles. The edges of the mesh define elastic forces keeping the shape of the object. The movement of object is achieved by adding forces to the mesh points.

Modelled elastic or rigid objects are immersed in the LB fluid flow. The fluid interacts with an elastic object resulting in its deformation; this immediately generates forces acting back on the fluid. The aim is to describe the immersed object using the notion of particles, and to create bonds between these particles representing elastic or rigid forces.

The objects are composed of a membrane encapsulating the fluid inside the object. For now, the inside fluid must have the same density and viscosity as the outside fluid. The object is represented by its membrane (boundary), that is discretized using a triangulation. Such triangulation defines interacting particles distributed on the surface of the immersed object [23]:

- between two particles, corresponding to the edges in the triangulation (modelling the stretching of the membrane),
- between three particles, corresponding to the triangles of the triangulation (local area, or local surface preservation of the membrane),
- between four particles, corresponding to two triangles from the triangulation sharing a common edge (bending of the membrane).

The object immersed in the fluid moves under the influence of the deforming forces, defined through the bonds, and under the influence of the fluid motion. This interaction is based on the frictional force between the fluid and the surface particles. Therefore the object moves in the flow only if there is a nonzero difference between the fluid velocity and the particle velocity. In other words, there has to be at least small flow through the membrane, which is in most cases unphysical. However, this unphysical flow through the membrane is probably negligible in larger scales.

## 14.1. Membranes

With this approach, it is easy to model also elastic sheets, or free membranes that do not necessarily enclose a 3D object. In this case, area_force_global and volume_force interactions are not needed, since these two interactions are meant for closed immersed objects.

## 14.2. Parameters

There are several parameters involved in this model. All of them should be calibrated according to the intended application.

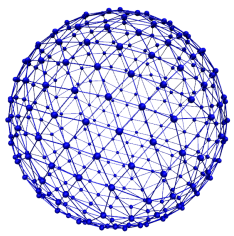
- Mass of the particles. Every particle has its mass, which influences the dynamics.
- Friction coefficient. The main parameter describing the fluid-particle interaction is the `friction` parameter from the ESPResSo command `lbfluid`.
- Parameters of elastic moduli. Elastic behaviour can be described by five different elastic moduli: hyperelastic stretching, linear stretching, bending, local and global area preservation and volume preservation. Each of them has its own scaling parameter: $k_s, ks_{lin}, k_b, k_{al}, k_{ag}, k_v$. Their mathematical formulations have been taken from [23].

The mass of the particles and the friction coefficient can be calibrated using the drag coefficients of the ellipsoidal objects. These drag coefficients have known analytical values and the mass and friction can be calibrated to fit this values. More details about the calibration can be found in [14].

The elastic parameters are specific to the immersed objects. They correspond to their physical values. More details about their mechanical and biological meaning is presented in [16] specifically for red blood cells. However, the proper calibration to fit the experimental data has been performed in [14].

## 14.3. Geometry

The membrane of the immersed object is triangulated. In doc/tutorials/03-object_in_fluid you can find an example using deformable objects in the fluid.



Triangulation can be obtained using various software tools. Two files are needed for mesh input: `mesh-nodes.dat` and `mesh-triangles.dat`. The parameters of the mesh are the number of particles on the surface of the immersed object, denoted by `mesh_nnode`, and the number of triangular faces in the triangulation, denoted by `mesh_ntriangle`. These parameters are obtained automatically from `mesh-nodes.dat` and `mesh-triangles.dat`by counting the number of lines in respective files.

The `mesh-nodes.dat` thus contains `mesh_nnode` lines with three real numbers separated by blank space, representing three coordinates of the corresponding particle. The membrane is thus discretized into `mesh_nnode` particles with IDs starting from 0 to `mesh_nnode-1`. The IDs are assigned in the same order as in the `mesh-nodes.dat` file.

The `mesh-triangles.dat` contains `mesh_ntriangle` lines with three nonnegative integers separated by blank space. Each line represents one triangle in the triangulation. For algorithmic purposes it is crucial to have defined a correct orientation of the triangle. The orientation is defined using the normal vector associated with the triangle. The important rule is that the normal vector of the triangle must point inside the immersed object.

As an example, let us have one line in the file `mesh-triangles.dat` with numbers 4, 0 and 7. This means that particles with IDs 4, 0 and 7 form one triangular face of the triangulation. The orientation is defined as follows: create two vectors $v_1$ and $v_2$, such that $v_1$ is pointing from particle 4 to particle 0, and $v_2$ is pointing from particle 4 to particle 7. Be careful, the order of vectors and particles matters!

The normal vector $n$ is computed as a vector product $v_1 \times v_2$. The direction of $n$ can be determined by the rule of right hand: the thumb points in the $v_1$ direction, the index finger in the $v_2$ direction and the middle finger in the $n$ direction. Following this principle, all the lines in the `mesh-triangles.dat` files must be such that the normal vectors of the corresponding triangles points inside the immersed object.

These two files are sufficient to describe the geometry and topology of the triangulation. The following geometric entities are necessary for the definition of bonded interactions: position of the particles, edges, lengths of the edges, triangles, areas of triangles, angles between two triangles sharing a common edge, surface of the immersed object, volume of the immersed object. All these geometrical entities can be computed using the information from the files `mesh-nodes.dat` and `mesh-triangles.dat` and the computation is done in the script `scripts/object_in_fluid.tcl`.

The script `scripts/object_in_fluid.tcl` reads both mesh files, generates list of edges, and computes all geometrical entities needed for definition of bonded interactions. It then executes commands creating the particles, interactions and bonds.An example of `part` command is as follows:

```
part 0 pos 3.0 3.0 6.0 type 1 mol 1 mass 1
```

Note, the is feature `mol` that used for the particles. We use this feature we distinguish between different objects. The upper limit for the number of objects is 10000. However it can be increased by changing the `MAX_OBJECTS_IN_FLUID` constant.

The following example shows an interaction.

```
inter 106 oif_local_force 1.0 0.5 0.0 1.7 0.6 0.2 0.3 1.1
```

This command ("invisible" for the user who executes the `scripts/object_in_fluid.tcl` script) takes care of stretching, bending and local area conservation all in one interaction with ID 106. Detailed description of the available types of interactions is presented in Section 5.4.

## 14.4. Available commands

In order to use the object-in-fluid (OIF) commands and work with immersed objects, the following ESPResSo features need to be compiled in: `MASS, EXTERNAL_FORCES`. We do not specifically require `LB, LB_BOUNDARIES, CONSTRAINTS, SOFT_SPHERE, MEMBRANE_COLLISION, OIF_LOCAL_FORCES, OIF_GLOBAL_FORCES`. They are most likely to be used (for objects immersed in fluid and interacting with boundaries and each other), but they are not necessary for the following commands. For up-to-date overview of available oif commands see the OIF user guide at cell-in-fluid.fri.uniza.sk/oif-documentation.

### 14.4.1. Initialisation

*TCL Syntax*

`oif_init`

*Description*
Must be used before any other OIF command, initializes all global variables and lists, does not take any arguments.

### 14.4.2. Information about object-in-fluid structures

*TCL Syntax*

`oif_info`

*Description*
Prints information about whole framework, *e.g.* all global variables, currently available templates and objects, etc. Does not take any arguments.

### 14.4.3. Templates for objects

*TCL Syntax*

`oif_create_template` [1,2] `template-id` *tid* `nodes-file` *nodes.dat*
`triangles-file` *triangles.dat* [`stretch` *x y z*] [`mirror` *x y z*]
[`ks` *ks_value*] [`kslin` *kslin_value*] [`kb` *kb_value*] [`kal` *kal_value*]
[`kag` *kag_value*] [3] [`kv` *kv_value*] [4] [`normal`] [5]

Required features: [1] `MASS` [2] `EXTERNAL_FORCES` [3] `OIF_GLOBAL_FORCES`
[4] `OIF_LOCAL_FORCES` [5] `MEMBRANE_COLLISION`

*Description*

This command creates a template that will be used for all objects that share the same elastic properties and have the same triangulation.

*Arguments*

- *tid* specifies a unique ID for each template. The first template has the ID 0. The following ones need to be be numbered consecutively.
- *nodes.dat* input file, each line contains three real numbers. These are the $x, y, z$ coordinates of individual surface mesh nodes of the objects.
- *triangles.dat* input file, each line contains three integers. These are the ID numbers of the mesh nodes as they appear in *nodes.dat*. Note, the first node has ID 0.
- [`stretch` *x y z*] coefficients by which the coordinates stored in *nodes.dat* will be stretched in the $x, y, z$ direction. The default values are 1.0 1.0 1.0.
- [`mirror` *x y z*] whether the respective coordinate will be flipped around 0. Coefficients $x, y, z$ must be either 0 or 1. The reflection of only one coordinate is allowed so at most one number is 1, others are 0. For example `mirror` *0 1 0* results in flipping the spatial point $(x, y, z)$ to $(x, -y, z)$. The default value is 0 0 0.
- [`ks` *ks_value*] elastic modulus for hyperelastic stretching forces
- [`kslin` *kslin_value*] elastic modulus for linear stretching forces
- [`kb` *kb_value*] elastic modulus for bending forces
- [`kal` *kal_value*] elastic modulus for local area forces
- [`kag` *kag_value*] elastic modulus for global area forces
- [`kv` *kv_value*] elastic modulus for volume forces
- [`normal`] switch to turn on the computation of local outward normal vectors

The four switches `ks`, `kslin`, `kb` and `kal` set elastic parameters for local interactions - `ks` for hyperelastic edge stiffness, `kslin` for linear edge stiffness, `kb` for angle preservation stiffness and `kal` for triangle surface preservation stiffness. This stiffness can be either uniform over the whole object, or non-uniform. In case of stretching modulus, we can have spring stiffness the same for all edges in the whole object, or we can choose the value for every edge of the object separately. Analogically, for `kslin`, for `kal` and `kb`. Therefore, there are two options for setting `ks`, `kslin`, `kal` and `kb` stiffness. Here is the explanation for `ks`:

- **Uniform stiffness:** To set uniform hyperelastic stiffness for all edges in the object, use `ks` *ks_value*

- **Non-uniform stiffness:** To set non-uniform hyperelastic stiffness, prepare a file *filename* with number of lines equal to the number of edges of the triangulation. Each line should contain a real number between 0 and 1, so called "weight". Then call `ks` *filename ksMin ksMax* This command reads the weights $weight_i$ for each edge and the stiffness for that edge is set to

$$ks_i = ksMin * (1 - weight_i) + ksMax * (weight_i)$$

For bending stiffness, *filename* must contain the same number of lines as there are edges in the object. However, for local area preservation, the stiffness constant is linked to triangles. Therefore, *filename* must contain the same number of lines as there are triangles in the object.

**Warning:** At least one elastic modulus needs to be set for the object.

### 14.4.4. Elastic objects

*TCL Syntax*

`oif_add_object` [1,2, possibly 3,4] `object-id` *oid* `template-id` *tid* `origin` *x y z* `part-type` *type* [`rotate` *x y z*] [`mass` *m*]

Required features: [1] `MASS` [2] `EXTERNAL_FORCES` [3] `OIF_GLOBAL_FORCES` [4] `OIF_LOCAL_FORCES` [5] `MEMBRANE_COLLISION`

*Description*

Using a previously defined template *tid*, this command creates a new object. Features `OIF_LOCAL_FORCES`, `OIF_GLOBAL_FORCES`, `OIF_MEMBRANE_COLLISION` are needed, if the template used the corresponding elastic moduli.

*Arguments*

- *oid* unique ID for each object, the first object has the ID 0. The following ones should be numbered consecutively.
- *tid* object will be created using nodes, triangle incidences, elasticity parameters and initial stretching saved in this template.
- `origin` *x y z* center of the object will be at this point.
- `part-type` *type* can be any integer starting at 0. All particles of one object have the same `part-type`. One can have more objects with the same type of particles, but this is not recommended, because the interactions between objects are set up using these types.
- [`rotate` *x y z*] angles in radians, by which the object is initially rotated around the $x, y, z$ axis. Default values are 0.0 0.0 0.0.
- [`mass` *m*] this parameter refers to the mass of one particle (one mesh point of the triangulation). For the proper setting, the mass of the whole membrane must be distributed to all mesh points. Default value is 1.0.

### 14.4.5. Mesh analysis

*TCL Syntax*

`oif_mesh_analyze nodes-file` *nodes.dat* `triangles-file` *triangles.dat* [`orientation`] [`repair` *output_file.dat method*] [`shift-node-ids` *output_file.dat*]

*Description*

This command is useful for some preparatory work with mesh before it is used for creating elastic objects.

*Arguments*

- `nodes-file` *nodes.dat* - file with coordinates of the mesh nodes. The center of the object should be as close to (0,0,0) as possible.
- `triangles-file` *triangles.dat* - file with incidences for all triangles. Each line of this file contains three integer IDs (starting from 0) with indices of three vertices forming one triangle.
- [`orientation`] checks whether all triangles of the surface mesh are properly oriented. For now, only works for convex (or almost convex) objects.
- [`repair` *output_file.dat method*] outputs the corrected *triangles.dat* file into *output_file.dat*. For now, only works for convex (or almost convex) objects. *method* needs to be set to 1.
- [`shift-node-ids` *output_file.dat*] subtracts 1 from all numbers in *triangles.dat* and saves a new file *output_file.dat*. This is useful, if the mesh generating software starts numbering the particles from 1 instead of 0.

### 14.4.6. Output information about specific object

*TCL Syntax*

`oif_object_output` [1,2, possibly 3,4] `object-id` *oid* [`vtk-pos` *output_file1.dat*] [`vtk-pos-folded` *output_file2.dat*] [`vtk-aff` *output_file3.dat*] [`mesh-nodes` *output_file4.dat*]

Required features: [1] `MASS` [2] `EXTERNAL_FORCES` [3] `OIF_GLOBAL_FORCES` [4] `OIF_LOCAL_FORCES`

*Description*

This command is used to output information about the object that can be used for visualisation or as input for other simulations.

*Arguments*

- *oid* - the id of the object
- [`vtk-pos` *output_file1.dat*] outputs the mesh of the object to the desired *output_file1.dat*. Paraview can directly visualize this file.

- [`vtk-pos-folded` *output_file2.dat*] the same as the previous option, however the whole object is shift such that it is visualized within the simulation box. This option is useful for simulating periodical processes when objects flowing out on one side of simulation box are transferred to the opposite side.
- [`vtk-aff` *output_file3.dat*] outputs affinity bonds that are currently activated. If no bonds are present, the file will be generated anyway with no bonds to visualize. Paraview can directly visualize this file.
- [`mesh-nodes` *output_file4.dat*] outputs the positions of the mesh nodes to *output_file4.dat*. In fact, this command creates a new *nodes.dat* file that can be used by `oif_object_set`. The center of the object is located at point (0,0,0). This command is aimed to store the deformed shape in order to be loaded later.

### 14.4.7. Descriptive information about specific object

*TCL Syntax*

`oif_object_analyze` [1,2, possibly 3,4] `object-id` *oid* [`origin`] [`pos-bounds` *bname*] [`approx-pos`] [`edge-statistics`] [`volume`] [`surface-area`] [`velocity`] [`elastic-forces` *name(s)* *output_file.dat*] [`f-metric` *name*]

Required features: [1] `MASS` [2] `EXTERNAL_FORCES` [3] `OIF_GLOBAL_FORCES` [4] `OIF_LOCAL_FORCES`

*Description*

This command is used to output information about the properties of the object. Some of these properties can also be visualized.

*Arguments*

- *oid* - the id of the object
- [`origin`] - outputs the location of the center of the object
- [`pos-bounds` *bname*] computes six extremal coordinates of the object. More precisely, runs through the all mesh points and remembers the minimal and maximal $x$-coordinate, $y$-coordinate and $z$-coordinate. If *bname* is one of these: *z-min, z-max, x-min, x-max, y-min, y-max* then the procedure returns one number according to the value of *bname*. If *bname* is *all*, then the procedure returns a list of six numbers, namely *x-min, x-max, y-min, y-max, z-min, z-max*.
- [`approx-pos`] - outputs the approximate location of the center of the object. It is computed as average of 6 mesh points that have extremal $x$, $y$ and $z$ coordinates at the time of object loading.
- [`edge-statistics`] - outputs the minimum, average and maximum edge length of the object and corresponding standard deviation
- [`volume`] - outputs the current volume of the object

- [`surface-area`] - outputs the current surface of the object
- [`velocity`] - outputs the current average velocity of the object. Runs over all mesh points and calculates their average velocity.

### 14.4.8. Setting properties for specific object

*TCL Syntax*

> `oif_object_set` [1,2, possibly 3,4] `object-id` *oid* [`force` *x y z*] [`origin` *x y z*] [`mesh-nodes` *mesh_nodes.dat*] [`kill-motion`] [`un-kill-motion`]
>
> Required features: [1] `MASS` [2] `EXTERNAL_FORCES` [3] `OIF_GLOBAL_FORCES` [4] `OIF_LOCAL_FORCES`

*Description*

This command sets some properties of the object.

*Arguments*

- *oid* - the id of the object
- [`force` *x y z*] - sets the force vector $(x, y, z)$ to all mesh nodes of the object. Setting is done using ESPResSo command `part $i set ext_force` $\$x$ $\$y$ $\$z$. Note, that this command sets the external force in each integrate step. So if you want to use the external force only in one iteration, you need to set zero external force in the following integrate step
- [`origin` *x y z*] - moves the object so that the origin has coordinates $(x, y, z)$
- [`mesh-nodes` *mesh_nodes.dat*] - deforms the object such that its origin stays unchanged, however the relative positions of the mesh points are taken from file *mesh_nodes.dat*. The file *mesh_nodes.dat* should contain the coordinates of the mesh points with the origin's location at (0,0,0). The procedure also checks whether number of lines in the *mesh_nodes.dat* file is the same as the number of triangulation nodes of the object.
- `kill-motion` - stops all the particles in the object (analogue to the `part` *pid* `fix 1 1 1` command for single particles).
- `un-kill-motion` - releases the particles in the object (analogue to the `part` *pid* `unfix` command for single particles).

# 15. Immersed Boundary Method for soft elastic objects

Please contact the Biofluid Simulation and Modeling Group at the University of Bayreuth if you plan to use this feature.

This section describes an alternative way to include soft elastic objects somewhat different from the previous chapter. In the Immersed Boundary Method (IBM), soft particles are considered as an infinitely thin shell filled with liquid (see e.g. [47, 15, 38]). When the shell is deformed by an external flow it responds by elastic restoring forces which are transmitted into the fluid. In the present case, the inner and outer liquid are of the same type and are simulated using Lattice-Boltzmann.

Numerically, the shell is discretized by a set of marker points connected by triangles. The marker points are advected with *exactly* the local fluid velocity, i.e., they do not possess a mass nor a friction coefficient (this is different from the Object-in-Fluid method of the previous chapter). We implement these marker points as virtual particles in ESPResSo which are not integrated using the usual velocity-verlet scheme, but instead are propagated using a simple Euler algorithm with the local fluid velocity (if the `IMMERSED_BOUNDARY` feature is turned on).

To compute the elastic forces, three new bonded interactions are defined ibm_triel, ibm_tribend and ibm_volCons:

- ibm_triel is a discretized elastic force with the following syntax

  *TCL Syntax*

  `inter` *ID* `ibm_triel` *ind1 ind2 ind3 max law*

  Required features: `IMMERSED_BOUNDARY`

  *Description*

  where *ind1*, *ind2* and *ind3* represent the indices of the three marker points making up the triangle. The parameter *max* specifies the maximum stretch above which the bond is considered broken. The final parameter *law* can be either

  `NeoHookean <k>`

  or

  `Skalak <k1> <k2>`

  which specifies the elastic law and its corresponding parameters (see e.g. [38]).

- ibm_tribend is a discretized bending potential with the following syntax

  *TCL Syntax*

  > `inter` *ID* `ibm_tribend` *ind1 ind2 ind3 ind4 method kb* [`flat|initial`]
  > Required features: `IMMERSED_BOUNDARY`

  *Description*

  where *ind1*, *ind2*, *ind3* and *ind4* are four marker points corresponding to two neighboring triangles. The indices *ind1* and *ind3* contain the shared edge. Note that the marker points within a triangle must be labelled such that the normal vector $\vec{n} = (\vec{r}_{\text{ind2}} - \vec{r}_{\text{ind1}}) \times (\vec{r}_{\text{ind3}} - \vec{r}_{\text{ind1}})$ points outward of the elastic object.

  The parameter *method* allows to specify different numerical ways of computing the bending interaction. Currently, two methods are implemented, where the first one ([TriangleNormals]) follows [38] and the second one ([NodeNeighbors]) follows [27]. In both cases, *kb* is the bending modulus. The options [flat] or [initial] specify whether the reference shape is a flat configuration or whether the initial configuration is taken as reference shape, this option is only available for the [TriangleNormals] method.

- ibm_volCons is a volume-conservation force. Without this correction, the volume of the soft object tends to shrink over time due to numerical inaccuracies. Therefore, this implements an artificial force intended to keep the volume constant. If volume conservation is to be used for a given soft particle, the interaction must be added to every marker point belonging to that object. The syntax is

  *TCL Syntax*

  > `inter` *ID* `ibm_volCons` *softID kv*
  > Required features: `IMMERSED_BOUNDARY`

  *Description*

  where *softID* identifies the soft particle and *kv* is a volumetric spring constant [38].

# 16. External package: mbtools

mbtools[1] is a set of tcl packages for setting up, analyzing and running simulations of lipid membrane systems.

mbtools comes with a basic set of tabulated forces and potentials for lipid interactions and some example scripts to help explain the syntax of the commands. If you make use of mbtools or of these potentials please acknowledge us with a citation to:

* Cooke, I. R., Kremer, K. and Deserno, M. (2005): Tunable, generic model for fluid bilayer membranes, Phys. Rev. E. 72 - 011506

## 16.1. Introduction

mbtools is located in the folder `Espresso/packages/mbtools`.

One of the main features of mbtools is the ability to easily create initial lipid configurations with interesting geometries. These include flat membranes, cylinders, spheres, toroids, and randomly distributed gases. Each of these shapes is referred to as a geometry and any number of geometries can be combined in a single simulation. Once the geometry has been chosen the user specifies the molecules which should be placed in this geometry. For example one could choose sphere as a geometry and then define two different lipids and/or a protein to be placed on the sphere. Within reason (e.g. size restrictions) it should be possible to use any mixture of known molecule types on any geometry. The molecule types available at present include proteins, lipids of any length, and spherical colloids.

mbtools includes several miscellaneous utility procedures for performing tasks such as warmup, setting tabulated interactions, designating molecules to be trapped and a variety of topology related sorting or data analysis functions.

The analysis part of the mbtools package is designed to wrap together all the analysis for a simulation into a single simple interface. At the beginning of the simulation the user specifies which analyses should be performed by appending its name and arguments to a variable, `analysis_flags`. After the analysis is setup one can then simply call `do_analysis` to perform all the specified proceedures. Analysis will store a data value each time `do_analysis` is called. Then when a call to `print_averages` is made the average of all stored values is printed to a file and the store of values is reset to nil.

[1] This documentation was written by Ira R. Cooke and published on his website. It has been transcripted by Tristan Bereau.

## 16.2. Installing and getting started

Since mbtools is provided as part of the espresso molecular dynamics simulation package you will need to download and install Espresso before you can use it. Espresso can be downloaded free from `http://espressomd.org`.

Once you have installed espresso you can find mbtools by looking inside the `packages` subdirectory. Inside the `packages/mbtools` directory you will see a directory for each of the mbtools subpackages as well as an `examples` directory. All of the examples scripts should work out of the box except those involving colloids which require you to install icover.sh (see documentation for hollowsphere molecule type). To run the simplebilayer example cd to the examples directory and then type:

```
$ESPRESSO_SOURCE/$PLATFORM/Espresso scripts/main.tcl simplebilayer.tcl
```

The first part of this command is simply the full path to the appropriate espresso executable on your machine when running on multiple processors). Obviously you will need to have the `$ESPRESSO_SOURCE` and `$PLATFORM` environment variables set for it to work. After this executable the relative paths to two tcl scripts are given. The first of these `main.tcl` is given as an argument to espresso and is therefore interpreted first by the espresso tcl interpreter. The second tcl script `simplebilayer.tcl` is in turn passed as an argument to `main.tcl`.

Why separate the tcl commands into two files ?

This is really a matter of preference but if we keep all of the key commands and complex coding in a single file `main.tcl` and delegate simple parameter setting to a separate file it tends to be much easier to manage large numbers of jobs with regularly changing requirements. Regardless of your personal preferences, the important point to note is that all of the important commands are contained in `main.tcl` and you should probably start there to get an understanding for how mbtools works.

Running the simplebilayer example should produce a directory called `simplebilayer` which contains the output from your simulation. To view the results cd to the simplebilayer directory and look at the contents. The directory contains many files including:

- The configurations generated during warmup : `warm.*.gz`
- pdb files corresponding to warmup configurations : `warm.vmd*.gz`
- The configurations generated during the main run : `simplebilayer.*.gz`
- pdb files corresponding to main run configs : `simplebilayer.vmd*.gz`
- The most recently generated checkpoint file : `checkpoint.latest.gz`
- A directory containing the second most recent checkpoint file: `checkpoint_bak`
- A file containing the topology of the system : `simplebilayer.top`
- The original parameter file with which you ran the simulation : `simplebilayer.tcl`

- A original parameter file with which you ran the simulation : `simplebilayer.tcl`
- Files containing analysis output for example : `time_vs_boxl_tmp`
- Force and energy tables : `*.tab`
- VMD script for visualising the warmup : `warm_animation.script`
- VMD script for visualising the main trajectory : `vmd_animation.script`

To visualise your results using the vmd scripts you need to make sure that you have vmd installed properly and that you have the special vmd procedures used by the espresso team (i.e. support for the loadseries command). Then you can visualise by typing:

```
vmd -e vmd_animation.script
```

## 16.3. The `main.tcl` script

The `main.tcl` file provided in the `examples/scripts` directory is a relatively complete script written using mbtools. It is designed to run all of the examples provided but no more. No doubt you will need to extend it for your own purposes.

### 16.3.1. Variables used by `main.tcl`

`main.tcl` expects the user to set various parameters in a `parameters.tcl` file (e.g. `simplebilayer.tcl`). Some of these parameters have defaults and generally don't need to be worried about except for specific cases. In the following list variables that have no default and therefore must be set in the parameter file are noted with an asterisk.

- *thermo* [*Langevin*] The type of thermostat to be used. Set to *DPD* for a dpd thermostat. Any other value gives a langevin
- *dpd_gamma* Required if you set the thermo to *DPD*
- *dpd_r_cut* Required if you set the thermo to *DPD*
- *warmup_temp* [$*systemtemp*] The temperature at which the warmup is performed. The default behaviour is to use the system temperature
- *warmsteps* [100] Number of integrate steps per warmup cycle
- *warmtimes* [20] Number of calls to integrate over which the warmup occurs
- *free_warmsteps* [0] Warmup steps to be used for the warmup that occurs after particles are freed of any temporary constraints.
- *free_warmtimes* [0] Warmup times to be used for the warmup that occurs after particles are freed of any temporary constraints.

- *npt* [*off*] Whether to use the constant pressure barostat
- *p_ext* The pressure you want to simulate at. Required if npt is set to *on*
- *piston_mass* box mass. Required if npt is set to "on"
- *gamma_0* Required if npt is *on*. Usually set to 1 as for langevin gamma
- *gamma_v* Required if npt is *on*. Box friction
- *use_vmd* [*offline*] vmd mode
- *mgrid* [8] The number of meshpoints per side for dividing the bilayer plane into a grid
- *stray_cut_off* [1000.0] Distance of the end tail bead from the bilayer midplane beyond which a lipid is deemed to have strayed from the membrane bulk.
- **systemtemp* The temperature of the simulation during the main run
- **outputdir* Directory for output
- **tabledir* Directory where forcetables are kept
- **ident* a name for the simulation
- **topofile* the name of the file where the topology is written. Usually `$ident.top`
- **tablenames* A list of forcetable names to be used
- **setbox_l* Box dimensions
- **bonded_parms* A complete list of the bonded interactions required
- **nb_interactions* A complete list of the non-bonded interactions required
- **system_specs* A list of system specifications (see documentation for the `setup_system` command in 16.5)
- **moltypes* A list of molecule types (see documentation in 16.5)
- **warm_time_step* timestep to be used during warmup integration
- **main_time_step* timestep for the main integration run
- **verlet_skin* skin used for verlet nesting list criterion
- **langevin_gamma* langevin friction term
- **int_n_times* number of times to do main integration
- **int_steps* number of steps in each main integration

- **analysis_write_frequency* How often to calculate the analysis
- **write_frequency* How often to print out configurations
- *vmdcommands* a list of additional lines of commands to be written to the `vmd_animation.script` file

## 16.4. Analysis

The analysis package is designed to help organise the many possible analysis routines that can be performed during a simulation. This documentation describes the basic user interface commands and then all of the possible analysis routines. Instructions on how to add a new analysis routine are given at the end of this section.

### 16.4.1. Basic commands

The following commands comprise the user interface to the analysis package.

At the start of a simulation all of the analysis that is to be performed is specified using the `setup_analysis` command. Each time you want the analysis performed a call to `do_analysis` should be made. One can then call `print_averages` to write results to file.

```
::mbtools::analysis::setup_analysis : -outputdir.arg -suffix.arg
-iotype.arg -g.arg -str.arg
```

- *commands* [`./`] A tcl list where each element of the list is a string specifying the name and complete argument list for a particular analysis to be carried out.
- *outputdir* [`./`] The directory where analysis output files will be created
- *suffix* [`tmp`] Suffix that will be appended to standard file names for analysis output
- *iotype* [*a*] The iotype that will be used when opening files for analysis. For an explanation of the different iotypes see the documentation for the standard tcl command open
- *g* [8] Number of grid points per side with which to divide the bilayer for height profile analyses
- *str* [4.0] Distance of a tail bead from bilayer midplane beyond which a lipid is deemed to be a stray lipid.

Sets up the analysis package for a simulation run or analysis run that is about to be performed. This routine needs to be called before any analysis can be performed.

```
::mbtools::analysis::do_analysis :
```

Calls all of the `analyze` routines corresponding to commands setup in `setup_analysis`. `do_analysis` should be called only after `setup_analysis` has already been called.

```
::mbtools::analysis::reset_averages  :
```

Calls all of the `resetav` routines corresponding to commands setup in `setup_analysis`. These routines vary from command to command but they typically reset the storage and counter variables used for analysis results. `reset_averages` is typically only called internally by `print_averages`

```
::mbtools::analysis::print_averages :
```

Calls all of the `printav` routines corresponding to commands setup in `setup_analysis`. These routines typically print results to a file buffer. After printing the `reset_averages` routine is called internally. `print_averages` should be called only after `setup_analysis` has already been called.

### 16.4.2. Available analysis routines

```
boxl  :  -verbose  : output ||  time_vs_boxl
```

Simply obtains the box dimensions from ESPResSo.

```
clusters : -alipid.arg -verbose : output || time_vs_clust,
           sizehisto.[format %05d $time]
```

- alipid [1.29] Value for the area per lipid to be used in a making a rough calculation of the area of clusters

Calls the espresso command `analyze aggregation` which groups molecules in the system into aggregates. Output to `time_vs_clust` is: maximum cluster size, minimum cluster size, average size of clusters including those of size 2 or greater, standard deviation of clusters including those of size 2 or greater, number of clusters of size 2 or greater, total average cluster size, total cluster size standard deviation, total number of clusters, length of the interface between clusters, standard deviation of the interface length, number of clusters for which length was calculate.

Additionally, at each call of `print_averages` the complete size histogram is printed to a file with the formatted name `sizehisto.[format %05d $time]`.

```
density_profile  :  -nbins.arg -hrange.arg -beadtypes.arg
                   -colloidmoltypes.arg -r.arg -nogrid
                   -verbose  : output ||  av_zprof
```

- *nbins* [100] Number of slices into which the height range is divided for the purpose of calculating densities

- *hrange* [6] The maximum vertical distance from the bilayer midplane for which to calculate densities. Note that the complete vertical range is therefore 2*varhrange
- *beadtypes* [0] A tcl list of the bead types for which to calculate a density profile
- *colloidmoltypes* [] A tcl list of molecule types identifying the molecules which are colloids in the system. The default value is a null list
- *r* [0] A tcl list of sphere radii corresponding to the radii for each colloid type in the system. If this is non-zero the density profile will be calculated in spherical shells about the colloids in the system identified via colloidmoltypes or if colloidmoltypes is not set then the system center of mass is assumed for the colloid/vesicle center
- *nogrid* If this is set a grid mesh will not be used to refine the density profile calculation by taking into account vertical differences between mesh points

Calculates the number density of each of the beadtypes given in beadtypes as a function of the vertical distance from the bilayer midplane. Lipids are also sorted according to their orientation and assigned to upper or lower leaflets accordingly. Thus for a system with 3 beadtypes we would obtain 6 columns of output corresponding to 0 (lower) 1 (lower) 2 (lower) 2 (upper) 1 (upper) 0 (upper) where the number refers to the bead type and upper or lower refers to the bilayer leaflet.

```
energy : -verbose : output || time_vs_energy
```

Obtains the internal energies of the system from the `analyze energy` command of ESPResSo.

```
flipflop : -verbose : output || time_vs_flip
```

Makes a call to the `analyze get_lipid_orients` command of ESPResSo and compares this with a reference set of lipid orients obtained at the start of the simulation with `setup_analysis`. Based on this comparison the number of lipids which have flipped from their original positions is calculated

```
fluctuations : -verbose : output || powav.dat
```

Routine for calculating the power spectrum of height and thickness fluctuations for a flat bilayer sheet. Uses the `modes_2d` routine in ESPResSo to calculate the height and thickness functions and perform the fft. See the documentation in the file `fluctuations.tcl` for detail on what is calculated and how to obtain a stiffness value from the resulting output. Note that this routine causes a crash if it detects a large hole in the bilayer.

```
localheights : -range.arg -nbins.arg -rcatch.arg -verbose :
               output || av_localh
```

- *range* [1.0] Range of local height deviations over which to bin

- *nbins* [100] Number of slices to divide up the height range into for the purposes of creating a profile
- *rcatch* [1.9] The distance about a single lipid to use a starting value for finding the 6 closest neighbours

For each lipid we calculate its 6 nearest neighbours and then calculate the height difference between the central lipid and these neighbours. Taking these 6 values for each lipid we then create a histogram of number densities as a function of the height difference.

```
localorients : -range.arg -nbins.arg -verbose : output || av_localo
```

- range [1.0] Range of orientation deviations to consider
- nbins [100] Number of bins to use for histogram

Calculates the projection of the lipid orientation vector onto the $xy$ plane for each lipid and then bins the absolute values of these vectors.

```
orient_order : -verbose : output || time_vs_oop
```

Calculates the orientational order parameter $S$ for each lipid through a call to the espresso command `analyze lipid_orient_order`.

```
stress_tensor : -verbose : output || time_vs_stress_tensor
```

Calculates all 9 elements of the pressure tensor for the system through a call to the espresso command `analyze stress_tensor`

```
pressure : -verbose : output || time_vs_pressure
```

Calculates the isotropic pressure through a call to `analyze pressure`. Results are printed as a list of the various contributions in the following order: *p_inst*, *total*, *ideal*, *FENE*, *harmonic*, *nonbonded*. Where *p_inst* is the instantaneous pressure obtained directly from the barostat.

```
stray : -verbose : output || time_vs_stray
```

Calculates the number of stray lipids based on a call to `analyze get_lipid_orients`.

### 16.4.3. Adding a new routine

To add a new analysis routine you should create a new file called `myanalysis.tcl` which will contain all of your code. At the top of this file you should declare a namespace for your analysis code and include all of the internal variables inside that namespace as follows;

```
namespace eval ::mbtools::analysis::myanalysis {
    variable av_myresult
    variable av_myresult_i
    variable f_tvsresult
    variable verbose

    namespace export setup_myanalysis
    namespace export analyze_myanalysis
    namespace export printav_myanalysis
    namespace export resetav_myanalisis
}
```

Import your new file into the analysis package by adding a line like the following to the `analysis.tcl` file.

```
source [file join [file dirname [info script]] myanalysis.tcl]
```

You then need to implement the following essential functions within your new namespace.

- `::mbtools::analysis::myanalysis::setup_myanalysis { args }`

  Typically you would use this function to initialise variables and open files.

  Called by `::mbtools::analysis::setup_analysis`. Arguments are allowed.

- `::mbtools::analysis::myanalysis::printav_myanalysis { void }`

  This function should print results to a file.

  Called by `::mbtools::analysis::print_averages`. Arguments are not allowed.

- `::mbtools::analysis::myanalysis::analyze_myanalysis { void }`

  This function performs the actual analysis and should update the storage and averaging variables. Called by `::mbtools::analysis::do_analysis`. Arguments are not allowed.

- `::mbtools::analysis::myanalysis::resetav_myanalysis { void }`

  This function should update averages and reset variables accordingly depending on your requirements.

  Called by `::mbtools::analysis::reset_averages`. Arguments are not allowed.

If any of these functions is not implemented the program will probably crash.

## 16.5. System generation

Package for setting up lipid membrane systems in a variety of geometrical shapes.

### 16.5.1. Basic commands

```
::mbtools::system_generation::setup_system  :  [system_specs]
          [iboxl] [moltypes]
```

- `system_specs` This is a list of structures called system specifications. Each such system specification in turn should be a list consisting of a geometry and a list detailing the number of each molecule type i.e.

  ```
  set system_spec { geometry n_molslist }
  ```

  The *geometry* should be specified as a list with two elements. The first element should be a string "geometry" identifying this list as a geometry. The second element is a string containing the name of a geometry type *mygeometry* followed by arguments to be passed to the routine `create_mygeometry`.

  The *n_molslist* should be specified as a list with two elements. The first element should be a string "n_molslist" identifying this list as an n_molslist. The second element is a list each element of which specifies a molecule type and the number of such molecules.

- *boxl* A list containing the lengths of each of the box side lengths.

- *moltypes* A list, each element of which specifies a molecule type and type information. The exact format and requirements of this list are detailed for each molecule separately (see below for a list of molecule types and their requirements) however regardless of mol type the first two elements of the list must be a *moltypeid* and a string specifying the moltype respectively.

Sets up the system including generating topologies and placing molecules into specified geometries. Each geometry and list of molecules to be placed into that geometry are grouped into a system spec.

Example:

The following code sets out the molecule types to be used in the simulation by setting a list called *moltypes*. In this case two different lipid types are setup and assigned to moltypeids 0 and 1 respectively. Moltype 0 will consist of three beads per lipid, the first of which is of atomtype 0 and the second and third of which are of atomtype 1. Bonds in the lipid will be of type 0 and 1 respectively.(see the `::mbtools::system_generation::place_lipid_linear` function for further details).

```
set moltypes [list { 0 lipid { 0 1 1 } { 0 1 } }
                   { 1 lipid { 0 2 2 2 } { 0 2 } } ]
```

We then construct system specs for a flat bilayer and a spherical bilayer and group these into a *system_specs* list.

First the spherical *system_specs*

```
set geometry { geometry  "sphere -shuffle -c { 0.0 0.0 15.0 } " }
set n_molslist { n_molslist {  { 0 1000 } } }
lappend spherespec $geometry
lappend spherespec $n_molslist
```

The flat system_spec

```
set geometry { geometry "flat -fixz" }
set n_molslist { n_molslist {  { 1 3000 } } }
lappend bilayerspec $geometry
lappend bilayerspec $n_molslist
```

Now group together the *system*$_s$*pecs* into a master list

```
lappend system_specs $spherespec
lappend system_specs $bilayerspec
```

Make the call to `setup_system`

```
::mbtools::system_generation::setup_system $system_specs
         [setmd box_l] $moltypes

::mbtools::system_generation::get_trappedmols :
```

returns the internal list variable *trappedmols* which keeps track of all molecules that have been trapped by their center of mass. This function should be called after setup and would then typically be passed to the function `::mbtools::utils:trap_mols`.

```
::mbtools::system_generation::get_userfixedparts :
```

returns the internal list variable *userfixedparts* which keeps track of all particles that have been fixed in position during the setup. This is useful for later releasing particles after warmup routines have been completed.

```
::mbtools::system_generation::get_middlebead :
```

returns the internal variable *middlebead*.

### 16.5.2. Available geometries

```
flat  :  -fixz -bondl.arg -crystal -half -pancake -shuffle
```

- *fixz* Fix the vertical positions of all particles. The ids of these particles are added to the list of *userfixedparts* which can later be obtained through a call to `::mbtools::system_generation::get_userfixedparts`.
- *crystal* Sets lipids on a grid, instead of randomly.

- *half* Creates a halfbilayer (i.e. periodic only along one direction). Useful to measure a line tension.
- *pancake* Creates a spherical and flat bilayer. The diameter of the pancake cannot exceed the box_l.
- *shuffle* shuffle the topology prior to placing the lipids. This is required for a random lipid distribution because otherwise the lipids will be placed on the sphere in the order they appear in the topology

Creates a flat bilayer in the XY plane by random placement of lipids.

`sphere : -c.arg -initarea.arg -bondl.arg -shuffle`

- *c* [{`0.0 0.0 0.0`}] The location of the center of the sphere relative to the center of the box
- *initarea* [1.29] An initial guess for the area per lipid. This guess is used to compute initial sphere dimensions based on the number of lipids. This initial guess is then iteratively refined until all lipids can be fit uniformly on the sphere.
- *shuffle* shuffle the topology prior to placing the lipids. This is required for a random lipid distribution because otherwise the lipids will be placed on the sphere in the order they appear in the topology

Creates a spherical vesicle by placing molecules in an ordered manner at uniform density on the surface of the sphere. Molecules are assumed to have a uniform cross sectional area and closely matched (though not identical) lengths. The radius of the vesicle will depend on the number of lipids and the area per lipid.

`sphere_cap : -r.arg -half -c.arg -initarea.arg -bondl.arg -shuffle`

- *r* [10.0] The radius of the whole sphere where the cap is shaped
- *half* Create a half of sphere with the amount of molecules available
- *c* [{`0.0 0.0 0.0`}] The location of the center of the sphere relative to the center of the box
- *initarea* [1.29] An initial guess for the area per lipid. This guess is used to compute initial sphere dimensions based on the number of lipids. This initial guess is then iteratively refined until all lipids can be fit uniformly on the sphere.
- *shuffle* shuffle the topology prior to placing the lipids. This is required for a random lipid distribution because otherwise the lipids will be placed on the sphere in the order they appear in the topology

Creates a spherical cap which is part of a vesicle of a radius $r$, by placing molecules in an ordered manner at uniform density on the surface of the sphere. Molecules are assumed to have a uniform cross sectional area and closely matched (though not identical) lengths. If the option *half* is defined, the radius of the vesicle will depend on the number of lipids and the area per lipid.

`torus : -c.arg -initarea.arg -ratio.arg -bondl.arg -shuffle`

- *c* [{`0.0 0.0 0.0`}] The location of the center of the torus relative to the center of the box.
- *initarea* [1.29] An initial guess for the area per lipid. This guess is used to compute initial radii based on the number of lipids. This initial guess is then iteratively refined until all lipids can be fit uniformly on the torus.
- *ratio* [1.4142] Ratio of major toroidal radius to minor toroidal radius. Default value is for the Clifford torus.
- *shuffle* shuffle the topology prior to placing the lipids. This is required for a random lipid distribution because otherwise the lipids will be placed on the torus in the order they appear in the topology.

Creates a toroidal vesicle by placing molecules in an ordered manner at uniform density on the surface of the torus. Molecules are assumed to have a uniform cross sectional area and closely matched (though not identical) lengths. The two radii of the torus will depend on the number of lipids, the area per lipid and the ratio between radii.

`cylinder : -c.arg -initarea.arg -bondl.arg -shuffle`

- *c* [0.0 0.0 0.0]
- *initarea* [1.29]
- *shuffle* shuffle the topology prior to placing the lipids.

Creates a cylinder which spans the box along one dimension by placing molecules uniformly on its surface. Works in a similar way to the sphere routine.

`random : -exclude.arg -inside.arg -shuffle -bondl.arg`

- *exclude.arg* [] an exclusion zone definition suitable for passing to `::mbtools::utils::isoutside`.
- *inside.arg* [] an inclusion zone definition suitable for passing to `::mbtools::utils::isoutside`.
- *shuffle* shuffle the topology prior to placing the lipids.

Places molecules randomly in space with a (sortof) random orientation vector. If an exclusion zone is defined, then no molecules will be placed such that their positions are within the zone. If an inclusion zone if defined, then no molecules will be place outside this zone. For instance,

```
set geometry { geometry "random -exclude { sphere { 0.0 0.0 0.0 } 4.0 }
              -inside { cuboid { 0.0 0.0 0.0 } { 15.0 15.0 15.0 } }" }
```

will randomly place molecules within the volume between a sphere with a radius of 4.0 and a cuboid with dimension $15.0 \times 15.0 \times 15.0$ at the origin.

```
readfile : -ignore.arg -f.arg -t.arg
```

- *ignore.arg* [] particle properties to be ignored during the file read.
- *f.arg* [] The file containing the configuration to be used for setup. Must be an espresso blockfile with box length, particle and bonding information.
- *t.arg* [] The topology file corresponding to the file to be read.
- *tol.arg* [0.000001] Tolerance for comparison of box dimensions.

Use particle positions contained in a file to initialise the locations of particles for a particular geometry. The box dimensions in the file and those set by the user are compared and an error is returned if they are not the same to within a tolerance value of *tol*. Even though we read from a file we also generate a topology from the $n_m$ *olslist* and this topology is compared with the topology that is read in to check if the number of particles are the same.

```
singlemol : -c.arg -o.arg -trapflag.arg -ctrap.arg
            -trapspring.arg -bondl.arg
```

- *c.arg* [ 0.0 0.0 0.0 ] The molecule center. Exactly what this means depends on the molecule type.
- *o.arg* [ 0.0 0.0 1.0 ] The orientation vector for the molecule. This is also molecule type dependent
- *trapflag.arg* [ 0 0 0 ] Set this optional argument to cause a molecule to be trapped by its center of mass. You should give three integers corresponding to each of the three coordinate axes. If a value of 1 is given then motion in that axis is trapped.
- *ctrap.arg* [ "" ] Set this optional argument to the central point of the trap. This works much like an optical trap in that molecules will be attracted to this point via a simple harmonic spring force
- *trapspring.arg* [ 20 ] The spring constant for the trap potential (harmonic spring).

Simply place a single molecule at the desired position with the desired orientation.

### 16.5.3. Adding a new geometry

To create a routine for setting up a system with a new type of geometry *mygeom*. Start by creating a new file `mygeom.tcl` inside the `system_generation` directory. The new file should declare a new namespace *mygeom* as a sub namespace of `::mbtools::system_generation` and export the proceedure `create_mygeom`. Thus your `mygeom.tcl` file should begin with the lines

```
namespace eval ::mbtools::system_generation::mygeom {
  namespace export create_mygeom
}
```

Import your new file into the `system_generation` package by adding a line like the following to the `system_generation.tcl` file

```
source [file join [file dirname [info script]] mygeom.tcl]
```

You then need to implement the `create_mygeom` proceedure within your new namespace as follows

```
::mbtools::system_generation::mygeom::create_mygeom  args
```

### 16.5.4. Available molecule types

```
lipid  : typeinfo :  { moltypeid "lipid" particletypelist
         bondtypelist }
```

- *particletypelist* A list of the particle types for each atom in the lipid. The particles are placed in the order in which they appear in this list.
- *bondtypelist* A list of two *bondtypeid*s. The first id is used for bonds between consecutive beads in the lipid. The second *bondtypeid* defines the pseudo bending potential which is a two body bond acting across beads separated by exactly one bead.

Places atoms in a line to create a lipid molecule.

```
hollowsphere : typeinfo : { moltypeid "hollowsphere"
               sphereparticlelist bondtype natomsfill }
```

- *sphereparticlelist* A list of the particle types for each atom in the hollowsphere. The atoms that make up the outer shell must be listed first followed by the atoms that make up the inner filling.
- *bondtype* The typeid for bonds linking atoms in the outer shell.
- *natomsfill* Number of filler atoms. The atom types for these will be obtained from the last *natomsfill* in the *sphereparticlelist*.

Creates a sphere of beads arranged such that they have an approximate spacing of *bondl* and such that they optimally cover the sphere. The optimal covering is obtained using the `icover` routines which are copyright R. H. Hardin, N. J. A. Sloane and W. D. Smith, 1994, 2000. Thus the routine will only work if you have installed icover and if you can successfully run it from the command line in the directory that you started your espresso job. These routines are serious overkill so if anybody can think of a nice simple algorithm for generating a covering of the sphere let us know.

```
protein : typeinfo : { moltypeid "protein" particletypelist
          bondtypelist }
```

- *particletypelist* A list of the particle types for each atom in the protein.
- *bondtypelist* A list of bondtypeids.

Create a protein molecule.

```
spanlipid : typeinfo : { moltypeid "protein" particletypelist
bondtypelist }
```

- *particletypelist* A list of the particle types for each atom in the lipid. Since this is a spanning lipid the first and last elements of this list would typically be head beads.
- *bondtypelist* A list of two *bondtypeid*s with the same meaning as explained above for standard lipids.

Create a lipid which spans across the bilayer.

### 16.5.5. Adding a new molecule type

To add a new molecule type you need to define a proceedure which determines how the atoms that make up the molecule should be placed. This proc will live directly in the `::mbtools::system_generation` namespace. Examples can be found in `place.tcl`.

In order to register your new molecule type to allow placement in any geometry you need to add a call to it in the function `::mbtools::system_generation::placemol`. Make sure that all arguments to your `place_mymolecule` routine are included in this function call.

## 16.6. Utils

Useful utilities routines for various types. Includes file management, basic geometry and math procedures.

### 16.6.1. Setup commands

```
::mbtools::utils::setup_outputdir  :  [outputdir] -paramsfile.arg
          -tabdir.arg -tabnames.arg -startf.arg -ntabs.arg
```

- *outputdir* Complete path of the directory to be setup. At least the parent of the directory must exist
- *paramfile* [] Name of a file to be copied to the output directory
- *tabdir* [] Full path name of the directory where forcetables are kept
- *tabnames* [] Complete list of forcetables to be used in the simulation. These will be copied to the output directory

This routine is designed to setup a directory for simulation output. It copies forcetables and the parameter file to the directory after creating it if necessary.

```
::mbtools::utils::read_startfile : [file]
```

- *file* Complete path of the file to be read. Should be an espresso blockfile.

Read in particle configuration from an existing file or simulation snapshot

```
::mbtools::utils::read_checkpoint : [dir]
```

- *dir* Directory containing the checkpoint file which must be called `checkpoint.latest.gz`.

Read in a checkpoint and check for success. Warn if the checkpoint does not exist.

```
::mbtools::utils::read_topology : [file]
```

- *file* Complete path of the file that contains the topology information.

Read in the topology from a file and then execute the `analyze set "topo_part_sync"` command of ESPResSo.

```
::mbtools::utils::set_topology : [topo]
```

- *topo* A valid topology.

Set the given topology and then execute the `analyze set "topo_part_sync"` command of ESPResSo.

```
::mbtools::utils::set_bonded_interactions : [bonded_parms]
```

- $bonded_p arms$ A list of bonded interactions. Each element of this list should contain all the appropriate arguments in their correct order for a particular call to the espresso `inter` command. See the espresso `inter` command for a list of possible bonded interactions and correct syntax.

Set all the bonded interactions.

```
::mbtools::utils::set_nb_interactions : [nb_parms]
```

- *nb_parms* A list of interactions. Each element of this list should contain all the appropriate arguments in their correct order for a particular call to the espresso `inter` command. See the espresso `inter` command for a list of possible non-bonded interactions and correct syntax.

Set all the bonded interactions.

```
::mbtools::utils::init_random : [n_procs]
```

- *n_procs* The number of processors used in this job.

Initialize the random number generators on each processor based on the current time with a fixed increment to the time seed used for each proc.

```
::mbtools::utils::initialize_vmd : [flag] [outputdir]
          [ident] -extracommands.arg
```

- *flag* Depending on the value of this parameter initialize vmd to one of its possible states:
  - interactive : VMD is started and a connection to espresso established for immediate viewing of the current espresso process. With some luck this might even work sometimes! If VMD doesn't get a proper connection to espresso then it will crash.
  - offline : Just constructs the appropriate `psf` and `vmd_animation.script` files and writes them to the output directory so that `pdb` files generated with writepdb can be viewed with `vmd -e vmd_animation.script`.
  - default : Any value other than those above for flag will just result in vmd not being initialized.
- *outputdir* The directory where vmd output will be written.
- *ident* A basename to be be given to vmd files.
- *extracommands* [] A list of strings each of which will be written to the end of the `vmd_animationscript`. Use this to give additional commands to vmd.

Prepare for vmd output.

### 16.6.2. Warmup commands

```
::mbtools::utils::warmup  :  [steps] [times] -mindist.arg
          -cfgs.arg -outputdir.arg -vmdflag.arg -startcap.arg
          -capgoal.arg
```

- *steps* number of integration steps used in each call to integrate.
- *times* number of times to call the integrate function during warmup.
- *mindist* [0] Terminate the warmup when the minimum particle distance is greater than this criterion. A value of 0 (default) results in this condition being ignored. If a condition is imposed this routine can become very very slow for large systems.
- *cfgs* [-1] Write out a configuration file every cfgs calls to integrate.
- *outputdir* [./] The directory for writing output.
- *vmdflag* [offline] If this flag is set to "offline" (default) pdb files will be generated for each configuration file generated.
- *startcap* [5] Starting value for the forcecap.
- *capgoal* [1000] For the purposes of calculating a cap increment this value is used as a goal. The final forcecap will have this value.

Perform a series of integration steps while increasing forcecaps from an initially small value.

### 16.6.3. Topology procs

```
::mbtools::utils::maxpartid  :  [topo]
```

- *topo* A valid topology.

Find the maximum particle id in a given topology.

```
::mbtools::utils::maxmoltypeid : [topo]
```

- *topo* A valid topology.

Find the maximum molecule type id.

```
::mbtools::utils::listnmols : [topo]
```

- *topo* A valid topology.

Construct a list with the number of molecules of each molecule type.

```
::mbtools::utils::minpartid : [topo]
```

- *topo* A valid topology.

Minimum particle id for the given topology.

```
::mbtools::utils::minmoltype : [topo]
```

- *topo* A valid topology/

Minimum molecule type id for this topology.

```
::mbtools::utils::listmoltypes : [topo]
```

- *topo* A valid topology.

Make a list of all the molecule types in a topology. Makes a check for duplication which would occur for an unsorted topology.

```
::mbtools::utils::listmollengths : [topo]
```

- *topo* A valid topology.

Works out the length (number of atoms) of each molecule type and returns a list of these lengths.

### 16.6.4. Math procs

```
::mbtools::utils::dot_product  :  A B
```

Returns A dot B

```
::mbtools::utils::matrix_vec_multiply : A B
```

Return the product of a matrix A with a vector B

```
::mbtools::utils::calc_proportions : ilist
```

Calculate the number of times each integer occurs in the list ilist.

```
::mbtools::utils::average : data from to
```

- *data* A list of numbers to be averaged
- *from* Optional starting index in data
- *to* Optional ending index in data

Calculate the mean of a list of numbers starting from *from* going up to *to*.

```
::mbtools::utils::stdev : data from to
```

- *data* A list of numbers to find the std deviation of
- *from* Optional starting index in data
- *to* Optional ending index in data

Calculate the standard deviation of a list of numbers starting from *from* going up to *to*.

`::mbtools::utils::acorr : data`

- *data* Data for which an autocorrelation is to be calculated

Calculate an autocorrelation function on a set of data.

`::mbtools::utils::distance : pos1 pos2`

- *pos1* A position vector
- *pos2* A position vector

Calculate the distance between two points whose position vectors are given.

`::mbtools::utils::distance_min : pos1 pos2`

- *pos1* A position vector
- *pos2* A position vector

Calculate the minimum image distance between two position vectors.

`::mbtools::utils::min_vec : pos1 pos2`

- *pos1* A position vector
- *pos2* A position vector

Calculate the minimum image vector from position vector2 to postition 1, *i.e.* pos1 - pos2.

`::mbtools::utils::normalize : vec`

- *vec* The vector to be normalised

Normalize a vector

`::mbtools::utils::scalevec : vec scale`

- *vec* The vector to be scaled
- *scale* Scaling factor

Multiply all elements of a vector by a scaling factor

`::mbtools::utils::uniquelist : original`

- *original* A list possibly containing duplicate elements

Construct a list of all the unique elements in the original list removing all duplication.

### 16.6.5. Miscellaneous procs

`::mbtools::utils::trap_mols : molstotrap`

- *molstotrap* A list of trap values for molecules. This list would typically be obtained by calling `::mbtools::get_trappedmols` immediately after the system has been setup.

Set the trap value for a list of molecules.

`::mbtools::utils::isoutside : [pos] [zone]`

- *pos* The point whose status is to be determined
- *zone* This will be a tcl list. The first element of the list must be a string with the name of the zone type and subsequent elements will be further information about the zone. Available zones are:
  - *sphere* : center radius
  - *cuboid* : center {L W H}

Determines whether the point at *pos* is outside the zone. Parameter center should be a tcl list. Returns 1 if it is and 0 if it is not.

`::mbtools::utils::calc_com : mol`

- *mol* The molecule

Calculate the center of mass of a molecule.

`::mbtools::utils::centersofmass_bymoltype : [moltypes]`

- *moltypes* A list of molecule type ids

Determine the center of mass of every molecule whose type matches an item in the list moltypes. Returns a nested list where each element in the list is itself a list of centers of mass for a given moltype.

## 16.7. mmsg

mmsg is designed to provide a more controlled way of printing messages than the simple `puts` commands of Tcl. It has an ability to turn on or off messages from particular namespaces.

### 16.7.1. Basic commands

The following commands represent the standard interface for the `mmsg` package. For consistency one should use these instead of a bare puts to standard out. `mbtools` makes extensive use of these commands.

`::mmsg::send : [namespace] [string] { [newline] }`

- *namespace* A namespace. Typically this should be the current namespace which one can get via namespace current
- *string* The message you want printed
- *newline* [yes] Set this to anything other than "yes" and no carriage return will be used after the message

The mmsg equivalent of puts. Designed for printing of simple status or progress messages.

`::mmsg::err : [namespace] [string] { [newline] }`

- *namespace* A namespace. Typically this should be the current namespace which one can get via namespace current
- *string* The message you want printed
- *newline* [yes] Set this to anything other than "yes" and no carriage return will be used after the message

Prints error messages and causes program to exit.

`::mmsg::warn : [namespace] [string] { [newline] }`

- *namespace* A namespace. Typically this should be the current namespace which one can get via namespace current
- *string* The message you want printed
- *newline* [yes] Set this to anything other than "yes" and no carriage return will be used after the message

Prints warning messages.

`::mmsg::debug : [namespace] [string] { [newline] }`

- *namespace* A namespace. Typically this should be the current namespace which one can get via namespace current
- *string* The message you want printed
- *newline* [yes] Set this to anything other than "yes" and no carriage return will be used after the message

Prints debug messages.

### 16.7.2. Control commands

`mmsg` does several checks before it decides to print a message. For any given message type it checks if that message type is allowed. It also checks to see if the namespace given as an argument is in the allowable namespaces list. The default behaviour is to print from the main mbtools namespaces and the global namespace

```
{ :: ::mbtools::system_generation ::mbtools::utils ::mbtools::analysis }
```

Note that children of these namespaces must be explicitly enabled. All message types except debug are also enabled by default. The following commands allow this default behaviour to be changed.

`::mmsg::setnamespaces : namespacelist`

- *namespacelist* A list of all namespaces from which messages are to be printed

Allows control over which namespaces messages can be printed from.

`::mmsg::enable : type`

- *type* A string indicating a single message type to enable. Allowable values are "err", "debug", "send" and "warn"

Allows particular message types to be enabled: For example one could enable debug output with

```
mmsg::enable "debug"
```

`::mmsg::disable : type`

- *type* A string indicating a single message type to disable. Allowable values are "err", "debug", "send" and "warn"

Allows particular message types to be disabled: For example one could disable warning output with

```
mmsg::enable "warn"
```

# 17. Under the hood

- Implementation issues that are interesting for the user
- Main loop in pseudo code (for comparison)

## 17.1. Internal particle organization

Since basically all major parts of the main MD integration have to access the particle data, efficient access to the particle data is crucial for a fast MD code. Therefore the particle data needs some more elaborate organisation, which will be presented here. A particle itself is represented by a structure (Particle) consisting of several substructures (e. g. ParticlePosition, ParticleForce or ParticleProperties), which in turn represent basic physical properties such as position, force or charge. The particles are organised in one or more particle lists on each node, called Cell cells. The cells are arranged by several possible systems, the cellsystems as described above. A cell system defines a way the particles are stored in ESPResSo, i. e. how they are distributed onto the processor nodes and how they are organised on each of them. Moreover a cell system also defines procedures to efficiently calculate the force, energy and pressure for the short ranged interactions, since these can be heavily optimised depending on the cell system. For example, the domain decomposition cellsystem allows an order N interactions evaluation.

Technically, a cell is organised as a dynamically growing array, not as a list. This ensures that the data of all particles in a cell is stored contiguously in the memory. The particle data is accessed transparently through a set of methods common to all cell systems, which allocate the cells, add new particles, retrieve particle information and are responsible for communicating the particle data between the nodes. Therefore most portions of the code can access the particle data safely without direct knowledge of the currently used cell system. Only the force, energy and pressure loops are implemented separately for each cell model as explained above.

The domain decomposition or link cell algorithm is implemented in ESPResSo such that the cells equal the ESPResSo cells, i. e. each cell is a separate particle list. For an example let us assume that the simulation box has size $20 \times 20 \times 20$ and that we assign 2 processors to the simulation. Then each processor is responsible for the particles inside a $10 \times 20 \times 20$ box. If the maximal interaction range is 1.2, the minimal possible cell size is 1.25 for 8 cells along the first coordinate, allowing for a small skin of 0.05. If one chooses only 6 boxes in the first coordinate, the skin depth increases to 0.467. In this example we assume that the number of cells in the first coordinate was chosen to be 6 and that the cells are cubic. ESPResSo would then organise the cells on each node in a $6 \times 12 \times 12$ cell grid embedded at the centre of a $8 \times 14 \times 14$ grid. The additional

cells around the cells containing the particles represent the ghost shell in which the information of the ghost particles from the neighbouring nodes is stored. Therefore the particle information stored on each node resides in 1568 particle lists of which 864 cells contain particles assigned to the node, the rest contain information of particles from other nodes.a

Classically, the link cell algorithm is implemented differently. Instead of having separate particle lists for each cell, there is only one particle list per node, and a the cells actually only contain pointers into this particle list. This has the advantage that when particles are moved from one cell to another on the same processor, only the pointers have to be updated, which is much less data (4 rsp. 8 bytes) than the full particle structure (around 192 bytes, depending on the features compiled in). The data storage scheme of ESPResSo however requires to always move the full particle data. Nevertheless, from our experience, the second approach is 2-3 times faster than the classical one.

To understand this, one has to know a little bit about the architecture of modern computers. Most modern processors have a clock frequency above 1GHz and are able to execute nearly one instruction per clock tick. In contrast to this, the memory runs at a clock speed around 200MHz. Modern double data rate (DDR) RAM transfers up to 3.2GB/s at this clock speed (at each edge of the clock signal 8 bytes are transferred). But in addition to the data transfer speed, DDR RAM has some latency for fetching the data, which can be up to 50ns in the worst case. Memory is organised internally in pages or rows of typically 8KB size. The full $2 \times 200$ MHz data rate can only be achieved if the access is within the same memory page (page hit), otherwise some latency has to be added (page miss). The actual latency depends on some other aspects of the memory organisation which will not be discussed here, but the penalty is at least 10ns, resulting in an effective memory transfer rate of only 800MB/s. To remedy this, modern processors have a small amount of low latency memory directly attached to the processor, the cache.

The processor cache is organised in different levels. The level 1 (L1) cache is built directly into the processor core, has no latency and delivers the data immediately on demand, but has only a small size of around 128KB. This is important since modern processors can issue several simple operations such as additions simultaneously. The L2 cache is larger, typically around 1MB, but is located outside the processor core and delivers data at the processor clock rate or some fraction of it.

In a typical implementation of the link cell scheme the order of the particles is fairly random, determined e. g. by the order in which the particles are set up or have been communicated across the processor boundaries. The force loop therefore accesses the particle array in arbitrary order, resulting in a lot of unfavourable page misses. In the memory organisation of ESPResSo, the particles are accessed in a virtually linear order. Because the force calculation goes through the cells in a linear fashion, all accesses to a single cell occur close in time, for the force calculation of the cell itself as well as for its neighbours. Using the domain decomposition cell scheme, two cell layers have to be kept in the processor cache. For 10000 particles and a typical cell grid size of 20, these two cell layers consume roughly 200 KBytes, which nearly fits into the L2 cache. Therefore every cell has to be read from the main memory only once per force calculation.

# 18. Getting involved

Up to date information about the development of ESPResSo can be found at the web page `http://espressomd.org` As the important information can change in time, we will not describe its contents in detail but rather request the reader to go directly to the URL. Among other things, one can find information about the following topics there:

- FAQ
- Latest stable release of ESPResSo and older releases
- Obtaining development version of ESPResSo
- Archives of both developers' and users' mailing lists
- Registering to ESPResSo mailing lists
- Submitting a bug report

## 18.1. Community support and mailing lists

If you have any questions concerning ESPResSo which you cannot resolve by yourself, you may post a message to the mailing list. Instructions on how to register to the mailing lists and post messages can be found on the homepage `http://espressomd.org`. Before posting a question and waiting for someone to answer, it may be useful to search the mailing list archives or FAQ and see if you can get the answer immediately. For several reasons it is recommended to send all questions to the mailing lists rather than to contact individual developers:

- All registered users get your message and you have a higher probability that it is answered soon.
- Your question and the answers are archived and the archives can be searched by others.
- The answer may be useful also to other registered users.
- There may not be a unique answer to your problem and it may be useful to get suggestions from different people.

Please remember that this is a community mailing list. It is other users and developers who are answering your questions. They do it in their free time and are not paid for doing it.

## 18.2. Contributing your own code

If you are planning to make an extension to ESPResSo or already have a piece of your own code which could be useful to others, you are very welcome to contribute it to the community. Before you start making any changes to the code, you should obtain the current development version of it. For more information about how to obtain the development version, refer to the homepage `http://espressomd.org`.

It is also generally a good idea to contact the mailing lists before you start major coding projects. It might be that someone else is already working on the problem or has a solution at hand.

## 18.3. Developers' guide

Besides the User guide, ESPResSo also contains a Developers' guide which is a programmer documentation automatically built from comments in the source code and using Doxygen. It provides a cross-referenced documentation of all functions and data structures available in ESPResSo source code. It can be built by typing

```
make dg
```

in the build directory. Afterwards it can be found in the subdirectory of the build directory: `doc/dg/html/index.html`.

A recent version of this guide can also be found on the ESPResSo homepage `http://espressomd.org`.

## 18.4. User's guide

If, while reading this user guide, you notice any mistakes or badly (if at all) described features or commands, you are very welcome to contribute to the guide and have others benefit from your knowledge.

For this, you should also checkout the development version as described on the homepage. As the user guide, like all ESPResSo code, is always in flow and changes are made regularly, there are already many paragraphs marked with a "todo" box. To turn on these boxes, edit the main file `doc/ug/ug.tex` and adapt the inclusion of the LaTeX package `todonotes`.

You can then build the user guide by typing

```
make ug
```

# A. ESPResSo quick reference

| | |
|---|---|
| `part` *pid* [`pos` *x y z*] [`type` *typeid*] [`v` *vx vy vz*] [`f` *fx fy fz*] [`bond` *bondid pid2 ...*] [`q` *charge*] [1] [`quat` *q1 q2 q3 q4*] [2] [`omega_body/lab` *x y z*] [2] [`torque_body/lab` *x y z*] [2] [`rinertia` *x y z*] [2] [[`un`]`fix` *x y z*] [3] [`ext_force` *x y z*] [3] [`ext_torque` *x y z*] [2,3] [`exclude` *pid2...*] [4] [`exclude delete` *pid2...*] [4] [`mass` *mass*] [5] [`dipm` *moment*] [6] [`dip` *dx dy dz*] [6] [`virtual` *v*] [7,8] [`vs_relative` *pid* `distance`] [8] [`vs_auto_relate_to` *pid*] [8] [`temp` *T*] [9] [`gamma` *g*] [9] [`gamma_rot` *grot*] [2,9] [`rotation` *rot*] [10] [`solvation` *lA kA lB kB*] [11] [`swimming` ( ( `v_swim` *v_swim* \| `f_swim` *f_swim* ) \| `off` )] [12] [`swimming` ( ( `v_swim` *v_swim* \| `f_swim` *f_swim* ) ( `pusher` \| `puller` ) `dipole_length` *dipole_length* `rotational_friction` *rotational_friction* \| `off` )] [12,13]<br>Required features: [1] `ELECTROSTATICS` [2] `ROTATION` [3] `EXTERNAL_FORCES` [4] `EXCLUSION` [5] `MASS` [6] `DIPOLES` [7] `VIRTUAL_SITES_COM` [8] `VIRTUAL_SITES_RELATIVE` [9] `LANGEVIN_PER_PARTICLE` [10] `ROTATION_PER_PARTICLE` [11] `SHANCHEN` [12] `ENGINE` [13] `LB` or `LB_GPU` | 33 |
| `part` *pid* `print` [( `id` \| `pos` \| `type` \| `folded_position` \| `type` \| `q` \| `v` \| `f` \| `torque_body` \| `torque_lab` \| `body_frame_velocity` \| `fix` \| `ext_force` \| `ext_torque` \| `bond` \| `exclusions connections` [*range*] \| `swimming` )]...<br>`part` | 37 |
| `part` *pid* `delete`<br>`part deleteall` | 38 |
| `part auto_exclusions` [*range*]<br>`part delete_exclusions`<br>Required features: `EXCLUSIONS` | 38 |
| `polymer` *num_polymers monomers_per_chain bond_length* [`start` *pid*] [`pos` *x y z*] [`mode` ( `RW` \| `SAW` \| `PSAW` ) [*shield* [$try_{\max}$]]] [`charge` *q*] [1] [`distance` $d_{\text{charged}}$] [1] [`types` $typeid_{\text{neutral}}$ [$typeid_{\text{charged}}$]] [`bond` *bondid*] [`angle` $\phi$ [$\theta$ [*x y z*]]] [`constraints`] [2]<br>Required features: [1] `ELECTROSTATICS` [2] `CONSTRAINTS` | 39 |
| `counterions` *N* [`start` *pid*] [`mode` ( `SAW` \| `RW` ) [*shield* [$try_{\max}$ ]]] [`charge` *val*] [1] [`type` *typeid*]<br>Required features: [1] `ELECTROSTATICS` | 41 |

`salt` $N_+$ $N_-$ [`start` *pid*] [`mode` ( `SAW` | `RW` ) [*shield* [$try_{\max}$]]] 41
[`charges` $val_+$ [$val_-$]] [1] [`types` $typeid_+$ [$typeid_-$]] [`rad` $r$]
Required features: [1] `ELECTROSTATICS`

`diamond` *a bond_length monomers_per_chain* [`counterions` $N_{\mathrm{CI}}$] 42
[`charges` $val_{\mathrm{node}}$ $val_{\mathrm{monomer}}$ $val_{\mathrm{CI}}$] [1] [`distance` $d_{\mathrm{charged}}$] [1] [`nonet`]
Required features: [1] `ELECTROSTATICS`

`icosaeder` *a monomers_per_chain* [`counterions` $N_{\mathrm{CI}}$] 43
[`charges` $val_{\mathrm{monomers}}$ $val_{\mathrm{CI}}$] [1] [`distance` $d_{\mathrm{charged}}$] [1]
Required features: [1] `ELECTROSTATICS`

`crosslink` *num_polymer monomers_per_chain* [`start` *pid*] [`catch` $r_{\mathrm{catch}}$] 44
[`distLink` *link_dist*] [`distChain` *chain_dist*] [`FENE` *bondid*]
[`trials` $try_{\max}$]

`copy_particles` [`set` *id1 id2* `...`| `range` *from* `to` `...`] [`shift` $s_x$ $s_y$ $s_z$] 45

| | |
|---|---|
| `constraint wall normal` $n_x$ $n_y$ $n_z$ `dist` $d$ `type` $id$ [`penetrable` *flag*] [`reflecting` *flag*] [`only_positive` *flag*] [`tunable_slip` *flag*] | 46 |
| `constraint sphere center` $c_x$ $c_y$ $c_z$ `radius` *rad* `direction` *direction* `type` *id* [`penetrable` *flag*] [`reflecting` *flag*] | |
| `constraint cylinder center` $c_x$ $c_y$ $c_z$ `axis` $n_x$ $n_y$ $n_z$ `radius` *rad* `length` *length* `direction` *direction* `type` *id* [`penetrable` *flag*] [`reflecting` *flag*] | |
| `constraint rhomboid corner` $p_x$ $p_y$ $p_z$ `a` $a_x$ $a_y$ $a_z$ `b` $b_x$ $b_y$ $b_z$ `c` $c_x$ $c_y$ $c_z$ `direction` *direction* `type` *id* [`penetrable` *flag*] [`reflecting` *flag*] | |
| `constraint maze nsphere` $n$ `dim` $d$ `sphrad` $r_s$ `cylrad` $r_c$ `type` *id* [`penetrable` *flag*] | |
| `constraint pore center` $c_x$ $c_y$ $c_z$ `axis` $n_x$ $n_y$ $n_z$ `radius` *rad* [`outer_radius` $rad_S$] [`smoothing_radius` $rad_S$] `length` *length* `type` *id* | |
| `constraint stomatocyte center` $x$ $y$ $z$ `orientation` $ox$ $oy$ $oz$ `outer_radius` $Ro$ `inner_radius` $Ri$ `layer_width` $w$ `direction` *direction* `type` *id* [`penetrable` *flag*] [`reflecting` *flag*] | |
| `constraint slitpore pore_mouth` $z$ `channel_width` $c$ `pore_width` $w$ `pore_length` $l$ `upper_smoothing_radius` $us$ `lower_smoothing_radius` $ls$ | |
| `constraint rod center` $c_x$ $c_y$ `lambda` *lambda* [1] | |
| `constraint plate height` $h$ `sigma` *sigma* [1] | |
| `constraint ext_magn_field` $f_x$ $f_y$ $f_z$ [2,3] | |
| `constraint plane cell` $x$ $y$ $z$ `type` *id* | |
| `constraint mindist_position` $x$ $y$ $z$ | |
| `constraint hollow_cone center` $x$ $y$ $z$ `orientation` $ox$ $oy$ $oz$ `outer_radius` $Ro$ `inner_radius` $Ri$ `width` $w$ `opening_angle` *alpha* `direction` *direction* `type` *id* [`penetrable` *flag*] [`reflecting` *flag*] | |
| `constraint spherocylinder center` $c_x$ $c_y$ $c_z$ `axis` $n_x$ $n_y$ $n_z$ `radius` *rad* `length` *length* `direction` *direction* `type` *id* [`penetrable` *flag*] [`reflecting` *flag*] | |
| `constraint mindist_position` $x$ $y$ $z$ | |
| Required features: `CONSTRAINTS` [1] `ELECTROSTATICS` [2] `ROTATION` [3] `DIPOLES` | |
| `constraint delete` [*num*] | 49 |
| `constraint force` $n$ | 50 |
| `constraint` [*num*] | 50 |
| `harmonic_well` { $x$ $y$ $z$ } $k$ | 50 |
| Required features: `CUDA` | |
| `part gc` ( *type* \| ( ( `find` \| `delete` \| `status` \| `number` ) *type* ) ) | 53 |
| `part gc` *type* | 54 |
| `inter` | 55 |

| | |
|---|---|
| `inter` *type1 type2* `tabulated` *filename* | 55 |
| Required features: `TABULATED` | |
| `inter` *type1 type2* `lennard-jones` $\epsilon$ $\sigma$ $r_{\text{cut}}$ [( $c_{\text{shift}}$\|`auto` ) [$r_{\text{off}}$ [$r_{\text{cap}}$ [ $r_{\text{min}}$]]]] | 56 |
| Required features: `LENNARD_JONES` | |
| `inter` *type1 type2* `lj-gen` $\epsilon$ $\sigma$ $r_{\text{cut}}$ $c_{\text{shift}}$ $r_{\text{off}}$ $e_1$ $e_2$ $b_1$ $b_2$ [( $r_{\text{cap}}$\|`auto` ) $\lambda$ $\delta$] | 56 |
| Required features: `LENNARD_JONES_GENERIC` | |
| `inter` *type1 type2* `lj-cos` $\epsilon$ $\sigma$ $r_{\text{cut}}$ $r_{\text{off}}$<br>`inter` *type1 type2* `lj-cos2` $\epsilon$ $\sigma$ $r_{\text{off}}$ $\omega$ | 57 |
| Required features: `LJCOS` `LJCOS2` | |
| `inter` *type1 type2* `smooth-step` $\sigma_1$ $n$ $\epsilon$ $k_0$ $\sigma_2$ $r_{\text{cut}}$ | 58 |
| Required features: `SMOOTH_STEP` | |
| `inter` *type1 type2* `bmhtf-nacl` $A$ $B$ $C$ $D$ $\sigma$ $r_{\text{cut}}$ | 59 |
| Required features: `BMHTF_NACL` | |
| `inter` *type1 type2* `morse` $\epsilon$ $\alpha$ $r_{\text{min}}$ $r_{\text{cut}}$ | 59 |
| Required features: `MORSE` | |
| `inter` *type1 type2* `buckingham` $A$ $B$ $C$ $D$ $r_{\text{cut}}$ $r_{\text{discont}}$ $\epsilon_{\text{shift}}$ | 60 |
| Required features: `BUCKINGHAM` | |
| `inter` *type1 type2* `soft-sphere` $a$ $n$ $r_{\text{cut}}$ $r_{\text{offset}}$ | 60 |
| Required features: `SOFT_SPHERE` | |
| `inter` *type1 type2* `membrane` $a$ $n$ $d_{\text{cut}}$ $d_{\text{offset}}$ | 60 |
| Required features: `MEMBRANE_COLLISION` | |
| `inter` *type1 type2* `hat` $F_{\texttt{max}}$ $r_c$ | 61 |
| Required features: `HAT` | |
| `inter` *type1 type2* `hertzian` $\sigma$ $\epsilon$ | 61 |
| Required features: `HERTZIAN` | |
| `inter` *type1 type2* `gaussian` $\sigma$ $\epsilon$ $r_{\text{cut}}$ | 62 |
| Required features: `GAUSSIAN` | |
| `inter` *type1 type2* `lj-angle` $\epsilon$ $\sigma$ $r_{\text{cut}}$ $b1_a$ $b1_b$ $b2_a$ $b2_b$ [$r_{\text{cap}}$ $z_0$ $\delta z$ $\kappa$ $\epsilon'$] | 62 |
| Required features: `LJ_ANGLE` | |
| `inter` *type1 type2* `gay-berne` $\epsilon_0$ $\sigma_0$ $r_{\text{cutoff}}$ $k1$ $k2$ $\mu$ $\nu$ | 63 |
| Required features: `ROTATION` `GAY_BERNE` | |
| `inter` *type1 type2* `affinity` $\alpha_1$ $\alpha_2$ | 64 |
| Required features: `SHANCHEN` | |
| `inter` *bondid* `fene` $K$ $\Delta r_{\text{max}}$ [$r_0$] | 65 |
| `inter` *bondid* `harmonic` $K$ $R$ [$r_{\text{cut}}$] | 65 |
| `inter` *bondid* `harmonic_dumbbell` $k_1$ $k_2$ $r$ [$r_{\text{cut}}$] | 66 |
| Required features: `ROTATION` | |
| `inter` *bondid* `quartic` $K_0$ $K_1$ $R$ [$r_{\text{cut}}$] | 67 |

| | |
|---|---|
| `inter` *bondid* `bonded_coulomb` $\alpha$ | 67 |
| `inter` *bondid* `subt_lj` *reserved* $R$ | 68 |
| `inter` *bondid* `rigid_bond` *constrained_bond_distance positional_tolerance velocity_tolerance*<br>Required features: `BOND_CONSTRAINT` | 68 |
| `inter` *bondid* `tabulated bond` *filename*<br>`inter` *bondid* `tabulated angle` *filename*<br>`inter` *bondid* `tabulated dihedral` *filename* | 68 |
| `inter` *bondid* `virtual_bond` | 69 |
| `inter` *bondid* `oif_local_force` $L^0_{AB}$ $k_s$ $k_{slin}$ $\phi$ $k_b$ $A_1$ $A_2$ $k_{al}$<br>Required features: `OIF_LOCAL_FORCES` | 69 |
| `inter` *bondid* `oif_global_force` $S^0$ $k_{ag}$ $V^0$ $k_v$<br>Required features: `OIF_GLOBAL_FORCES` | 70 |
| `inter` *bondid* `oif_out_direction`<br>Required features: `MEMBRANE_COLLISION` | 70 |
| `inter` *bondid* `angle_harmonic` $K$ $[\phi_0]$<br>`inter` *bondid* `angle_cosine` $K$ $[\phi_0]$<br>`inter` *bondid* `angle_cossquare` $K$ $[\phi_0]$<br>Required features: `BOND_ANGLE` | 73 |
| `inter` *bondid* `dihedral` $n$ $K$ $p$ | 74 |
| `inter coulomb 0.0`<br>`inter coulomb`<br>`inter coulomb` *parameters* | 74 |
| `inter coulomb` $l_B$ `p3m [gpu]` $r_{\text{cut}}$ ( *mesh* \| $\{mesh_x\ mesh_y\ mesh_z\}$ ) *cao alpha*<br>Required features: `ELECTROSTATICS` | 76 |
| `inter coulomb` $l_B$ `p3m [gpu]` ( `tune` \| `tunev2` ) `accuracy` *accuracy* [`r_cut` $r_{\text{cut}}$] [`mesh` *mesh*] [`cao` *cao*] [`alpha` $\alpha$]<br>Required features: `ELECTROSTATICS` | 77 |
| `inter coulomb` [`epsilon` ( `metallic` \| *epsilon* )] [`n_interpol` *points*] [`mesh_off` *xoff yoff zoff*] | 78 |
| `inter coulomb` $l_B$ `ewaldgpu` $r_{\text{cut}}$ ( $K_{\text{cut}}$ \| $\{K_{\text{cut,x}}\ K_{\text{cut,y}}\ K_{\text{cut,x}}\}$ ) *alpha*<br>Required features: `ELECTROSTATICS` | 79 |
| `inter coulomb` $l_B$ `ewaldgpu tune accuracy` *accuracy* `precision` *precision* `K_max` $K_{\max}$<br>Required features: `ELECTROSTATICS` | 80 |
| `inter coulomb` $l_B$ `ewaldgpu tunealpha` $r_{\text{cut}}$ ( $K_{\text{cut}}$ \| $\{K_{\text{cut,x}}\ K_{\text{cut,y}}\ K_{\text{cut,x}}\}$ ) *precision*<br>Required features: `ELECTROSTATICS` | 80 |

| | |
|---|---|
| inter coulomb $l_B$ dh $\kappa$ $r_{\text{cut}}$ [1]<br>inter coulomb $l_B$ dh $\kappa$ $r_{\text{cut}}$ $\varepsilon_{\text{int}}$ $\varepsilon_{ext}$ $r_0$ $r_1$ $\alpha$ [1,2]<br>Required features: [1] ELECTROSTATICS [2] COULOMB_DEBYE_HUECKEL | 81 |
| inter coulomb $l_B$ mmm2d *maximal_pairwise_error* [*fixed_far_cutoff*] [dielectric $\epsilon_t$ $\epsilon_m$ $\epsilon_b$] [dielectric-contrasts $\Delta_t$ $\Delta_b$] [capacitor $U$]<br>Required features: ELECTROSTATICS | 81 |
| efield_caps ( total \| induced \| applied )<br>Required features: ELECTROSTATICS | 82 |
| inter coulomb $l_B$ mmm1d *switch_radius* *maximal_pairwise_error*<br>inter coulomb $l_B$ mmm1d tune *maximal_pairwise_error*<br>Required features: ELECTROSTATICS PARTIAL_PERIODIC | 83 |
| inter coulomb $l_B$ mmm1dgpu *switch_radius* [*bessel_cutoff*] *maximal_pairwise_error*<br>inter coulomb $l_B$ mmm1dgpu tune *maximal_pairwise_error*<br>Required features: CUDA ELECTROSTATICS PARTIAL_PERIODIC MMM1D_GPU | 84 |
| inter coulomb $l_B$ memd $f$_*mass* *mesh* [epsilon $\varepsilon_\infty$]<br>Required features: ELECTROSTATICS | 84 |
| inter coulomb $l_B$ memd localeps node *node_x* *node_y* *node_z* dir $X/Y/Z$ eps $\varepsilon$<br>Required features: ELECTROSTATICS | 85 |
| inter coulomb $l_B$ memd adaptive *scaling* parameters $f$_*mass* *mesh*<br>Required features: ELECTROSTATICS | 85 |
| scafacos_methods<br>Required features: ELECTROSTATICS SCAFACOS | 87 |
| inter coulomb $l_b$ scafacos *method* [*parameters*] [tolerance_field *prec*]<br>Required features: ELECTROSTATICS SCAFACOS | 88 |
| inter coulomb elc *maximal_pairwise_error* *gap_size* [*far_cutoff*] [noneutralization] [dielectric $\epsilon_t$ $\epsilon_m$ $\epsilon_b$] [dielectric-contrasts $\Delta_t$ $\Delta_b$] [capacitor $U$]<br>Required features: ELECTROSTATICS | 89 |
| iccp3m *n_induced_charges* convergence *convergence_criterion* areas *areas* normals *normals* sigmas *sigmas* epsilons *epsilons* [eps_out *eps_out* ] [relax *relaxation_parameter* ] [max_iterations *max_iterations* ] [ext_field *ext_field*]<br>Required features: ELECTROSTATICS | 89 |

| | |
|---|---|
| `dielectric sphere center` *cx cy cz* `radius` *r* `res` *res*<br>`dielectric wall normal` *nx ny nz* `dist` *d* `res` *res*<br>`dielectric cylinder center` *cx cy cz* `axis` *ax ay az* `radius` *r* `direction` *d*<br>`dielectric pore center` *cx cy cz* `axis` *ax ay az* `radius` *r* `length` *l* `smoothing_radius` *rs* `res` *res*<br>`dielectric slitpore pore_mouth` *z* `channel_width` *c* `pore_width` *w* `pore_length` *l* `upper_smoothing_radius` *us* `lower_smoothing_radius` *ls* | 90 |
| `inter magnetic 0.0`<br>`inter magnetic`<br>`inter magnetic` *parameters* | 91 |
| `inter magnetic` $l_B$ `p3m` $r_{\text{cut}}$ *mesh cao alpha*<br>Required features: `DIPOLES` | 92 |
| `inter magnetic` $l_B$ `p3m` ( `tune` \| `tunev2` ) `accuracy` *accuracy* [`r_cut` $r_{\text{cut}}$] [`mesh` *mesh*] [`cao` *cao*] [`alpha` $\alpha$]<br>Required features: `DIPOLES` | 92 |
| `inter magnetic mdlc` *accuracy gap_size* [*far_cutoff*]<br>Required features: `DIPOLES` | 93 |
| `inter magnetic` $l_B$ `dawaanr`<br>Required features: `DIPOLES` | 93 |
| `inter magnetic` $l_B$ `mdds n_cut` *value_n_cut*<br>Required features: `DIPOLES MAGNETIC_DIPOLAR_DIRECT_SUM` | 94 |
| `inter magnetic` $l_B$ `dds-gpu`<br>Required features: `DIPOLES CUDA` | 94 |
| `scafacos_methods`<br>Required features: `DIPOLES SCAFACOS_DIPOLES SCAFACOS` | 95 |
| `inter magnetic` $l_b$ `scafacos` *method* [*parameters*] [`tolerance_field` *prec*]<br>Required features: `DIPOLES SCAFACOS_DIPOLES SCAFACOS` | 95 |
| `inter` *type1 type2* `tunable_slip` $T$ $\gamma_L$ $r_{\text{cut}}$ $\delta t$ $v_x$ $v_y$ $v_z$<br>Required features: `TUNABLE_SLIP` | 95 |
| `inter` *type1 type2* `inter_dpd` *gamma r_cut wf tgamma tr_cut twf*<br>Required features: `INTER_DPD` | 95 |
| `inter` *typeid1 typeid1* `comfixed` *flag*<br>Required features: `COMFIXED` | 96 |
| `inter` *typeid1 typeid2* `comforce` *flag dir force fratio*<br>Required features: `COMFORCE` | 97 |
| `inter forcecap` ( $F_{\text{max}}$ \| `individual` ) | 97 |
| `setmd` *variable*<br>`setmd` *variable* [*value*]`+` | 98 |

`thermostat` 98
`thermostat off`
`thermostat` *parameters*

`thermostat langevin` *temperature gamma_trans* [*gamma_rotate*] [`on/off`] 100
[`on/off`]

`thermostat ghmc` *temperature n_md phi* [`-no_flip | -flip | -random_flip`] 102
[`-no_scale | -scale`]

`thermostat dpd` *temperature gamma r_cut* [ `WF` *wf tgamma tr_cut* `TWF` *twf*] 103
Required features: `DPD or TRANS_DPD`

`thermostat inter_dpd` *temperature* 105
Required features: `INTER_DPD`

`setmd dpd_ignore_fixed_particles 0` 105
Required features: `INTER_DPD`

`thermostat npt_isotropic` *temperature gamma0 gammaV* 107
Required features: `NPT`

`thermostat cpu` *temperature* 107

`thermostat sd` *temperature* 108
Required features: `CUDA and SD`

`nemd exchange` *n_slabs n_exchange* 108
`nemd shearrate` *n_slabs shearrate*
`nemd off`
`nemd`
`nemd profile`
`nemd viscosity`
Required features: `NEMD`

`cellsystem domain_decomposition` [`-no_verlet_list`] 108

`cellsystem nsquare` 109

`cellsystem layered` *n_layers* 110

`cuda list` 110
`cuda setdevice` *id*
`cuda getdevice`

`on_collision` 111
`on_collision off`
`on_collision` [`exception`] `bind_centers` *d bond1*
`on_collision` [`exception`] `bind_at_point_of_collision` *d bond1 bond2 type*
`on_collision` [`exception`] `glue_to_surface` *d bond1 bond2 type type2 type3*
*type4 d2*
`on_collision` [`exception`] `bind_three_particles` *d bond1 bond2 res*

`reaction reactant_type` *rt* `catalyzer_type` *ct* `product_type` *pt* `range` *r* 111
`ct_rate` *k_ct* [`eq_rate` *k_eq*] [`react_once` *on/off*] [`swap` *on/off*]
`reaction off`
`reaction print`

Required features: `CATALYTIC_REACTIONS`[a]

[a] The current implementation also requires the use of verlet lists and domain decomposition.

---

`kill_particle_motion` [`rotation`] [1] 112

Required features: [1] `ROTATION`

---

`kill_particle_forces` [`torques`] [1] 115

Required features: [1] `ROTATION`

---

`system_CMS` 116

---

`system_CMS_velocity` 116

---

`galilei_transform` 117

---

`integrate` *steps* [`recalc_forces`] [`reuse_forces`] 117
`integrate set` [`nvt`]
`integrate set npt_isotropic` $p_{ext}$ *piston* [*x y z*] [`-cubic_box`]

---

`time_integration` 117
`time_integration` *steps*

---

`minimize_energy` $f_{\max}$ *steps gamma maxdisplacement* 118

---

`tune_skin` *min max tol steps* 119

---

`change_volume` $V_{\text{new}}$ 119
`change_volume` $L_{\text{new}}$ ( `x` | `y` | `z` | `xyz` )

---

`rotate_system` *phi theta alpha* 120

---

`lees_edwards_offset` $offset_{\text{new}}$ 120

Required features: `LEES_EDWARDS`

---

`velocities` $v_{\max}$ [`start` *pid*] [`count` *N*] 121

---

`sort_particles` 121

---

`parallel_tempering::main -rounds` *N* `-swap` *swap* `-perform` *perform* 122
[`-init` *init*] [`-values` $\{T_i\}$] [`-connect` *master*] [`-port` *port*]
[`-load` $j_{\text{node}}$] [`-resrate` $N_{\text{reset}}$] [`-info` *info*]

---

`parallel_tempering::set_shareddata` *data* 123

| | |
|---|---|
| `metadynamics` | 123 |
| `metadynamics set off` | |
| `metadynamics set distance` *pid₁ pid₂* $d_{\min}$ $d_{\max}$ $b_{\text{height}}$ $b_{\text{width}}$ $f_{\text{bound}}$ $d_{\text{bins}}$ *numrelaxationsteps* | |
| `metadynamics set relative_z` *pid₁ pid₂* $z_{\min}$ $z_{\max}$ $b_{\text{height}}$ $b_{\text{width}}$ $f_{\text{bound}}$ $z_{\text{bins}}$ *numrelaxationsteps* | |
| `metadynamics print_stat current_coord` | |
| `metadynamics print_stat coord_values` | |
| `metadynamics print_stat profile` | |
| `metadynamics print_stat force` | |
| `metadynamics load_stat` *profile_list force_list* | |
| `Required features: METADYNAMICS` | |
| `integrate_sd` *steps* | 127 |
| `setmd smaller_time_step 0.001` | 127 |
| `part` *i* `smaller_timestep 1` | |
| `Required features: MULTI_TIMESTEP` | |
| `analyze mindist` [*type_list_a type_list_b*] | 129 |
| `analyze distto` *pid* | |
| `analyze distto` *x y z* | |
| `analyze nbhood` *pid r_catch* | 130 |
| `analyze nbhood` *x y z* $r_c atch$ | |
| `analyze distribution` *part_type_list_a part_type_list_b* [*rmin* [*rmax* [*rbins* [*log_flag* [*int_flag*]]]]] | 131 |
| `analyze radial_density_map` *xbins ybins xrange yrange* [*axisofrotation centerofrotation beadtypelist* [*thetabins*]] | 132 |
| `analyze cylindrical_average` *center direction length radius bins_axial bins_radial* [*types*] | 132 |
| `analyze modes2d` | 133 |
| `analyze get_lipid_orients` | 134 |
| `analyze lipid_orient_order` | |
| `analyze bilayer_set` | 135 |
| `analyze bilayer_density_profile` | |
| `analyze cell_gpb` *Manningparameter outercellradius innercellradius* [*accuracy* [*numberofinteractions*]] | 135 |
| `analyze get_folded_positions` [`-molecule`] [`shift` *x y z*] | 136 |
| `analyze Vkappa` [( `reset` \| `read` \| `set` $V_{\kappa,1}$ $V_{\kappa,2}$ *avk* ) ] | 136 |
| `analyze` ( `rdf` \| `<rdf>` ) *part_type_list_a part_type_list_b* [*rmin rmax rbins*] | 136 |
| `analyze structurefactor` *types order* | 136 |
| `analyze vanhove` *type rmin rmax rbins* [*tmax*] | 137 |
| `analyze centermass` *part_type* | 137 |

| | |
|---|---|
| `analyze momentofinertiamatrix` *typeid*<br>`analyze find_principal_axis` *typeid* | 138 |
| `analyze gyration_tensor` [*typeid*] | 139 |
| `analyze aggregation` *dist_criteria s_mol_id f_mol_id*<br>[*min_contact* [*charge_criteria*]] | 139 |
| `analyze necklace` *pearl_threshold back_dist space_dist first length* | 139 |
| `analyze holes` $typeid_{\text{probe}}$ *mesh_size*<br>Required features: `LENNARD_JONES` | 139 |
| `analyze fluid temp` [1 or 2 or 3]<br>Required features: [1]`LB` [2]`LB_GPU` [3]`ELECTROKINETICS` | 140 |
| `analyze momentum` [( `particles` \| `lbfluid` )] | 140 |
| `analyze energy`<br>`analyze energy` ( `total` \| `kinetic` \| `coulomb` \| `magnetic` )<br>`analyze energy bonded` *bondid*<br>`analyze energy nonbonded` *typeid1 typeid2* | 141 |
| `analyze pressure`<br>`analyze pressure total`<br>`analyze pressure` ( `totals` \| `ideal` \| `coulomb` \|<br>`tot_nonbonded_inter` \| `tot_nonbonded_intra` \| `vs_relative` )<br>`analyze pressure bonded` *bondid*<br>`analyze pressure nonbonded` *typeid1 typeid2*<br>`analyze pressure nonbonded_intra` [*typeid*]<br>`analyze pressure nonbonded_inter` [*typeid*] | 141 |
| `analyze stress_tensor`<br>`analyze stress_tensor total`<br>`analyze stress_tensor` ( `totals` \| `ideal` \| `coulomb` \|<br>`tot_nonbonded_inter` \| `tot_nonbonded_intra` )<br>`analyze stress_tensor bonded` $bond_type$<br>`analyze stress_tensor nonbonded` *typeid1 typeid2*<br>`analyze stress_tensor nonbonded_intra` [*typeid*]<br>`analyze stress_tensor nonbonded_inter` [*typeid*] | 142 |
| `analyze local_stress_tensor` *periodic_x periodic_y periodic_z range_start_x*<br>*range_start_y range_start_z range_x range_y range_z bins_x bins_y*<br>*bins_z* | 142 |
| `analyze set chains` [*chain_start n_chains chain_length*]<br>`analyze set topo_part_sync`<br>`analyze set` | 143 |
| `analyze` ( `re` \| `<re>` ) [*chain_start n_chains chain_length*] | 144 |
| `analyze` ( `rg` \| `<rg>` ) [*chain_start n_chains chain_length*] | 145 |
| `analyze` ( `rh` \| `<rh>` ) [*chain_start n_chains chain_length*] | 146 |
| `analyze` ( `internal_dist` \| `<internal_dist>` ) [*chain_start n_chains chain_length*] | 146 |

| | |
|---|---|
| `analyze ( bond_dist \| <bond_dist> ) [index` *index*`]` [*chain_start n_chains chain_length*] | 147 |
| `analyze ( bond_l \| <bond_l> )` [*chain_start n_chains chain_length*] | 147 |
| `analyze ( formfactor \| <formfactor> )` *qmin qmax qbins* [*chain_start n_chains chain_length*] | 148 |
| `analyze rdfchain` *rmin rmax rbins* [*chain_start n_chains chain_length*] | 148 |
| `analyze ( <g1>\| <g2>\| <g3> )` [*chain_start n_chains chain_length*]<br>`analyze g123 [-init]` [*chain_start n_chains chain_length*] | 148 |
| `analyze append`<br>`analyze remove` [*index*]<br>`analyze replace` *index*<br>`analyze push` [*size*]<br>`analyze configs` *config* | 149 |
| `analyze configs`<br>`analyze stored` | 149 |
| `uwerr` *data nrep col* [*s_tau*] [`plot`]<br>`uwerr` *data nrep f* [*s_tau* [*f_args*]] [`plot`] | 150 |
| `observable new` *name* [*parameters+*] | 151 |
| `observable` *id* `print` [`no_calculation`] [`formatted`] | 151 |
| `observable` *id* `write_checkpoint` *filename* [`binary`] | 154 |
| `observable` *id* `read_checkpoint` *filename* [`binary`] | 157 |
| `observable` *id* `delete` | 158 |
| `observable new needs_profile_specs` [*other_parameters*] [ `minx` *minx* ] [ `maxx` *maxx* ] [ `miny` *miny* ] [ `maxy` *maxy* ] [ `minz` *minz* ] [ `maxz` *maxz* ] [ `xbins` *xbins* ] [ `ybins` *ybins* ] [ `zbins` *zbins* ] | 158 |
| `observable new needs_radial_profile_specs` [*other_parameters*] [ *center* `<cx> <cy> <cx>` ] [ `maxr` *maxr* ] [ `minz` *minz* ] [ `maxz` *maxz* ] [ `rbins` *rbins* ] [ `phibins` *phibins* ] [ `zbins` *zbins* ] | 158 |
| `correlation new obs1` *id1* [`obs2` *id2*] `corr_operation` *operation* `dt` *dt* `tau_max` *tau_max* [`tau_lin` *tau_lin*] [`compress1` *name* [`compress2` *name*] ] | 159 |
| `correlation`<br>`correlation n_corr` | 159 |
| `correlation` *id* `autoupdate { start \| stop}`<br>`correlation` *id* `update`<br>`correlation` *id* `finalize` | 160 |
| `correlation` *id* `write_to_file` *filename*<br>`correlation` *id* `print`<br>`correlation` *id* `print [ average1 \| variance1 \| correlation_time ]`<br>`correlation` *id* `print [ average_errorbars ]` | 162 |

| | |
|---|---|
| `correlation` *id* `write_checkpoint_binary` *filename*<br>`correlation` *id* `write_checkpoint_ascii` *filename* | 163 |
| `correlation` *id* `read_checkpoint_binary` *filename*<br>`correlation` *id* `read_checkpoint_ascii` *filename* | 163 |
| `h5md_init` */where/to/store/data/file* [*list_of_p_ids*] | 166 |
| `h5md_write_positions` [`folded`] | 166 |
| `h5md_write_velocities` | 168 |
| `h5md_write_forces` | 168 |
| `h5md_observable1D_init` *name_of_observable* | 168 |
| `h5md_observable1D_write` *name_of_observable value* | 169 |
| `h5md_observable2D_init` *name_of_observable* | 169 |
| `h5md_observable2D_write` *name_of_observable value* | 169 |
| `h5md_close` | 169 |
| `blockfile` *channel* `write variable` {*varname1 varname2 ...*}<br>`blockfile` *channel* `write variable all` | 169 |
| `blockfile` *channel* `write tclvariable` { *varname1 varname2 ...*}<br>`blockfile` *channel* `write tclvariable all`<br>`blockfile` *channel* `write tclvariable reallyall` | 170 |
| `blockfile` *channel* `write particles` *what* ( *range* \| `all` )<br>`blockfile` *channel* `write bonds` *range*<br>`blockfile` *channel* `write interactions` | 173 |
| `blockfile` *channel* `write random`<br>`blockfile` *channel* `write seed` | 173 |
| `blockfile` *channel* `write configs` | 174 |
| `blockfile` *channel* `write start` *tag*<br>`blockfile` *channel* `write end`<br>`blockfile` *channel* `write` *tag* [*arg*]... | 174 |
| `blockfile` *channel* `read start`<br>`blockfile` *channel* `read toend`<br>`blockfile` *channel* `read auto`<br>`blockfile` *channel* `read` ( `particles` \| `interactions` \| `bonds` \| `variable` \| `seed` \| `random` \| `bitrandom` \| `configs` ) | 175 |
| `writemd` *channel* [`posx`\|`posy`\|`posz`\|`vx`\|`vy`\|`vz`\|`fx`\|`fy`\|`fz`]... | 175 |
| `readmd` *channel* | 175 |
| `mpiio` *filename* [`read`\|`write`] [`pos`\|`v`\|`bond`\|`type`]... | 177 |
| `writevsf` *channelId* [( `short` \| `verbose` )] [`radius` ( *radii* \| `auto` )] [`typedesc` *typedesc*] | 177 |
| `writevcf` *channelId* [( `short` \| `verbose` )] [( `folded` \| `absolute` )] [`pids` ( *pids* \| `all` )] [`userdata` *userdata*] | 177 |
| `vtfpid` *pid* | 179 |
| `writevtk` *filename* [( `all` \| *types* )] | 180 |

| | |
|---|---|
| `writepsf` *file* [`-molecule`] $N_P$ $MPC$ $N_CI$ $N_pS$ $N_nS$ | 180 |
| `writepdb` *file*<br>`writepdbfoldchains` *file chain_start n_chains chain_length box_l*<br>`writepdbfoldtopo` *file shift* | 181 |
| `readpdb pdb_file` *pdbfile* `type` *type* `first_id` *firstid*<br>[ `itp_file` *itpfile* `first_type` *fisttype*]<br>[`lj_with` *othertype epsilon sigma* [1]] [`lj_rel_cutoff` *cutoff* [1] ]<br>[`fit_to_box`]<br>Required features: [1] `LENNARD_JONES` | 182 |
| `imd connect` [*port*]<br>`imd positions` [( `-unfolded` \|`-fold_chains` )]<br>`imd listen` *seconds*<br>`imd disconnect` | 182 |
| `prepare_vmd_connection` *filename* [`start`] [`wait` *wait*] [`localhost`]<br>[`constraints`] ...<br>`prepare_vmd_connection` [*filename* [*wait* [*start* [*constraints*]]]] | 182 |
| `lbfluid` [`gpu`] [2] [`agrid` *agrid*] [1 or 2] [`dens` *density* ] [1 or 2 or 3]<br>[`visc` *viscosity*] [1 or 2 or 3] [`tau` *lb_timestep*] [1 or 2]<br>[`bulk_viscosity` *bulk_viscosity*] [1 or 2 or 3] [`ext_force` $f_x$ $f_y$ $f_z$] [1 or 2 or 3]<br>[`friction` *gamma* ] [1 or 2 or 3] [`couple` *2pt/3pt* ] [2]<br>[`gamma_odd` *gamma_odd*] [1 or 2 or 3] [`gamma_even` *gamma_even*] [1 or 2 or 3]<br>[`mobility`] *mobilities* [3] [`sc_coupling`] *coupling_constants* [3]<br>Required features: [1] `LB` [2] `LB_GPU` [3] `SHANCHEN` | 183 |
| `lbfluid print_interpolated_velocity` $x$ $y$ $z$ | 184 |
| `lbfluid save_ascii_checkpoint` *filename*<br>`lbfluid save_binary_checkpoint` *filename*<br>`lbfluid load_ascii_checkpoint` *filename*<br>`lbfluid load_binary_checkpoint` *filename* | 187 |
| `thermostat lb` [1 or 2 or 3] $T$<br>Required features: [1] `LB` [2] `LB_GPU` [3] `SHANCHEN` | 187 |
| `lbnode` $x$ $y$ $z$ ( `print` \| `set` ) *args*<br>Required features: [1] `LB` [2] `LB_GPU` [3] `SHANCHEN` | 188 |
| `lbfluid remove_momentum`<br>Required features: [1] `LB` [2] `LB_GPU` [3] `SHANCHEN` | 188 |
| `lbfluid print` [`vtk`] *property filename*<br>`lbfluid print vtk velocity` [`bb1_x bb1_y bb1_z bb2_x bb2_y bb2_z`]<br>*filename* | 194 |
| `lbboundary` *shape shape_args* [`velocity` *vx vy vz*]<br>`lbboundary force` [$n_{\text{boundary}}$]<br>Required features: `LB_BOUNDARIES` | 195 |

| | |
|---|---|
| `lbfluid cpu`<br>`lbfluid gpu`<br>Required features: [1] `LB` [2] `LB_GPU` | 195 |
| `setmd mu_E` $\mu E_x$ $\mu E_y$ $\mu E_z$<br>Required features: `LB` `LB_ELECTROHYDRODYNAMICS` | 196 |
| `countBonds` *particle$_l$ist* | 199 |
| `findPropPos` *particle$_p$roperty$_l$ist property* | 200 |
| `findBondPos` *particle$_p$roperty$_l$ist* | 200 |
| `timeStamp` *path prefix postfix suffix* | 200 |

# B. Features

This chapter describes the features that can be activated in ESPResSo. Even if possible, it is not recommended to activate all features, because this will negatively effect ESPResSo's performance.

Features can be activated in the configuration header `myconfig.hpp` (see section 3.5 on page 30). Too activate `FEATURE`, add the following line to the header file:

```
#define FEATURE
```

## B.1. General features

The list contains all features, but there are tons of docs missing!

- `PARTIAL_PERIODIC` By default, all coordinates in ESPResSo are periodic. With `PARTIAL_PERIODIC` turned on, the ESPResSo global variable `periodic` (see section 6.1 on page 100) controls the periodicity of the individual coordinates. Note that this slows the integrator down by around $10 - 30\%$.
- `ELECTROSTATICS` This switches on the various electrostatics algorithms, such as P3M. See section 5.7 on page 77 for details on these algorithms.
- `DIPOLES` This activates the dipole-moment property of particles; In addition, the various magnetostatics algorithms, such as P3M are switched on. See section 5.7 on page 77 for details on these algorithms.
- `ROTATION` Switch on rotational degrees of freedom for the particles, as well as the corresponding quaternion integrator. See section 4.1.1 on page 32 for details. Note, that when the feature is activated, every particle has three additional degrees of freedom, which for example means that the kinetic energy changes at constant temperature is twice as large.
- `ROTATION_PER_PARTICLE` Allows to set whether a particle has rotational degrees of freedom.
- `LANGEVIN_PER_PARTICLE` Allows to choose the Langevin temperature and friction coefficient per particle.
- `ROTATIONAL_INERTIA`
- `EXTERNAL_FORCES` Allows to define an arbitrary constant force for each particle individually. Also allows to fix individual coordinates of particles, *e.g.* keep them at a fixed position or within a plane.

- `CONSTRAINTS` Turns on various spatial constraints such as spherical compartments or walls. This constraints interact with the particles through regular short ranged potentials such as the Lennard–Jones potential. See section 4.3 on page 46 for possible constraint forms.
- `TUNABLE_SLIP` Switch on tunable slip conditions for planar wall boundary conditions. See section 5.9.1 on page 96 for details.
- `MASS` Allows particles to have individual masses. Note that some analysis procedures have not yet been adapted to take the masses into account correctly.
- `EXCLUSIONS` Allows to exclude specific short ranged interactions within molecules.
- `COMFORCE` Allows to pull apart groups of particles
- `COMFIXED` Allows to fix the center of mass of all particles of a certain type.
- `MOLFORCES` Docs missing
- `BOND_CONSTRAINT` Turns on the RATTLE integrator which allows for fixed lengths bonds between particles. How to use it?
- `VIRTUAL_SITES_COM` Virtual sites are particles, the position and velocity of which is not obtained by integrating equations of motion. Rather, they are placed using the position (and orientation) of other particles. The feature `VIRTUAL_SITES_COM` allows to place a virtual particle into the center of mass of a set of other particles. See section 4.4 for details.
- `VIRTUAL_SITES_RELATIVE` Virtual sites are particles, the position and velocity of which is not obtained by integrating equations of motion. Rather, they are placed using the position (and orientation) of other particles. The feature `VIRTUAL_SITES_RELATIVE` allows for rigid arrangements of particles. See section 4.4 for details.
- `VIRTUAL_SITES_NO_VELOCITY`
- `VIRTUAL_SITES_THERMOSTAT`
- `THERMOSTAT_IGNORE_NON_VIRTUAL`
- `BOND_VIRTUAL`
- `MODES`
- `ADRESS`
- `METADYNAMICS`
- `LANGEVIN_PER_PARTICLE` Allows to define the temperature and friction coefficient for individual particles. See 4.1.1 for details.

- `CATALYTIC_REACTIONS` Allows the user to define three particle types to be reactant, catalyzer, and product. Reactants get converted into products in the vicinity of a catalyst according to a used-defined reaction rate constant. It is also possible to set up a chemical equilibrium reaction between the reactants and products, with another rate constant. See section 6.7 for details.
- `OVERLAPPED`
- `COLLISION_DETECTION` Allows particles to be bound on collision. See section **??**
- `OLD_RW_VERSION` This switches back to the old, *wrong* random walk code of the polymer. Only use this if you rely on the old behaviour and *know what you are doing*.
- `H5MD` Allows to write data to H5MD formatted hdf5 files.

In addition, there are switches that enable additional features in the integrator or thermostat:

- `NEMD` Enables the non-equilbrium (shear) MD support (see section 6.3 on page 109).
- `NPT` Enables an on–the–fly NPT integration scheme (see section 6.2.4 on page 108).
- `DPD` Enables the dissipative particle dynamics thermostat (see section 6.2.3 on page 105).
- `TRANS_DPD` Enables the transversal dissipative particle dynamics thermostat (see section 6.2.3 on page 106).
- `INTER_DPD` Enables the dissipative particle dynamics thermostat implemented as an interaction, allowing to choose different parameters between different particle types (see section 6.2.3 on page 107).
- `INTER_RF` Documentation!
- `DPD_MASS_RED` Enables masses in DPD using reduced, dimensionless mass units.
- `DPD_MASS_LIN` Enables masses in DPD using absolute mass units.
- `LB` Enables the lattice-Boltzmann fluid code (see section 12 on page 194).
- `LB_GPU` Enables the lattice-Boltzmann fluid code support for GPU (see section 12 on page 194).
- `SHANCHEN` Enables the Shan Chen bicomponent fluid code on the GPU (see section 12 on page 194).
- `LB_ELECTROHYDRODYNAMICS` Enables the implicit calculation of electro-hydrodynamics for charged particles and salt ions in an electric field.

- `SD` enable Stokesian Dynamics.
- `SD_NOT_PERIODIC` disable periodic boundary conditions in Stokesian Dynamics.
- `SD_USE_FLOAT` use float instead of double in Stokesian Dynamics.
- `SD_FF_ONLY` disable nearfield of Stokesian Dynamics.
- `SD_DEBUG` enable debug Stokesian Dynamics.

## B.2. Interactions

The following switches turn on various short ranged interactions (see section 5.1 on page 55):

- `TABULATED` Enable support for user–defined interactions.
- `LENNARD_JONES` Enable the Lennard–Jones potential.
- `LENNARD_JONES_GENERIC` Enable the generic Lennard–Jones potential with configurable exponents and individual prefactors for the two terms.
- `LJCOS` Enable the Lennard–Jones potential with a cosine–tail.
- `LJCOS2` Same as LJCOS, but using a slightly different way of smoothing the connection to 0.
- `LJ_ANGLE` Enable the directional Lennard–Jones potential.
- `GAY_BERNE`
- `HERTZIAN`
- `MOL_CUT`
- `NO_INTRA_NB`
- `MORSE` Enable the Morse potential.
- `BUCKINGHAM` Enable the Buckingham potential.
- `SOFT_SPHERE` Enable the soft sphere potential.
- `SMOOTH_STEP` Enable the smooth step potential, a step potential with two length scales.
- `BMHTF_NACL` Enable the Born-Meyer-Huggins-Tosi-Fumi potential, which can be used to model salt melts.

Some of the short range interactions have additional features:

- `LJ_WARN_WHEN_CLOSE` This adds an additional check to the Lennard–Jones potentials that prints a warning if particles come too close so that the simulation becomes unphysical.
- `OLD_DIHEDRAL` Switch the interface of the dihedral potential to its old, less flexible form. Use this for older scripts that are not yet adapted to the new interface of the dihedral potential.

If you want to use bond-angle potentials (see section 5.5 on page 74), you need the followig features.

- `BOND_ANGLE`
- `BOND_ANGLEDIST`
- `BOND_ENDANGLEDIST`

BOND_ANGLEDIST and BOND_ENDANGLEDIST are completely undocumented.

## B.3. Debug messages

Finally, there are a number of flags for debugging. The most important one are

- `ADDITIONAL_CHECKS` Enables numerous additional checks which can detect inconsistencies especially in the cell systems. This checks are however too slow to be enabled in production runs.
- `MEM_DEBUG` Enables an internal memory allocation checking system. This produces output for each allocation and freeing of a memory chunk, and therefore allows to track down memory leaks. This works by internally replacing `malloc`, `realloc` and `free`.

The following flags control the debug output of various sections of Espresso. You will however understand the output very often only by looking directly at the code.

- `COMM_DEBUG` Output from the asynchronous communication code.
- `EVENT_DEBUG` Notifications for event calls, i. e. the `on_?` functions in `initialize.c`. Useful if some module does not correctly respond to changes of e. g. global variables.
- `INTEG_DEBUG` Integrator output.
- `CELL_DEBUG` Cellsystem output.
- `GHOST_DEBUG` Cellsystem output specific to the handling of ghost cells and the ghost cell communication.
- `GHOST_FORCE_DEBUG`

- `VERLET_DEBUG` Debugging of the Verlet list code of the domain decomposition cell system.
- `LATTICE_DEBUG` Universal lattice structure debugging.
- `HALO_DEBUG`
- `GRID_DEBUG`
- `PARTICLE_DEBUG` Output from the particle handling code.
- `P3M_DEBUG`
- `ESR_DEBUG` debugging of $P^3M$s real space part.
- `ESK_DEBUG` debugging of $P^3M$s $k$–space part.
- `EWALD_DEBUG`
- `FFT_DEBUG` Output from the unified FFT code.
- `MAGGS_DEBUG`
- `RANDOM_DEBUG`
- `FORCE_DEBUG` Output from the force calculation loops.
- `PTENSOR_DEBUG` Output from the pressure tensor calculation loops.
- `THERMO_DEBUG` Output from the thermostats.
- `LJ_DEBUG` Output from the Lennard–Jones code.
- `MORSE_DEBUG` Output from the Morse code.
- `FENE_DEBUG`
- `ONEPART_DEBUG` Define to a number of a particle to obtain output on the forces calculated for this particle.
- `STAT_DEBUG`
- `POLY_DEBUG`
- `MOLFORCES_DEBUG`
- `LB_DEBUG` Output from the lattice–Boltzmann code.
- `VIRTUAL_SITES_DEBUG`
- `ASYNC_BARRIER` Introduce a barrier after each asynchronous command completion. Helps in detection of mismatching communication.

- `FORCE_CORE` Causes ESPResSo to try to provoke a core dump when exiting unexpectedly.
- `MPI_CORE` Causes ESPResSo to try this even with MPI errors.
- `SD_DEBUG` Causes ESPResSo to check more things in the Stokesian Dynamics code. If *warnings* is larger 1, SD prints more informations.

# C. Sample scripts

In the directory ESPResSo/samples you find several scripts that can serve as samples how to use ESPResSo.

**lj_liquid.tcl** Simple Lennard-Jones particle liquid. Shows the basic features of ESPResSo: How to set up system parameters, particles and interactions. How to warm up and integrate. How to write parameters, configurations and observables to files. How to handle the connection to VMD.

**pe_solution.tcl** Polyelectrolyte solution under poor solvent condition. Test case for comparison with data produced by polysim9 from M.Deserno. Note that the equilibration of this system takes roughly $15000\tau$.

**pe_analyze.tcl** Example for doing the analysis after the actual simulation run (offline analysis). Calculates the integrated ion distribution $P(r)$ for several different time slaps, compares them and presents the final result using gnuplot to generate some ps-files.

**harmonic_oscillator.tcl** A chain of harmonic oscillators. This is a $T = 0$ simulation to test the energy conservation.

**espresso_logo.tcl** The ESPResSo-logo, the exploding espresso cup, has been created with this script. It is a regular simulation of a polyelectrolyte solution. It makes use of some nice features of the part command (see section 4.1 on page 32, namely the capability to fix a particle in space and to apply an external force.

# D. Maxwell Equations Molecular Dynamics (MEMD)

In this chapter, we want to give a more thorough introduction to the MEMD (or "Maggs") algorithm for the calculation of Coulomb interactions that is implemented in ESPResSo. For an even more detailed description, we refer to the publications [41, 46]. The method is intimately related to the Car–Parrinello approach, while being equivalent to solving Maxwell's equations with freely adjustable speed of light.

## D.1. Equations of motion

Denoting the particle masses with $m_i$, their charges with $q_i$, their coordinates and momentum with $\vec{r}_i$ and $\vec{p}_i$ respectively, the interparticle potential (of *non*-electromagnetic type) with $U$, for the coupled system of charges and fields we write the following equations of motion

$$\dot{\vec{r}}_i = \frac{1}{m_i}\vec{p}_i \tag{D.1}$$

$$\dot{\vec{p}}_i = -\frac{\partial U}{\partial \vec{r}_i} + q_i\vec{E}(\vec{r}_i) - \frac{\zeta}{m_i}\vec{p}_i + \vec{f}_i \tag{D.2}$$

$$\dot{\vec{A}} = -\vec{E} \tag{D.3}$$

$$\dot{\vec{E}} = c^2\vec{\nabla}\times\left(\vec{\nabla}\times\vec{A}\right) - \frac{1}{\epsilon_0}\vec{j}, \tag{D.4}$$

where $\epsilon_0$ is the vacuum dielectric constant, $c$ the speed of light, $\vec{A}$ the vector-potential, $\vec{E}$ the electric field, $\vec{j}$ the current density; $\zeta$ is the particle friction constant, and $\vec{f}_i$ is a random force satisfying the standard fluctuation-dissipation theorem:

$$\left\langle f_i^\alpha(t) f_j^\beta(t')\right\rangle = 2\zeta k_B T\delta_{ij}\delta_{\alpha\beta}\delta(t-t'), \tag{D.5}$$

where $\alpha$ and $\beta$ denote Cartesian indices.

If we introduce the vector $\vec{B} = \nabla\times A$ the system of equations can be rewritten in a form similar to the usual Maxwell equations. Currently in ESPResSo the version with $\vec{B}$ and $\vec{E}$ is implemented.

## D.2. Discretization

For implementation on the computer, the equations need to be discretized with respect to both space and time.We consider a domain of physical space as being an affine space and divide it into subdomains of contiguous cells of cubic shape. The charges live on the vertices of our lattice which has the spacing $a$. The electric fields $E(l)$ and vector potentials $A(l)$ live on the edges or links and are aligned with them. We need also the operator $\nabla \times$ . It gives the vector $\vec{B}$, which lives on the faces of the cube or on the plaquettes, Fig. D.1.

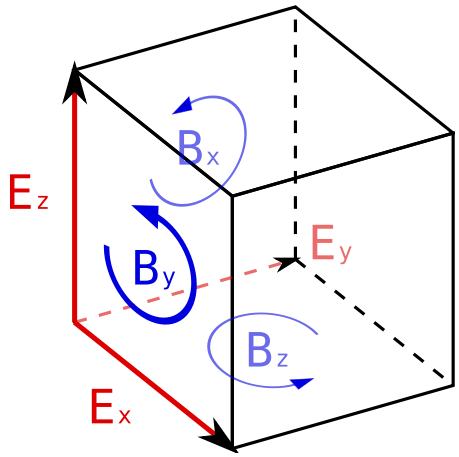



Figure D.1.: Spatial elements of a cell complex

In the implementation of the algorithm we assume that particles with masses $m_i$ and charges $q_i$ live in the continuum (off–lattice approach). The charges are interpolated on the lattice with grid spacing $a$ using a linear interpolation scheme.

## D.3. Initialization of the algorithm

The algorithm as it is implemented only calculates stepwise time updates of the exact field solution. Therefore in order to start the simulation for the given random distribution of charges we have to calculate the initial electrostatic field, i. e. the exact solution of the electrostatic problem. We find a particular solution of Gauss' law as the result of the following recursive procedure (see Fig. D.2):

1. The charge in the plane $z = z_{\text{plane}}$ is

$$q_{\text{plane}} = \frac{1}{N_z} \sum_i q(\vec{r}_i)\delta(z_i - z_{\text{plane}}), \tag{D.6}$$

$N_z$ is the number of charges in plane $z = z_{\text{plane}}$. Update the $z$-field according to the formula

$$E_z^2 = E_z^1 + \frac{q_{\text{plane}}}{\epsilon_0 a^2}; \tag{D.7}$$

2. Subtract the charge $q_{\text{plane}}$ from the each charge on sites of $z_{\text{plane}}$. The charge of the wire $y = y_{\text{wire}}, z = z_{\text{plane}}$ is

$$q_{\text{wire}} = \frac{1}{N_y} \sum_i q(\vec{r}_i)\delta(z_i - z_{\text{plane}})\delta(y_i - y_{\text{wire}}), \tag{D.8}$$

$N_y$ now meaning the number of charges in the wire. Update $y$-field

$$E_y^2 = E_y^1 + \frac{q_{\text{wire}}}{\epsilon_0 a^2}; \tag{D.9}$$

3. Subtract the charge $q_{\text{wire}}$ from the each charge on the sites of $(y_{\text{wire}}, z_{\text{plane}})$. Update $x$ field

$$E_x^2 = E_x^1 + \frac{q_{\text{vertex}}}{\epsilon_0 a^2} \tag{D.10}$$

This scheme is repeated until the fields are completely relaxed (i. e. the energy is minimized). During repetition, the spatial dimensions are permutated to avoid a drift in one direction.

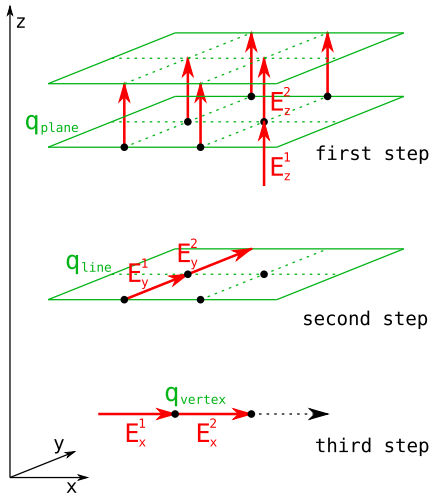



Figure D.2.: Recursive solution of Gauss' law

## D.4. Time integrator

For the time discretization we have adopted the elegant solution which was found by Rottler and Maggs [41] and allows to conserve *both* time–reversibility and phase–space volume conservation:

1. Update the particle momenta by half a time step.

2. Update the $\vec{B}$ field by half a time step.
3. Update the particle positions in $x$ direction by half a time step.
4. Update the electric field in $x$ direction by half a time step.
5. Update the particle positions in $y$ direction by half a time step.
6. Update the electric field in $y$ direction by half a time step.
7. Update the particle positions in $z$ direction by half a time step.
8. Update the electric field in $z$ direction by a full time step.
9. Update the particle positions in $z$ direction by half a time step.
10. Update the electric field in $y$ direction by half a time step.
11. Update the particle positions in $y$ direction by half a time step.
12. Update the electric field in $x$ direction by half a time step.
13. Update the particle positions in $x$ direction by half a time step.
14. Update the $\vec{B}$ field by half a time step.
15. Update the particle momenta by half a time step.

## D.5. Self–energy

The interpolation of the charges onto the lattice gives rise to the artificial force exerted on the particle by its own field. In order to cure this remedy, the direct subtraction of the self–energy is introduced.

For the interpolated charge cloud the self–energy can be directly calculated. For the simple cubic lattice in three dimensions the linear interpolation will give 8 charges which are placed at the corners of the cube with edge length $a$ (see Fig. D.3).

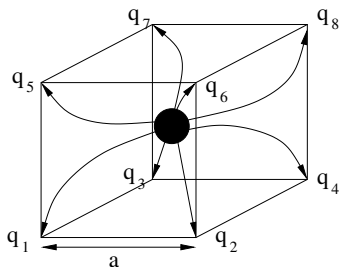



Figure D.3.: Linear interpolation scheme

Therefore in our case the self-energy is a symmetric bilinear form defined by the matrix $\{\alpha_{ij}\}$, the elements of which do not depend on the position of the charge. In our algorithm the values of the coefficients are

$$\alpha_{ij} = \frac{1}{4a\epsilon_0 L^3} \sum_{\vec{k}} \frac{\cos \vec{k}(\vec{R}_i - \vec{R}_j)}{\sum_{i=1}^{3}(1 - \cos \vec{k}\vec{a}_i)} \tag{D.11}$$

where $L$ is the number of lattice points per dimension, $\vec{R}_i$ coordinates of the interpolated charges and $\vec{k}$ the wave vector. Those values are calculated during the initialization step and are used in the calculation of the self-force. The value of the self-force which has to be subtracted from the overall forces is given by the following ansatz

$$\vec{F}_{self} = -\frac{\partial \mathcal{U}_{self}}{\partial \vec{r}} = -\sum_i \sum_j \alpha_{ij} \left[ q_i \frac{\partial q_j}{\partial \vec{r}} + q_j \frac{\partial q_i}{\partial \vec{r}} \right]. \tag{D.12}$$

## D.6. For which systems to use the algorithm

Although it is not very well known by now, this algorithm is a promising alternative to the often used Ewald-based methods. The main advantages and disadvantages shall be named here. However, it is still best to understand the concept of the algorithm and figure out for yourself, if it may be an option.

- The fields are not calculated for an arbitrary charge distribution, but updated from the last solution. Therefore, particles should not move too much between timesteps (less than a lattice cube).

- No procedure for error tuning yet. You have to adjust the parameters and determine the error yourself.

- Only 3D periodic systems are possible for now.

- With the given interpolation scheme, the short-range part of the potential is highy underestimated when two particles are in the same lattice cube!

- The initialization routine scales with $\mathcal{O}(N^3)$ and takes a long time for larger (and also inhomogenous) systems.

+ The algorithm is a local update scheme and spatially varying properties can be applied (in the future).

+ Because of the locality, the algorithm itself scales $\mathcal{O}(N)$ and has a big advantage in speed for larger systems.

+ Because of the locality, it is highly parallelized.

+ It is fast.

The last item is of course dependent on the system properties. But if the charges are evenly distributed and the system is not too sparse, this algorithm outperforms P3M easily. Especially for systems with more than 1000 charges.

Of course, if the system is not dense enough, one will have to set the lattice spacing in a way to avoid several particles in one cell and the mesh will be very fine for not so many charges. Also, if you have lots of charges but your simulation should only run for a short time, the initialization scheme takes too long in comparison.

But, if you have dense systems with more than 1000 charges or simulations that run for many timesteps, this method is definitely an option.

# E. The MMM family of algorithms

## E.1. Introduction

Cleanup: References, mathematics

In the MMM family of algorithms for the electrostatic interaction, a convergence factor approach to tackle the conditionally convergent Coulomb sum is used (even the authors of the original MMM method have no idea what this acronym stands for). Instead of defining the summation order, one multiplies each summand by a continuous factor $c(\beta, r_{ij}, n_{klm})$ such that the sum is absolutely convergent for $\beta > 0$, but $c(0, ., .) = 1$. The energy is then defined as the limit $\beta \to 0$ of the sum, i. e. $\beta$ is an artificial convergence parameter. For a convergence factor of $e^{-\beta n_{klm}^2}$ the limit is the same as the spherical limit, and one can derive the classical Ewald method quite conveniently through this approach [56]. To derive the formulas for MMM, one has to use a different convergence factor, namely $e^{-\beta|r_{ij}+n_{klm}|}$, which defines the alternative energy

$$\tilde{E} = \frac{1}{2}\lim_{\beta\to 0}\sum_{k,l,m}{\sum_{i,j=1}^{N}}' \frac{q_i q_j e^{-\beta|p_{ij}+n_{klm}|}}{|p_{ij}+n_{klm}|} =: \frac{1}{2}\lim_{\beta\to 0}\sum_{i,j=1}^{N} q_i q_j \phi_\beta(x_{ij}, y_{ij}, z_{ij}).$$

$\phi_\beta$ is given by $\phi_\beta(x, y, z) = \tilde{\phi}_\beta(x, y, z) + \frac{e^{-\beta r}}{r}$ for $(x, y, z) \neq 0$ and $\phi_\beta(0, 0, 0) = \tilde{\phi}_\beta(0, 0, 0)$, where

$$\tilde{\phi}_\beta(x, y, z) = \sum_{(k,l,m)\neq 0} \frac{e^{-\beta r_{klm}}}{r_{klm}}.$$

The limit $\tilde{E}$ exists, but differs for three dimensionally periodic systems by some multiple of the square of the dipole moment from the spherical limit as obtained by the Ewald summation[56]. From the physical point of view the Coulomb interaction is replaced by a screened Coulomb interaction with screening length $1/\beta$. $\tilde{E}$ is then the energy in the limit of infinite screening length. But because of the conditional convergence of the electrostatic sum, this is not necessarily the same as the energy of an unscreened system. Since the difference to the Ewald methods only depends on the dipole moment of the system, the correction can be calculated easily in linear time and can be ignored with respect to accuracy as well as to computation time.

For one or two dimensionally systems, however, $\tilde{E} = E$, *i.e.* the convergence factor approach equals the spherical summation limit of the Ewald sum, and MMM1D and MMM2D do not require a dipole correction.

Starting from this convergence factor approach, Strebel constructed a method of computational order $O(N \log N)$, which is called MMM [59]. The favourable scaling is obtained, very much like in the Ewald case, by technical tricks in the calculation of the far

formula. The far formula has a product decomposition and can be evaluated hierarchically similarly to the fast multipole methods.

For particles sufficiently separated in the z-axis one can Fourier transform the potential along both x and y. We obtain the far formula as

$$\phi(x,y,z) = u_x u_y \sum_{p,q\neq 0} \frac{e^{2\pi f_{pq} z} + e^{2\pi f_{pq}(\lambda_z - z)}}{f_{pq}\left(e^{2\pi f_{pq}\lambda_z} - 1\right)} e^{2\pi i u_y q y} e^{2\pi i u_x p x} + 2\pi u_x u_y \left(u_z z^2 - z + \frac{\lambda_z}{6}\right).$$

where $\lambda_{x,y,z}$ are the box dimensions, $f_{pq} = \sqrt{(u_x p)^2 + (u_y q)^2}$, $f_p = u_x p$, $f_q = u_x q$, $\omega_p = 2\pi u_x p$ and $\omega_q = 2\pi u_y q$. The advantage of this formula is that it allows for a product decomposition into components of the particles. For example

$$e^{2\pi f_{pq} z} = e^{2\pi f_{pq}(z_i - z_j)} = e^{2\pi f_{pq} z_i} e^{-2\pi f_{pq} z_j}$$

etc. Therefore one just has to calculate the sum over all these exponentials on the left side and on the right side and multiply them together, which can be done in $O(N)$ computation time. As can be seen easily, the convergence of the series is excellent as long as z is sufficiently large. By symmetry one can choose the coordinate with the largest distance as z to optimise the convergence. Similar to the Lekner sum, we need a different formula if all coordinates are small, i. e. for particles close to each other. For sufficiently small $u_y\rho$ and $u_x x$ we obtain the near formula as

$$\begin{aligned}
\tilde{\phi}(x,y,z) = \; & 2u_x u_y \sum_{p,q>0} \frac{\cosh(2\pi f_{pq} z)}{f_{pq}\left(e^{2\pi f_{pq}\lambda_z} - 1\right)} e^{2\pi i u_y q y} e^{2\pi i u_x p x} + \\
& 4u_x \sum_{l,p>0} \left(K_0(2\pi u_x p \rho_l) + K_N(2\pi u_x p \rho_{-l})\right) cos(2\pi u_x p x) - \\
& 2u_x \sum_{n\geq 1} \frac{b_{2n}}{2n(2n)!} \Re\left((2\pi u_y (z + iy))^{2n}\right) + \\
& u_x \sum_{n\geq 0} \binom{-\frac{1}{2}}{n} \frac{\left(\psi^{(2n)}(1+u_x x) + \psi^{(2n)}(1-u_x x)\right)}{(2n)!} \rho^{2n} - \\
& 2\log(4\pi).
\end{aligned}$$

Note that this time we calculate $\tilde{\phi}$ instead of $\phi$, i. e. we omit the contribution of the primary simulation box. This is very convenient as it includes the case of self energy and makes $\tilde{\phi}$ a smooth function. To obtain $\phi$ one has to add the $1/r$ contribution of the primary box. The self energy is given by

$$\tilde{\phi}(0,0,0) = 2u_x u_y \sum_{p,q>0} \frac{1}{f_{pq}\left(e^{2\pi f_{pq}\lambda_z} - 1\right)} + 8u_x \sum_{l,p>0} K_N(2\pi u_x \lambda_y p l) + 2u_x \psi^{(0)}(1) - 2\log(4\pi).$$

Both the near and far formula are derived using the same convergence factor approach, and consequently the same singularity in $\beta$ is obtained. This is important since otherwise the charge neutrality argument does not hold.

To obtain the $O(N \log N)$ scaling, some algorithm tricks are needed, which are not used in MMM1D, MMM2D or ELC and are therefore not discussed here. For details, see Strebel [59]. MMM is not implemented in ESPResSo.

## E.2. MMM2D

In the case of periodicity only in the x and y directions, the far formula looks like

$$\begin{aligned}\phi(x,y,z) = \quad & 4u_xu_y\sum_{p,q>0}\frac{e^{-2\pi f_{pq}|z|}}{f_{pq}}\cos(\omega_p x)\cos(\omega_q y)+\\ & 2u_xu_y\left(\sum_{q>0}\frac{e^{-2\pi f_q|z|}}{f_q}\cos(\omega_q y)+\sum_{p>0}\frac{e^{-2\pi f_p|z|}}{f_p}\cos(\omega_p x)\right)-\\ & 2\pi u_xu_y|z|\end{aligned}$$

,

and the near formula is

$$\begin{aligned}\tilde{\phi}(x,y,z) = \quad & 4u_x\sum_{l,p>0}\left(K_0(\omega_p\rho_l)+K_0(\omega_p\rho_{-l})\right)\cos(\omega_p x)-\\ & 2u_x\sum_{n\geq 1}\frac{b_{2n}}{2n(2n)!}\Re\left((2\pi u_y(z+iy))^{2n}\right)+\sum_{k=1}^{N_\psi-1}\left(\frac{1}{r_k}+\frac{1}{r_{-k}}\right)-\\ & u_x\sum_{n\geq 0}\binom{-\frac{1}{2}}{n}\frac{\left(\psi^{(2n)}(N_\psi+u_xx)+\psi^{(2n)}(N_\psi-u_xx)\right)}{(2n)!}(u_x\rho)^{2n}-\\ & 2u_x\log\left(4\pi\frac{u_y}{u_x}\right).\end{aligned}$$

As said before, the energy obtained from these potentials is equal to the electrostatic energy obtained by the spherical summation limit. The deeper reason for this is that in some sense the electrostatic sum is absolutely convergent [5].

The near formula is used for particles with a small distance along the z axis, for all other particles the far formula is used. Below is shown, that the far formula can be evaluated much more efficiently, however, its convergence breaks down for small z distance. To efficiently implement MMM2D, the layered cell system is required, which splits up the system in equally sized gaps along the z axis. The interaction of all particles in a layer S with all particles in the layers S-1,S,S+1 is calculated using the near formula, for the particles in layers $1,\ldots,S-2$, and in layers $S+2,\ldots,N$, the far formula is used.

The implementation of the near formula is relatively straight forward and can be treated as any short ranged force is treated using the link cell algorithm, here in the layered variant. The special functions in the formula are somewhat demanding, but for the polygamma functions Taylor series can be achieved, which are implemented in mmm-common.h. The Bessel functions are calculated using a Chebychev series.

The treatment of the far formula is algorithmically more complicated. For a particle i in layer $S_i$, the formula can product decomposed, as in

$$\begin{aligned}&\sum_{j\in I_S,S<S_i-1}q_iq_j\frac{e^{-2\pi f_{pq}|z_i-z_j|}}{f_{pq}}\cos(\omega_p(x_i-x_j))\cos(\omega_q(y_i-y_j)) =\\ &q_i\frac{e^{-2\pi f_{pq}z_i}}{f_{pq}}\cos(\omega_p x_i)\cos(\omega_q y_i)\sum_{j\in I_S,S<S_i-1}q_je^{2\pi f_{pq}z_j}\cos(\omega_p x_j)\cos(\omega_q y_j)+\\ &q_i\frac{e^{-2\pi f_{pq}z_i}}{f_{pq}}\cos(\omega_p x_i)\sin(\omega_q y_i)\sum_{j\in I_S,S<S_i-1}q_je^{2\pi f_{pq}z_j}\cos(\omega_p x_j)\sin(\omega_q y_j)+\\ &q_i\frac{e^{-2\pi f_{pq}z_i}}{f_{pq}}\sin(\omega_p x_i)\cos(\omega_q y_i)\sum_{j\in I_S,S<S_i-1}q_je^{2\pi f_{pq}z_j}\sin(\omega_p x_j)\cos(\omega_q y_j)+\\ &q_i\frac{e^{-2\pi f_{pq}z_i}}{f_{pq}}\sin(\omega_p x_i)\sin(\omega_q y_i)\sum_{j\in I_S,S<S_i-1}q_je^{2\pi f_{pq}z_j}\sin(\omega_p x_j)\sin(\omega_q y_j).\end{aligned}$$

This representation has the advantage, that the contributions of the two particles are decoupled. For all particles j only the eight terms

$$\xi_j^{(\pm,s/c,s/c)} = q_j e^{\pm 2\pi f_{pq} z_j} \sin / \cos(\omega_p x_j) \sin / \cos(\omega_q y_j)$$

are needed. The upper index describes the sign of the exponential term and whether sine or cosine is used for $x_j$ and $y_j$ in the obvious way. These terms can be used for all expressions on the right hand side of the product decomposition. Moreover it is easy to see from the addition theorem for the sine function that these terms also can be used to calculate the force information up to simple prefactors that depend only on p and q.

Every processor starts with the calculation of the terms $\xi_j^{(\pm,s/c,s/c)}$ and adds them up in each layer, so that one obtains

$$\Xi_s^{(\pm,s/c,s/c)} = \sum_{j \in S_s} \xi_j^{(\pm,s/c,s/c)}.$$

Now we calculate

$$\Xi_s^{(l,s/c,s/c)} = \sum_{t<s-1} \Xi_t^{(+,s/c,s/c)}$$

and

$$\Xi_s^{(h,s/c,s/c)} = \sum_{t>s+1} \Xi_t^{(-,s/c,s/c)},$$

which are needed for the evaluation of the product decomposition. While the bottom processor can calculate $\Xi_s^{(l,s/c,s/c)}$ directly, the other processors are dependent on its results. Therefore the bottom processor starts with the calculation of its $\Xi_s^{(l,s/c,s/c)}$ and sends up $\Xi_s^{(l,s/c,s/c)}$ and $\Xi_s^{(+,s/c,s/c)}$ of its top layer s to the next processor dealing with the layers above. Simultaneously the top processor starts with the calculation of the $\Xi_s^{(h,s/c,s/c)}$ and sends them down. After the communicated has been completed, every processor can use the $\Xi_j^{(l/h,s/c,s/c)}$ and the $\xi_j^{(\pm,s/c,s/c)}$ to calculate the force rsp. energy contributions for its particles.

In pseudo code, the far formula algorithm looks like:

1. for each layer $s = 1, \ldots, S$
   a) $\Xi_s^{(\pm,s/c,s/c)} = 0$
   b) for each particle $j$ in layer $s$
      i. calculate $\xi_j^{(\pm,s/c,s/c)}$
      ii. $\Xi_s^{(\pm,s/c,s/c)} += \xi_j^{(\pm,s/c,s/c)}$
2. $\Xi_3^{(l,s/c,s/c)} = \Xi_1^{(+,s/c,s/c)}$
3. for each layer $s = 4, \ldots, S$

a) $\Xi_s^{(l,s/c,s/c)} = \Xi_{s-1}^{(l,s/c,s/c)} + \Xi_{s-2}^{(+,s/c,s/c)}$

4. $\Xi_{S-2}^{(l,s/c,s/c)} = \Xi_S^{(-,s/c,s/c)}$

5. for each layer $s = (S-3), ..., 1$
   a) $\Xi_s^{(l,s/c,s/c)} = \Xi_{s+1}^{(l,s/c,s/c)} + \Xi_{s+2}^{(-,s/c,s/c)}$

6. for each layer $s = 1, ..., S$
   a) for each particle $j$ in layer $s$
      i. calculate particle interaction from $\xi_j^{(+,s/c,s/c)}\Xi_s^{(l,s/c,s/c)}$ and $\xi_j^{(-,s/c,s/c)}\Xi_s^{(h,s/c,s/c)}$

For further details, see Arnold and Holm [5, 4], Arnold et al. [6].

### E.2.1. Dielectric contrast

A dielectric contrast at the lower and/or upper simulation box boundary can be included comparatively easy by using image charges. Apart from the images of the lowest and topmost layer, the image charges are far enough to be treated by the far formula, and can be included as starting points in the calculation of the $\Xi$ terms. The remaining particles from the lowest and topmost layer are treated by direct summation of the near formula.

This means, that in addition to the algorithm above, one has to only a few things: during the calculation of the particle and cell blocks $\xi$ and $\Xi$, one additionally calculates the contributions of the image charges and puts them either in a separate array or, for the boundary layers, into two extra $\xi$ cell blocks outside the simulation box. The entries in the separate array are then added up over all processors and stored in the $\Xi$-terms of the lowest/topmost layer. This are all modifications necessary for the far formula part. In addition to the far formula part, there is an additional loop over the particles at the boundary to directly calculate their interactions with their images. For details, refer to Tyagi et al. [63].

## E.3. MMM1D

In one dimensionally periodic systems with z being the periodic coordinate, the far formula looks like

$$\begin{aligned} \phi(\rho,z) &= 4u_z \textstyle\sum_{p\neq 0} K_0(\omega\rho)\cos(\omega z) - 2u_z \log(\frac{\rho}{2\lambda_z}) - 2u_z\gamma \\ F_\rho(\rho,z) &= 8\pi u_z^2 \textstyle\sum_{p\neq 0} pK_1(\omega\rho)\cos(\omega z) + \frac{2u_z}{\rho} \\ F_z(\rho,z) &= 8\pi u_z^2 \textstyle\sum_{p\neq 0} pK_0(\omega\rho)\sin(\omega z), \end{aligned}$$

the near formula is

$$
\begin{aligned}
\tilde{\phi}(\rho,z) &= -u_z \sum_{n\geq 0} \binom{-\frac{1}{2}}{n} \frac{\left(\psi^{(2n)}(N_\psi+u_z z)+\psi^{(2n)}(N_\psi-u_z z)\right)}{(2n)!}(u_z\rho)^{2n} - 2u_z\gamma + \\
&\quad \sum_{k=1}^{N_\psi-1}\left(\frac{1}{r_k}+\frac{1}{r_{-k}}\right) \\
\tilde{F}_\rho(\rho,z) &= -u_z^3 \sum_{n\geq 0} \binom{-\frac{1}{2}}{n} \frac{\left(\psi^{(2n)}(N_\psi+u_z z)+\psi^{(2n)}(N_\psi-u_z z)\right)}{(2n)!}(u_z\rho)^{2n-1} + \\
&\quad \sum_{k=1}^{N_\psi-1}\left(\frac{\rho}{r_k^3}+\frac{\rho}{r_{-k}^3}\right) \\
\tilde{F}_z(\rho,z) &= -u_z^2 \sum_{n\geq 0} \binom{-\frac{1}{2}}{n} \frac{\left(\psi^{(2n+1)}(N_\psi+u_z z)+\psi^{(2n+1)}(N_\psi-u_z z)\right)}{(2n)!}(u_z\rho)^{2n} + \\
&\quad \sum_{k=1}^{N_\psi-1}\left(\frac{z+k\lambda_z}{r_k^3}+\frac{z-k\lambda_z}{r_{-k}^3}\right),
\end{aligned}
$$

where $\rho$ denotes the xy-distance of the particles. As for the two dimensional periodic case, the obtained energy is equal to the one dimensional Ewald sum. Algorithmically, MMM1D is uninteresting, since neither the near nor far formula allow a product decomposition or similar tricks. MMM1D has to be implemented as a simple NxN loop. However, the formulas can be evaluated efficiently, so that MMM1D can still be used reasonably for up to 400 particles on a single processor [3].

## E.4. ELC

The ELC method differs from the other MMM algorithms in that it is not an algorithm for the calculation of the electrostatic interaction, but rather represents a correction term which allows to use any method for threedimensionally periodic systems with spherical summation order for twodimensional periodicity. The basic idea is to expand the two dimensional slab system of height h in the non-periodic z-coordinate to a system with periodicity in all three dimensions, with a period of $\lambda_z > h$, which leaves an empty gap of height $\delta = \lambda_z - h$ above the particles in the simulation box.

Since the electrostatic potential is only finite if the total system is charge neutral, the additional image layers (those layers above or below the original slab system) are charge neutral, too. Now let us consider the n-th image layer which has an offset of $n\lambda_z$ to the original layer. If $n\lambda_z$ is large enough, each particle of charge q_j at position $(x_j, y_j, z_j + n\lambda_z)$ and its replicas in the xy-plane can be viewed as constituting a homogeneous charged sheet of charge density $\sigma_j = \frac{q_j}{\lambda_x\lambda_y}$. The potential of such a charged sheet at distance z is $2\pi\sigma_j|z|$. Now we consider the contribution from a pair of image layers located at $\pm n\lambda_z$, n¿0 to the energy of a charge q_i at position $(x_i, y_i, z_i)$ in the central layer. Since $|z_j - z_i| < n\lambda_z$, we have $|z_j - z_i + n\lambda_z| = n\lambda_z + z_j - z_i$ and $|z_j - z_i - n\lambda_z| = n\lambda_z - z_j + z_i$, and hence the interaction energy from those two image layers with the charge $q_i$ vanishes by charge neutrality:

$$
2\pi q_i \sum_{j=1}^{N} \sigma_j(|z_j - z_i + n\lambda_z| + |z_j - z_i - n\lambda_z|) = 4\pi q_i n\lambda_z \sum_{j=1}^{N} \sigma_j = 0.
$$

The only errors occurring are those coming from the approximation of assuming homogeneously charged, infinite sheets instead of discrete charges. This assumption should become better when increasing the distance $n\lambda_z$ from the central layer.

However, in a naive implementation, even large gap sizes will result in large errors. This is due to the order of summation for the standard Ewald sum, which is spherical, while the above approach orders the cells in layers, called slab–wise summation. Smith has shown that by adding to the Ewald energy the term

$$E_c = 2\pi M_z^2 - \frac{2\pi M^2}{3},$$

where M is the total dipole moment, one obtains the result of a slab–wise summation instead of the spherical limit [56]. Although this is a major change in the summation order, the difference is a very simple term. In fact, Smith shows that changes of the summation order always result in a difference that depends only on the total dipole moment.

Using the far formula of MMM2D, one can calculate the contributions of the additional layers up to arbitrarily precision, even for small gap sizes. This method is called electrostatic layer correction, ELC. The advantage of this approach is that for the image layers, z is necessarily large enough, so that all interactions can be represented using the product decomposition. This allows for an order N evaluation of the ELC term.

The electrostatic layer correction term is given by

$$E_{lc} = \sum_{i,j=1}^{N} q_i q_j \psi(p_i - p_j),$$

where

$$\begin{aligned}\psi(x,y,z) = \quad & 4u_x u_y \textstyle\sum_{p,q>0} \frac{\cosh(2\pi f_{pq} z)}{f_{pq}(e^{2\pi f_{pq}\lambda_z}-1)} \cos(\omega_p x)\cos(\omega_q y) + \\ & 2u_x u_y \textstyle\sum_{p>0} \frac{\cosh(2\pi f_p z)}{f_p(e^{2\pi f_p \lambda_z}-1)} \cos(\omega_p x) + \\ & 2u_x u_y \textstyle\sum_{q>0} \frac{\cosh(2\pi f_q z)}{f_q(e^{2\pi f_q \lambda_z}-1)} \cos(\omega_q y).\end{aligned}$$

The implementation is very similar to MMM2d, except that the separation between slices closeby, and above and below is not necessary.

## E.5. Errors

Common to all algorithms of the MMM family is that accuracy is cheap with respect to computation time. More precisely, the maximal pairwise error, i.e. the maximal error of the $\psi$ expression, decreases exponentially with the cutoffs. In turn, the computation time grows logarithmically with the accuracy. This is quite in contrast to the Ewald methods, for which decreasing the error bound can lead to excessive computation time. For example, P3M cannot reach precisions above $10^{-5}$ in general. The precise form of the error estimates is of little importance here, for details see Arnold et al. [6].

One important aspect is that the error estimates are also exponential in the non-periodic coordinate. Since the number of closeby and far away particles is different for particles near the border and in the center of the system, the error distribution is highly non–homogenous. This is unproblematic as long as the maximal error is really much smaller than the thermal energy. However, one cannot interpret the error simply as an additional error source.

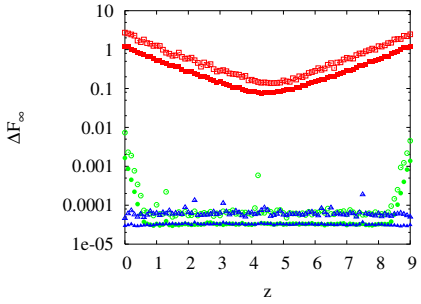



Figure E.1.: Error distribution of the ELC method.

Figure E.1 shows the error distribution of the ELC method for a gap size of 10% of the total system height. For MMM2D and MMM1D the error distribution is less homogenous, however, also here it is always better to have some extra precision, especially since it is computationally cheap.

# F. Bibliography


[1] Andersen. Molecular-Dynamics Simulations At Constant Pressure And/Or Temperature. *J. Chem. Phys*, 72(4):2384–2393, 1980. ISSN 0021-9606.

[2] Hans C. Andersen. Rattle: A "velocity" version of the shake algorithm for molecular dynamics calculations. *J. Comp. Phys.*, 51:24–34, 1983.

[3] A. Arnold and C. Holm. MMM1D: A method for calculating electrostatic interactions in 1D periodic geometries. *J. Chem. Phys.*, 123(12):144103, 2005.

[4] Axel Arnold and Christian Holm. A novel method for calculating electrostatic interactions in 2D periodic slab geometries. *Chem. Phys. Lett.*, 354:324–330, 2002.

[5] Axel Arnold and Christian Holm. MMM2D: A fast and accurate summation methodlimnb for electrostatic interactions in 2d slab geometries. *Comput. Phys. Commun.*, 148(3):327–348, 2002.

[6] Axel Arnold, Jason de Joannis, and Christian Holm. Electrostatics in Periodic Slab Geometries I+II. *J. Chem. Phys.*, 117:2496–2512, 2002.

[7] Axel Arnold, Olaf Lenz, Stefan Kesselheim, Rudolf Weeber, Florian Fahrenberger, Dominic Roehm, Peter Kosovan, and Christian Holm. ESPResSo 3.1 – molecular dynamics software for coarse-grained models. In Michael Griebel, Christian Rieger, and Marc Alexander Schweitzer, editors, *Proceedings of the Sixth International Workshop on Meshfree Methods for Partial Differential Equations*, Lecture Notes in Computational Science and Engineering. Springer, Berlin, Germany, submitted.

[8] V. Ballenegger, A. Arnold, and J. J. Cerda. Simulations of non-neutral slab systems with long-range electrostatic interactions in two-dimensional periodic boundary conditions. *J. Chem. Phys.*, 131(9):094107, 2009. doi: 10.1063/1.3216473. URL `http://link.aip.org/link/?JCP/131/094107/1`.

[9] C. W. J. Beenakker. Ewald sum of the rotne–prager tensor. *The Journal of Chemical Physics*, 85(3):1581–1582, 1986. doi: 10.1063/1.451199. URL `http://scitation.aip.org/content/aip/journal/jcp/85/3/10.1063/1.451199`.

[10] Tristan Bereau. Multi-timestep integrator for the modified andersen barostat. *in preparation.*

[11] A. Brodka. Ewald summation method with electrostatic layer correction for interactions of point dipoles in slab geometry. *Chem. Phys. Lett.*, 400:62–67, 2004.

[12] Juan J. Cerda, V. Ballenegger, O. Lenz, and C.Holm. P3M algorithm for dipolar interactions. *J. Chem. Phys.*, 129:234104, 2008.

[13] 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.

[14] I. Cimrák, M. Gusenbauer, and T. Schrefl. Modelling and simulation of processes in microfluidic devices for biomedical applications. *Computers & Mathematics with Applications*, 64(3):278–288, 2012.

[15] Lindsay M Crowl and Aaron L Fogelson. Computational model of whole blood exhibiting lateral platelet motion induced by red blood cells. *Int. J. Numer. Meth. Biomed. Engng.*, 26(3-4):471–487, 2010.

[16] M. Dao, C.T. Lim, and S. Suresh. Mechanics of the human red blood cell deformed by optical tweezers. *J. Mech. Phys. Solids*, 51:2259–2280, 2003.

[17] M. Deserno. *Counterion condensation for rigid linear polyelectrolytes*. PhD thesis, Universität Mainz, 2000.

[18] M. Deserno and C. Holm. How to mesh up Ewald sums. i. *J. Chem. Phys.*, 109: 7678, 1998.

[19] M. Deserno and C. Holm. How to mesh up Ewald sums. ii. *J. Chem. Phys.*, 109: 7694, 1998.

[20] M. Deserno, C. Holm, and H. J. Limbach. *Molecular Dynamics on Parallel Computers*, chapter How to mesh up Ewald sums. World Scientific, Singapore, 2000.

[21] M Doi and S F Edwards. *The theory of polymer dynamics*. Oxford Science Publications, 1986.

[22] Burkhard Dünweg and Anthony J.C. Ladd. Lattice Boltzmann Simulations of Soft Matter Systems. In *Lattice Boltzmann Simulations of Soft Matter Systems*, Advances in Polymer Science, pages 1–78. Springer Berlin Heidelberg, 2008. doi: 10.1007/12_2008_4. URL `http://dx.doi.org/10.1007/12_2008_4`.

[23] M.M. Dupin, I. Halliday, C.M. Care, and L. Alboul. Modeling the flow of dense suspensions of deformable particles in three dimensions. *Phys Rev E Stat Nonlin Soft Matter Phys.*, 75:066707, 2007.

[24] P.P. Ewald. Die berechnung optischer und elektrostatischer gitterpotentiale. *Ann. Phys.*, 64:253–287, 1921.

[25] F. Fahrenberger, O. A. Hickey, J. Smiatek, and C. Holm. Importance of varying permittivity on the conductivity of polyelectrolyte solutions. *Physical Review Letters*, 2015.

[26] Daan Frenkel and Berend Smit. *Understanding Molecular Simulation*. Academic Press, San Diego, second edition, 2002.

[27] G Gompper and D M Kroll. Random surface discretizations and the renormalization of the bending rigidity. *Journal de Physique I*, 6:1305–1320, 1996.

[28] Gary S. Grest and Kurt Kremer. Molecular dynamics simulation for polymers in the presence of a heat bath. *Phys. Rev. A*, 33(5):3628–31, 1986.

[29] Owen A. Hickey, Christian Holm, James L. Harden, and Gary W. Slater. Implicit Method for Simulating Electrohydrodynamics of Polyelectrolytes. *Phys. Rev. Lett.*, 105(14), SEP 29 2010. doi: {10.1103/PhysRevLett.105.148301}.

[30] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. IOP, 1988.

[31] F. Höfling, Karl-Ulrich Bamberg, and Thomas Franosch. Anomalous transport resolved in space and time by fluorescence correlation spectroscopy. *Soft Matter*, 7: 1358, 2011.

[32] Alan M. Horowitz. A generalized guided monte carlo algorithm. *Phys. Lett. B*, 268: 247 – 252, 1991.

[33] W. Humphrey, A. Dalke, and K. Schulten. VMD: Visual molecular dynamics. *J. Mol. Graphics*, 14:33–38, 1996.

[34] C. Junghans, M. Praprotnik, and K. Kremer. Transport properties controlled by a thermostat: An extended dissipative particle dynamics thermostat. *Soft Matter*, 4: 156, 2008.

[35] Stefan Kesselheim, Marcello Sega, and Christian Holm. Applying to dna translocation: Effect of dielectric boundaries. *Computer Physics Communications*, 182(1):33 – 35, 2011. ISSN 0010-4655. doi: 10.1016/j.cpc.2010.08.014. URL `http://www.sciencedirect.com/science/article/pii/S001046551000305X`. ¡ce:title¿Computer Physics Communications Special Edition for Conference on Computational Physics Kaohsiung, Taiwan, Dec 15-19, 2009¡/ce:title¿.

[36] Jiri Kolafa and John W. Perram. Cutoff errors in the ewald summation formulae for point charge systems. *Molecular Simulation*, 9(5):351–368, 1992.

[37] A. Kolb and B. Dünweg. Optimized constant pressure stochastic dynamics. *J. Chem. Phys.*, 111(10):4453–59, 1999.

[38] Timm Krüger. *Computer simulation study of collective phenomena in dense suspensions of red blood cells under shear*. PhD thesis, Universität Bochum, 2011.

[39] H. J. Limbach and C. Holm. Single-chain properties of polyelectrolytes in poor solvent. *J. Phys. Chem. B*, 107(32):8041–8055, 2003.

[40] D Magatti and F Ferri. Fast multi-tau real-time software correlator for dynamic light scattering. *Applied Optics*, 40(24):4011–4021, AUG 20 2001. ISSN 0003-6935. doi: {10.1364/AO.40.004011}.

[41] A. C. Maggs and V. Rosseto. Local simulation algorithms for coulombic interactions. *Phys. Rev. Lett.*, 88:196402, 2002.

[42] Bernward A. Mann. *The Swelling Behaviour of Polyelectrolyte Networks*. PhD thesis, Johannes Gutenberg-University, Mainz, Germany, December 2005.

[43] S. Marsili, G.F. Signorini, R. Chelli, M. Marchi, and P. Procacci. Orac: A molecular dynamics simulation program to explore free energy surfaces in biomolecular systems at the atomistic level. *J. Comp. Chem.*, 31:1106, 2009.

[44] B. Mehlig, D. W. Heermann, and B. M. Forrest. Hybrid monte carlo method for condensed-matter systems. *Phys. Rev. B*, 45:679–685, 1992.

[45] P. Nikunen, M. Karttunen, and I. Vattulainen. How would you integrate the equations of motion in dissipative particle dynamics simulations. *Com. Phys. Comm.*, 153:407, 2003.

[46] Igor Pasichnyk and Burkhard Dünweg. Coulomb interactions via local dynamics: A molecular-dynamics algorithm. *J. Phys.: Condens. Matter*, 16(38):3999–4020, September 2004.

[47] Charles S Peskin. The immersed boundary method. *Acta Numerica*, 11:479–517, 2003.

[48] Jorge Ramirez, Sathish K. Sukumaran, Bart Vorselaars, and Alexei E. Likhtman. Efficient on the fly calculation of time correlation functions in computer simulations. *J. Chem. Phys.*, 133(15):154103, OCT 21 2010. ISSN 0021-9606. doi: {10.1063/1.3491098}.

[49] D. Roehm and A. Arnold. Lattice boltzmann simulations on GPUs with ESPResSo. *Eur. Phys. J. ST*, 210:73–88, 2012.

[50] Michael Rubinstein and Ralph H. Colby. *Polymer Physics*. Oxford University Press, Oxford, UK, 2003.

[51] K. Schätzel, M. Drewel, and S Stimac. Photon-correlation measurements at large lag times - improving statistical accuracy. *Journal of Modern Optics*, 35(4):711–718, APR 1988. ISSN 0950-0340. doi: {10.1080/09500348814550731}.

[52] Heiko Schmitz and Florian Muller-Plathe. Calculation of the lifetime of positronium in polymers via molecular dynamics simulations. *J. Chem. Phys.*, 112(2):1040–1045, 2000. doi: 10.1063/1.480627. URL `http://link.aip.org/link/?JCP/112/1040/1`.

[53] M. Sega, M. Sbragaglia, S. S. Kantorovich, and A. O. Ivanov. Mesoscale structures at complex fluidfluid interfaces: a novel lattice boltzmann/molecular dynamics coupling. *Soft Matter*, 2013, in press. doi: 10.1039/C3SM51556G.

[54] X. Shan and H. Chen. Lattice boltzmann model for simulating flows with multiple phases and components. *Phys. Rev. E*, 47:1815, 1993.

[55] J. Smiatek, M. P. Allen, and F. Schmidt. Tunable slip boundaries for coarse-grained simulations of fluid flow. *Eur. Phys. J. E*, 26:115, 2008.

[56] E. R. Smith. Electrostatic energy in ionic crystals. *Proc. R. Soc. Lond. A*, 375: 475–505, 1981.

[57] T. Soddemann, B. Dünweg, and K. Kremer. A generic computer model for amphiphilic systems. *Eur. Phys. J. E*, 6:409, 2001.

[58] T. Soddemann, B. Dünweg, and K. Kremer. Dissipative particle dynamics: A useful thermostat for equilibrium and nonequilibrium molecular dynamics simulations. *Phys. Rev. E*, 68:046702, 2003.

[59] R. Strebel. *Pieces of software for the Coulombic m body problem.* Dissertation, ETH Zürich, 1999. URL `http://e-collection.ethbib.ethz.ch/show?type=diss&nr=13504`.

[60] S. Succi. *The lattice Boltzmann equation for fluid dynamics and beyond.* Oxford University Press, USA, 2001.

[61] A. P. Thompson, S. J. Plimpton, and W. Mattson. General formulation of pressure and stress tensor for arbitrary many-body interaction potentials under periodic boundary conditions. *Journal of Chemical Physics*, 131:154107, 2009.

[62] C. Tyagi, M. Süzen, M. Sega, M. Barbosa, S. Kantorovich, and C. Holm. An iterative, fast, linear-scaling method for computing induced charges on arbitrary dielectric boundaries. *J. Chem. Phys.*, 132:154112, 2010. doi: 10.1063/1.3376011.

[63] S. Tyagi, A. Arnold, and C. Holm. ICMMM2D: An accurate method to include planar dielectric interfaces via image charge summation. *J. Chem. Phys.*, 127: 154723, 2007.

[64] Sandeep Tyagi, Axel Arnold, and Christian Holm. Electrostatic layer correction with image charges: A linear scaling method to treat slab 2d + h systems with dielectric interfaces. *J. Chem. Phys.*, 129(20):204102, 2008.

[65] Ulli Wolff. Monte carlo errors with less errors. *Comput. Phys. Commun.*, 156: 143–153, 2004.

# Index


Affinity interaction, **65**
aggregation, **139**
analysis, 131
    aggregation, **139**
    bond distances internal first monomer, **148**
    bond lengths, **148**
    center of mass, **139**
    chains, **146**
    end-to-end distance of a chain, **146**
    energies, **142**
    finding holes, **140**
    fluid temperature, **141**
    form factor of a chain, **148**
    gyration tensor, **139**
    hydrodynamic radius of a chain, **147**
    internal distances within a chain, **147**
    local stress tensor, **144**
    minimal particle distance, **131**
    moment of inertia matrix, **139**
    particle distance, **131**
    particle distribution, **132**
    particles in the neighbourhood, **132**
    pearl-necklace structures, **140**
    pressure, **142**
    principal axis of the moment of inertia, **139**
    radial distribution function, **149**
    radial distribution function $g(r)$, **137**
    radius of gyration of a chain, **146**
    stress tensor, **143**
    structure factor $S(q)$, **137**
    system momentum, **141**
    topologies, **145**
    van Hove autocorrelation function $G(r,t)$, **138**
Analysis in the Core, 153
`analyze` (Tcl-command), **131**
Anisotropic interactions, 63

binary I/O, 177
`blockfile` (Tcl-command), **173**
blocks, 175
BMHTF interaction, **59**
bond distances internal first monomer, **148**
bond lengths, **148**
bond-angle interactions, **74**
bonded coulomb bond, **68**
bonded interaction type id, 65
bonded interactions, **65**
bonded interactions oif, **70**
`box_l` (global variable), **100**
Buckingham interaction, **60**
build directory, 27

`cell_grid` (global variable), **100**
`cell_size` (global variable), **100**
`cellsystem` (Tcl-command), **110**
center of mass, **139**
chains, **146**
`change_volume` (Tcl-command), **120**
configuration header, 30
configure, 15, 27
configure options, 27
`constraint` (Tcl-command), **46**
`copy_particles` (Tcl-command), **45**
`correlation` (Tcl-command), **160**
Correlations, 160
Coulomb interactions, **77**
`counterions` (Tcl-command), **41**
`crosslink` (Tcl-command), **44**

DAWAANR method, **94**
Debye-Hückel potential, **82**
`diamond` (Tcl-command), **42**
`dielectric` (Tcl-command), **92**
Dielectric interfaces, 87, 91, 92
dihedral interactions, **76**
Dipolar direct sum on gpu, **95**
Dipolar interactions, **92**
Directional Lennard-Jones interaction, **63**
DLC method, **94**
domain decomposition, 110
DPD, 97, **105**
DPD interaction, **97**
`dpd_gamma` (global variable), **100**
`dpd_ignore_fixed_particles` (global variable), **100**
`dpd_r_cut` (global variable), **100**

ELC method, **89**
`electrokinetics` (Tcl-command), **203**
Electrostatic interactions, **77**
end-to-end distance of a chain, **146**
energies, **142**
energy unit, 12
EwaldGPU method, **80**

features, 25, 30, **267**
- ADDITIONAL_CHECKS, **271**
- ADRESS, **268**
- ASYNC_BARRIER, **272**
- BMHTF_NACL, **270**
- BOND_ANGLE, **271**
- BOND_ANGLEDIST, **271**
- BOND_CONSTRAINT, **268**
- BOND_ENDANGLEDIST, **271**
- BOND_VIRTUAL, **268**
- BUCKINGHAM, **270**
- CATALYTIC_REACTIONS, **269**
- CELL_DEBUG, **271**
- COLLISION_DETECTION, **269**
- COMFIXED, **268**
- COMFORCE, **268**
- COMM_DEBUG, **271**
- CONSTRAINTS, **268**
- DIPOLES, **267**
- DPD, **269**
- DPD_MASS_LIN, **269**
- DPD_MASS_RED, **269**
- ELECTROSTATICS, **267**
- ESK_DEBUG, **272**
- ESR_DEBUG, **272**
- EVENT_DEBUG, **271**
- EWALD_DEBUG, **272**
- EXCLUSIONS, **268**
- EXTERNAL_FORCES, **267**
- FENE_DEBUG, **272**
- FFT_DEBUG, **272**
- FORCE_CORE, **273**
- FORCE_DEBUG, **272**
- GAY_BERNE, **270**
- GHOST_DEBUG, **271**
- GHOST_FORCE_DEBUG, **271**
- GRID_DEBUG, **272**
- H5MD, **269**
- HALO_DEBUG, **272**
- HERTZIAN, **270**
- INTEG_DEBUG, **271**
- INTER_DPD, **269**
- INTER_RF, **269**
- LANGEVIN_PER_PARTICLE, **267**
- LANGEVIN_PER_PARTICLE, **268**
- LATTICE_DEBUG, **272**
- LB, **269**
- LB_DEBUG, **272**
- LB_ELECTROHYDRODYNAMICS, **269**
- LB_GPU, **269**
- LENNARD_JONES, **270**
- LENNARD_JONES_GENERIC, **270**
- LJ_ANGLE, **270**
- LJ_DEBUG, **272**
- LJ_WARN_WHEN_CLOSE, **271**
- LJCOS, **270**
- LJCOS2, **270**
- MAGGS_DEBUG, **272**
- MASS, **268**
- MEM_DEBUG, **271**
- METADYNAMICS, **268**

MODES, **268**
MOL_CUT, **270**
MOLFORCES, **268**
MOLFORCES_DEBUG, **272**
MORSE, **270**
MORSE_DEBUG, **272**
MPI_CORE, **273**
NEMD, **269**
NO_INTRA_NB, **270**
NPT, **269**
OLD_RW_VERSION, **269**
OLD_DIHEDRAL, **271**
ONEPART_DEBUG, **272**
OVERLAPPED, **269**
P3M_DEBUG, **272**
PARTIAL_PERIODIC, **267**
PARTICLE_DEBUG, **272**
POLY_DEBUG, **272**
PTENSOR_DEBUG, **272**
RANDOM_DEBUG, **272**
ROTATION, **267**
ROTATION_PER_PARTICLE, **267**
ROTATIONAL_INERTIA, **267**
SD, **270**
SD_DEBUG, **270**, **273**
SD_FF_ONLY, **270**
SD_NOT_PERIODIC, **270**
SD_USE_FLOAT, **270**
SHANCHEN, **269**
SMOOTH_STEP, **270**
SOFT_SPHERE, **270**
STAT_DEBUG, **272**
TABULATED, **270**
THERMO_DEBUG, **272**
THERMOSTAT_IGNORE_NON_VIRTUAL, **268**
TRANS_DPD, **269**
TUNABLE_SLIP, **268**
VERLET_DEBUG, **272**
VIRTUAL_SITES_COM, **268**
VIRTUAL_SITES_DEBUG, **272**
VIRTUAL_SITES_NO_VELOCITY, **268**
VIRTUAL_SITES_RELATIVE, **268**
VIRTUAL_SITES_THERMOSTAT, **268**
FENE bond, **66**
FFTW, 14
finding holes, **140**
fluid temperature, **141**
form factor of a chain, **148**

`gamma` (global variable), **100**
Gaussian interaction, **62**
Gay-Berne interaction, **64**
Generic Lennard-Jones interaction, **57**
global variables, 173
`box_l`, **100**
`cell_grid`, **100**
`cell_size`, **100**
`dpd_gamma`, **100**
`dpd_ignore_fixed_particles`, **100**
`dpd_r_cut`, **100**
`gamma`, **100**
`integ_switch`, **100**
`lb_components`, **100**
`local_box_l`, **100**
`max_cut_bonded`, **101**
`max_cut_nonbonded`, **100**
`max_cut`, **100**
`max_num_cells`, **101**
`max_part`, **101**
`max_range`, **101**
`max_skin`, **101**
`min_global_cut`, **101**
`min_num_cells`, **101**
`n_layers`, **101**
`n_nodes`, **101**
`n_part_types`, **101**
`n_part`, **101**
`node_grid`, **101**
`npt_p_ext`, **101**
`npt_p_inst`, **101**
`npt_piston`, **101**
`nptiso_gamma0`, **101**
`nptiso_gammav`, **101**
`periodicity`, **101**
`sd_precision_random`, **102**, **129**
`sd_radius`, **102**, **129**

sd_random_state, **102**, **129**
sd_seed, **102**, **129**
sd_viscosity, **102**, **129**
skin, **102**
temperature, **102**
thermo_switch, **102**
time_step, **102**
time, **102**
timings, **102**
transfer_rate, **102**
verlet_flag, **102**
verlet_reuse, **102**
warnings, **102**
gyration tensor, **139**

harmonic bond, **67**
harmonic dumbbell bond, **67**
harmonic-well (Tcl-command), **50**
hat interaction, **61**
Hertzian interaction, **62**
hydrodynamic radius of a chain, **147**

ICC⋆, **91**
iccp3m (Tcl-command), **91**
icosaeder (Tcl-command), **43**
IMD, 183
imd (Tcl-command), **183**
Installation, **25**
integ_switch (global variable), **100**
integrate (Tcl-command), **118**
integrate_sd (Tcl-command), **129**
inter (Tcl-command), **55**
Interaction DPD, 107
interactions, **55**
Affinity, **65**
BMHTF, **59**
bond-angle, **74**
bonded, **65**
bonded oif, **70**
bonded_coulomb, **68**
Buckingham, **60**
Coulomb, **77**
DAWAANR method, **94**
Debye-Hückel, **82**
dihedral, **76**
Dipolar, **92**
Directional Lennard-Jones, **63**
DLC method, **94**
DPD, **97**
ELC method, **89**
Electrostatic, **77**
EwaldGPU, **80**
FENE, **66**
gaussian, **62**
Gay-Berne, **64**
Generic Lennard-Jones, **57**
harmonic, **67**
harmonic dumbbell, **67**
hat, **61**
hertzian, **62**
Lennard-Jones, **56**
Lennard-Jones cosine, **58**
Maggs method, **85**
Magnetostatic, **92**
MDDS method, **95**
membrane-collision, **61**
MEMD, **85**
MMM1D, **84**
MMM2D, **83**
Morse, **60**
non-bonded, **55**
oif global force, **73**
oif local force, **70**
out direction, **74**
P3M, **78**
quartic, **68**
rigid bond, **69**
smooth-step, **59**
soft-sphere, **60**
subtracted Lennard-Jones, **68**
tabulated, **56**
tabulated bond, **69**
Tunable-slip boundary interactions, **96**
interactive mode, 30
internal distances within a chain, **147**

label:DPDthermostat, 105

`lb` (Tcl-command), **194**
`lb_components` (global variable), **100**
Lees-Edwards Boundaries, 37, **121**
`lees_edwards_offset` (Tcl-command), **121**
length unit, 12
Lennard-Jones cosine interaction, **58**
Lennard-Jones interaction, **56**
local stress tensor, **144**
`local_box_l` (global variable), **100**

Maggs method, **85**
Magnetostatic interactions, **92**
make, 15
`max_cut` (global variable), **100**
`max_cut_bonded` (global variable), **101**
`max_cut_nonbonded` (global variable), **100**
`max_num_cells` (global variable), **101**
`max_part` (global variable), **101**
`max_range` (global variable), **101**
`max_skin` (global variable), **101**
Maxwell Equation Molecular Dynamics, **85**
MDDS method, **95**
membrane-collision interaction, **61**
MEMD, **85**
`metadynamics` (Tcl-command), **127**
`min_global_cut` (global variable), **101**
`min_num_cells` (global variable), **101**
minimal particle distance, **131**
`minimize_energy` (Tcl-command), **119**
MMM1D method, **84**
MMM2D method, **83**
moment of inertia matrix, **139**
momentum exchange method, 109
Morse interaction, **60**
MPI, 14
mpiio, 177
Multiple tau correlator, 164
`multitimestepping` (Tcl-command), **130**
myconfig.hpp, 30

`n_layers` (global variable), **101**
`n_nodes` (global variable), **101**
`n_part` (global variable), **101**
`n_part_types` (global variable), **101**
NEMD, 109
`nemd` (Tcl-command), **109**
`node_grid` (global variable), **101**
Non-bonded interactions, **55**
`npt_p_ext` (global variable), **101**
`npt_p_inst` (global variable), **101**
`npt_piston` (global variable), **101**
`nptiso_gamma0` (global variable), **101**
`nptiso_gammav` (global variable), **101**

`observable` (Tcl-command), **153**
Observables, 153
`oif` (Tcl-command), **213**
oif global force, **73**
oif local force, **70**
out direction, **74**

P3M method, **78**
`parallel_tempering` (Tcl-command), **123**
`part` (Tcl-command), **32**
particle distance, **131**
particle distribution, **132**
particles in the neighbourhood, **132**
pearl-necklace structures, **140**
`periodicity` (global variable), **101**
physical units, 12
`polymer` (Tcl-command), **39**
`prepare_vmd_connection` (Tcl-command), **184**
pressure, **142**
principal axis of the moment of inertia, **139**

quartic bond, **68**
quick reference of Tcl-commands, 252

radial distribution function, **149**
radial distribution function $g(r)$, **137**
radius of gyration of a chain, **146**
random number generators, 174
random seed, 174
Rattle Shake algorithm, 69
`readpdb` (Tcl-command), **182**

requirements, 14
rigid bond, **69**
rotate_system (Tcl-command), **121**

salt (Tcl-command), **41**
Scafacos, **89**, **95**
sd_precision_random (global variable), **102**, **129**
sd_radius (global variable), **102**, **129**
sd_random_state (global variable), **102**, **129**
sd_seed (global variable), **102**, **129**
sd_viscosity (global variable), **102**, **129**
setmd (Tcl-command), **100**
shear boundaries, 121
shear viscosity, 121
shear-rate method, 109
skin (global variable), **102**
smooth-step interaction, **59**
soft-sphere interaction, **60**
sort_particles (Tcl-command), **123**
source directory, 27
stored configurations, 150, 175
stress tensor, **143**
structure factor $S(q)$, **137**
subtracted Lennard-Jones bond, **68**
system momentum, **141**

tabulated bond interactions, **69**
tabulated interaction, **56**
Tcl global variables, 173
Tcl-commands
  analyze, **131**
  blockfile, **173**
  cellsystem, **110**
  change_volume, **120**
  constraint, **46**
  copy_particles, **45**
  correlation, **160**
  counterions, **41**
  crosslink, **44**
  diamond, **42**
  dielectric, **92**
  electrokinetics, **203**
  harmonic-well, **50**
  iccp3m, **91**
  icosaeder, **43**
  imd, **183**
  integrate, **118**
  integrate_sd, **129**
  inter, **55**
  lb, **194**
  lees_edwards_offset, **121**
  metadynamics, **127**
  minimize_energy, **119**
  multitimestepping, **130**
  nemd, **109**
  observable, **153**
  oif, **213**
  parallel_tempering, **123**
  part, **32**
  polymer, **39**
  prepare_vmd_connection, **184**
  readpdb, **182**
  rotate_system, **121**
  salt, **41**
  setmd, **100**
  sort_particles, **123**
  thermostat, **102**
  time_integration, **119**
  tune_skin, **120**
  uwerr, **151**
  velocities, **122**
  writepdb, **182**
  writepdbfoldchains, **182**
  writepdbfoldtopo, **182**
  writepsf, **182**
  writevcf, **180**
  writevsf, **179**
  writevtk, **181**
Tcl/Tk, 14
temperature (global variable), **102**
thermo_switch (global variable), **102**
thermostat (Tcl-command), **102**
time (global variable), **102**
time unit, 12
time_integration (Tcl-command), **119**
time_step (global variable), **102**

`timings` (global variable), **102**
topologies, **145**
`transfer_rate` (global variable), **102**
Tunable-slip boundary interaction, **96**
`tune_skin` (Tcl-command), **120**

units, 12
`uwerr` (Tcl-command), **151**

van Hove autocorrelation function $G(r, t)$, **138**
vcf, **178**
`velocities` (Tcl-command), **122**
`verlet_flag` (global variable), **102**
`verlet_reuse` (global variable), **102**
virtual sites, **50**
vsf, **178**
vtf, **178**

`warnings` (global variable), **102**
whitespace, 173
`writepdb` (Tcl-command), **182**
`writepdbfoldchains` (Tcl-command), **182**
`writepdbfoldtopo` (Tcl-command), **182**
`writepsf` (Tcl-command), **182**
`writevcf` (Tcl-command), **180**
`writevsf` (Tcl-command), **179**
`writevtk` (Tcl-command), **181**