

# **ESPResSo User's Guide**

for version 3.1.2

March 8, 2013



# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Guiding principles . . . . .	7
1.2	Available simulation methods . . . . .	8
1.3	Basic program structure . . . . .	9
1.4	On units . . . . .	9
1.5	Requirements . . . . .	10
1.6	Syntax description . . . . .	10
<b>2</b>	<b>First steps</b>	<b>12</b>
2.1	Quick installation . . . . .	12
2.2	Running ESPResSo . . . . .	13
2.3	Creating the first simulation script . . . . .	13
2.4	tutorial.tcl . . . . .	19
<b>3</b>	<b>Getting, compiling and running ESPResSo</b>	<b>20</b>
3.1	Running configure . . . . .	20
3.2	make: Compiling, testing and installing ESPResSo . . . . .	22
3.3	Running ESPResSo . . . . .	23
3.4	myconfig.h: Activating and deactivating features . . . . .	24
<b>4</b>	<b>Setting up particles</b>	<b>26</b>
4.1	part: Creating single particles . . . . .	26
4.2	Creating groups of particle . . . . .	30
4.3	constraint: Setting up constraints . . . . .	36
4.4	Virtual sites . . . . .	38
<b>5</b>	<b>Setting up interactions</b>	<b>42</b>
5.1	Isotropic non-bonded interactions . . . . .	42
5.2	Anisotropic non-bonded interactions . . . . .	47
5.3	Bonded interactions . . . . .	50
5.4	Bond-angle interactions . . . . .	53
5.5	Dihedral interactions . . . . .	54
5.6	Coulomb interaction . . . . .	55
5.7	Dipolar interaction . . . . .	62
5.8	Special interaction commands . . . . .	65

<b>6</b>	<b>Setting up the system</b>	<b>68</b>
6.1	setmd: Setting global variables. . . . .	68
6.2	thermostat: Setting up the thermostat . . . . .	70
6.3	nemd: Setting up non-equilibrium MD . . . . .	73
6.4	cellsystem: Setting up the cell system . . . . .	74
6.5	AdResS . . . . .	76
6.6	CUDA . . . . .	76
6.7	Creating bonds when particles collide . . . . .	77
<b>7</b>	<b>Running the simulation</b>	<b>79</b>
7.1	integrate: Running the simulation . . . . .	79
7.2	change_volume: Changing the box volume . . . . .	79
7.3	Stopping particles . . . . .	80
7.4	velocities: Setting the velocities . . . . .	80
7.5	invalidate_system . . . . .	80
7.6	Parallel tempering . . . . .	81
7.7	Metadynamics . . . . .	85
<b>8</b>	<b>Analysis in Tcl</b>	<b>87</b>
8.1	Available observables . . . . .	87
8.2	Analyzing groups of particles (molecules) . . . . .	97
8.3	Storing configurations . . . . .	101
8.4	uwerr: Computing statistical errors in time series . . . . .	103
<b>9</b>	<b>Analysis in the core</b>	<b>104</b>
9.1	Observables . . . . .	104
9.2	Correlations . . . . .	108
<b>10</b>	<b>Input / Output</b>	<b>115</b>
10.1	blockfile: Using the structured file format . . . . .	115
10.2	Checkpointing (deprecated) . . . . .	119
10.3	Writing VTF files . . . . .	121
10.4	Writing PDB/PSF files . . . . .	124
10.5	Online-visualisation with VMD . . . . .	125
10.6	Errorhandling . . . . .	127
<b>11</b>	<b>Auxilliary commands</b>	<b>128</b>
11.1	Center of mass motion . . . . .	128
11.2	Finding particles and bonds . . . . .	128
11.3	Additional Tcl math-functions . . . . .	130
11.4	Checking for features of ESPResSo . . . . .	136
<b>12</b>	<b>Lattice-Boltzmann</b>	<b>137</b>
12.1	Setting up a LB fluid . . . . .	137
12.2	LB as a thermostat . . . . .	139

12.3	Reading and setting single lattice nodes . . . . .	139
12.4	Setting up boundary conditions . . . . .	140
12.5	Choosing between the GPU and CPU implementations . . . . .	141
12.6	Electrohydrodynamics . . . . .	142
<b>13</b>	<b>External package: mbtools</b>	<b>143</b>
13.1	Introduction . . . . .	143
13.2	Installing and getting started . . . . .	144
13.3	The <code>main.tcl</code> script . . . . .	145
13.4	Analysis . . . . .	147
13.5	System generation . . . . .	151
13.6	Utils . . . . .	158
13.7	mmsg . . . . .	164
<b>14</b>	<b>Under the hood</b>	<b>167</b>
14.1	Internal particle organization . . . . .	167
<b>15</b>	<b>Getting involved</b>	<b>169</b>
15.1	Community support and mailing lists . . . . .	169
15.2	Contributing your own code . . . . .	170
15.3	Developers' guide . . . . .	170
15.4	User's guide . . . . .	170
<b>A</b>	<b>ESPResSo quick reference</b>	<b>171</b>
<b>B</b>	<b>Features</b>	<b>181</b>
B.1	General features . . . . .	181
B.2	Interactions . . . . .	183
B.3	Debug messages . . . . .	184
<b>C</b>	<b>Sample scripts</b>	<b>187</b>
<b>D</b>	<b>Maxwell Equations Molecular Dynamics (MEMD)</b>	<b>189</b>
D.1	Equations of motion . . . . .	189
D.2	Discretization . . . . .	190
D.3	Initialization of the algorithm . . . . .	190
D.4	Time integrator . . . . .	191
D.5	Self-energy . . . . .	192
D.6	For which systems to use the algorithm . . . . .	193
<b>E</b>	<b>The MMM family of algorithms</b>	<b>195</b>
E.1	Introduction . . . . .	195
E.2	MMM2D . . . . .	197
E.3	MMM1D . . . . .	199
E.4	ELC . . . . .	200

E.5 Errors . . . . .	201
<b>F Bibliography</b>	<b>203</b>
<b>Index</b>	<b>208</b>

# 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.* [19]. 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 Tcl 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

- Ensembles:
  - NVE
  - NVT
  - NpT
- Algorithms for charged systems:
  - P3M for fully periodic systems
  - ELC and MMM-family of algorithms for charged systems with non-periodic boundary conditions
  - MEMD (Maggs algorithm)
- Hydrodynamics:
  - DPD (as a thermostat)
  - Lattice-Boltzmann
- Non-equilibrium MD to simulate shear flow
- Parallel tempering
- Metadynamics



- Rigid bodies via virtual sites
- AdResS

### 1.3. Basic program structure

As already mentioned, ESPResSo consists of two components. The simulation engine is written in C for the sake of computational efficiency. The steering or control level is interfaced to the kernel via an interpreter of the Tcl 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 Tcl interpreter contains several special commands as an extension to Tcl, which provide the interaction with the simulation engine. Thus, the user has at hand the full realm of Tcl commands and constructs plus a few commands to communicate with the simulation engine. The interfacing commands are designed so that they can both set properties of the system (set up particles, interactions, thermostat) and retrieve information about the already set-up entities. The standard Tcl constructs allow for a flexible decision-making in the course of the simulation. This can be for example exploited to check whether a simulation has reached the desired state. 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 Tcl 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 Tcl commands, their behaviour and use in the simulation. It only describes certain technical details of implementation which are necessary for understanding how the commands work. Technical documentation of the code and program structure is contained in the Developers' guide (see section 15.3).

### 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. 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. The user can choose the unit system that suits the system best.

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.

The second choice to be made is the energy (and time-) scale. One can for example choose to set the Lennard-Jones parameter  $\epsilon$  to the energy in  $\frac{kJ}{mol}$ . 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.

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 <sup>1</sup> 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<sup>3</sup>M), ESPResSo needs the FFTW library version 3 or later <sup>2</sup> 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. 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.

---

<sup>1</sup><http://www.tcl.tk/>

<sup>2</sup><http://www.fftw.org/>

- [*argument*] specifies, that the augment *argument* is optional, *i.e.* it can be omitted.
- When an optional argument or a whole command is marked by a superscript label (<sup>1</sup>), this denotes that the argument can only be used, when the corresponding feature (see appendix B on page 181) 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 1ELECTROSTATICS 2ROTATION 3DIPOLES

```

## 2. First steps

### 2.1. Quick installation

If you have installed the requirements (see section 1.5 on page 10) 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 3.1.1 on page 21). Furthermore, many features of ESPResSo can be selectively turned on or off in the local configuration header (see section 3.4 on page 24) 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 3.1 on page 20.

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 3.2 on page 22.

When these steps have successfully completed, ESPResSo can be started with the command (see section 3.3 on page 23)

Espresso *file*

## 2.2. Running ESPResSo

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 <sup>1</sup>.

## 2.3. 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

---

<sup>1</sup><http://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html>

```

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 [20], 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<sup>3</sup>M 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 <sup>2</sup>:

```
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  $20 \times \text{integ\_steps}$  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 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 ljforcecap $cap; integrate $integ_steps
}
inter ljforcecap 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 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
```

---

<sup>2</sup>There also exists a Tcl function `degrees_of_freedom` which does the same.

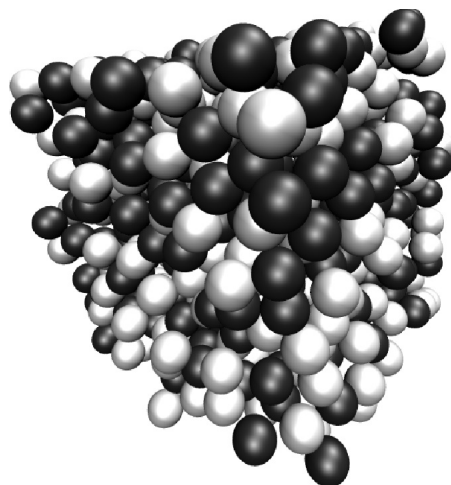


Figure 2.1.: VMD Snapshot of the salt system

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:

```
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 *n<sub>n</sub>odes script [config... ]*

where *n<sub>n</sub>odes* is the number of CPUs ESPReso should be running on.

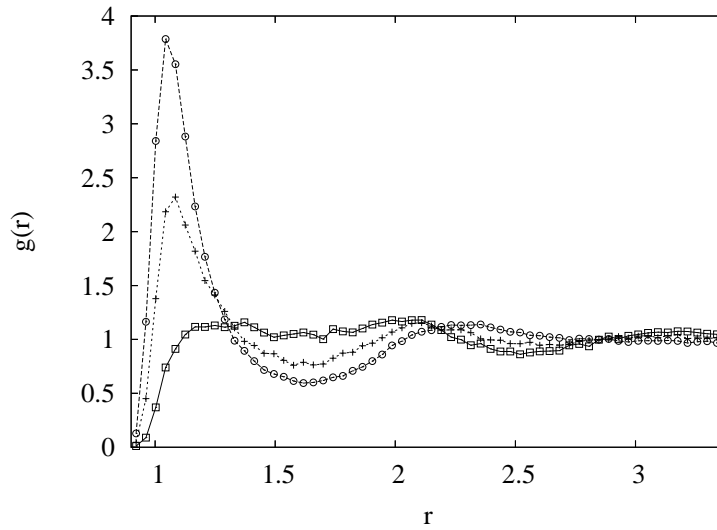


Figure 2.2.: Radial distribution functions  $g_{++}(r)$  between equal charges (rectangles) and  $g_{+-}(r)$  for opposite charges (circles). The plus symbols denote  $g(r)$  for an uncharged system.

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<sup>3</sup>M.

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 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.

## 2.4. 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<sup>1</sup>. 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`<sup>2</sup> from ESPResSo's project page at the Savannah GNU server<sup>3</sup>. 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. 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 the GNU autotools, which are developed since more than 20 years and allow to compile software easily on a wide range of platforms.

### 3.1. 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 ...]
```

---

<sup>1</sup><http://espressomd.org>

<sup>2</sup><http://git.org>

<sup>3</sup><https://savannah.nongnu.org/projects/espressomd/>

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.

### 3.1.1. 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 is 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.

**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
```

### 3.1.2. Options

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

```
configure --help
```

`--with-myconfig=MYCONFIG_HEADER` This option sets the name of the local configuration header (see 3.4 on page 24). It defaults to `"myconfig.h"`.

`--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. <sup>4</sup> Efence is not used by default.

---

<sup>4</sup><http://freshmeat.net/projects/efence/>

**--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 models 1.1 and 2.0, *i.e.* everything with a G90 chip or newer. Note that we require at least compute model 1.1.

## 3.2. 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 `madelung.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.h internal"`

will create the archive file and include the file `myconfig.h` 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`.

**uninstall** Uninstalls ESPResSo, *i.e.* removes all files that were installed during `make install`. The variables are identical to the variables of the `install`-target.

**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.

### 3.3. 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. *N\_processors* is the number of processors that are to be used. Any further arguments are passed to the script. 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.

### 3.4. myconfig.h: 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 181 lists and describes all available features. When no configuration header is provided by the user, a default header, found in `src/myconfig-default.h`, will be used that turns on the default features. The file `myconfig-sample.h` in the source directory contains a list of all possible features that can be copied into your own configuration file.

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.h`. The name of the configuration header can be changed either when the `configure-script` is called with the option `--with-myconfig` (see section 3.1 on page 20), or when `make` is called with the setting `myconfig=myconfig_header` (see section 3.2 on page 22).

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:

- `$builddir1/myconfig.h`:  
#define ELECTROSTATICS  
#define LENNARD-JONES
- `$builddir2/myconfig.h`:  
#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

##### Syntax

```
part pid [pos x y z] [type typeid] [v vx vy vz] [f fx fy fz]  
    [bond bondid pid2 ...] [temp T]9 [gamma g]9 [q charge]1  
    [quat q1 q2 q3 q4]2 [omega x y z]2 [torque x y z]2  
    [rinertia x y z]2 [[un]fix x y z]3 [ext_force x y z]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  
Required features: 1ELECTROSTATICS 2ROTATION 3EXTERNAL_FORCES 4EXCLUSION  
5MASS 6DIPOLES 7VIRTUAL_SITES_COM 8VIRTUAL_SITES_RELATIVE  
9LANGEVIN_PER_PARTICLE
```

##### 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 42) 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.
- [*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.

- `[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 than 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 than the Langevin thermostat.
- `[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  $x\ y\ z$ ]` Sets the rotational velocity of this particle in the particle's co-rotating frame.
- `[torque  $x\ y\ z$ ]` Sets the rotational force of this particle, in global frame. When printing the values using `part id_particle print`, there is an alternative named `[tbf]` which gives you the values of the torque in the body frame. Be aware: the values obtained when printing using `[torque]` are computed in the frame laboratory and are the ones one should usually look at. Nonetheless, in case you introduce torques using `[torque]` option, espresso will assume they are given in the body-frame. Thus `[tbf]` is useful to know which should be the numerical values you should reintroduce in order to have exactly the same conformation.
- `[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 `fix001` will fix motion only in  $z$ . Note that `fix` without arguments is equivalent to `fix111`.
- `[ext_force  $x\ y\ z$ ]` An additional external force is applied to the particle.
- `[unfix]` Release any external influence from the particle.
- `[exclude  $pid2\ \dots$ ]` Restrictions:  $pid2$  must be an existing particle. Between the current particle and 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  $pid2\ \dots$ ]` 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 dipol 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.

#### 4.1.2. Getting particle properties

##### Syntax

```
(1) part pid print [( id | pos | type | folded_position | type | q | v | f
    | fix | ext_force | bond | connections [range] )]...
(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.

##### 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...{{...}...}}
...
```

### 4.1.3. Deleting particles

#### 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

#### 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 ... 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

#### 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_charged]1 [types typeid_neutral [typeid_charged]]
      [bond bondid] [angle  $\phi$  [ $\theta$  [x y z]]] [constraints]2
Required features: 1 ELECTROSTATICS 2 CONSTRAINTS
```

#### Description

This command will create *num\_polymers* polymer or polyelectrolyte chains with *monomers\_per\_chain* monomers per chain. The length of the bond between two adjacent monomers will be set up to be *bond\_length*.

#### Arguments

- *num\_polymers* Sets the number of polymer chains.
- *monomers\_per\_chain* 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]* Sets the particle number of the start monomer to be used with the *part* command. This defaults to 0.
- *[pos x y z]* 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)** The monomers are randomly placed by a random walk with a steps size of *bond\_length*.
  - PSAW (Pruned self-avoiding walk)** 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)** 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  $d_{\text{charged}}$ ]** Sets the stride between the indices of two charged monomers. This defaults to 1, meaning that all monomers in the chain are charged.
- **[types  $typeid_{\text{neutral}}$   $typeid_{\text{charged}}$ ]** 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]** 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  $\phi$  [ $\theta$  [ $x$   $y$   $z$ ]]]** 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 36).

#### 4.2.2. counterions: Setting up counterions

##### Syntax

```
counterions N [start pid] [mode ( SAW | RW ) [shield [try_max ]]]
           [charge val]1 [type typeid]
Required features: 1ELECTROSTATICS
```

##### 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 [trymax ]]]` 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

##### Syntax

```
| salt N+ N- [start pid] [mode ( SAW | RW ) [shield [trymax ]]]
|      [charges val+ [val-]]1 [types typeid+ [typeid-]] [rad r]
| Required features: 1ELECTROSTATICS
```

##### 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 [trymax ]]]` 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

##### Syntax

```
| diamond a bond_length monomers_per_chain [counterions NCI]
|      [charges valnode valmonomer valCI]1 [distance dcharged]1 [nonet]
| Required features: 1ELECTROSTATICS
```

##### Description

Creates a diamond-shaped polymer network with 8 tetra-functional nodes connected by  $2 * 8$  polymer chains of length *monomers\_per\_chain* 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.

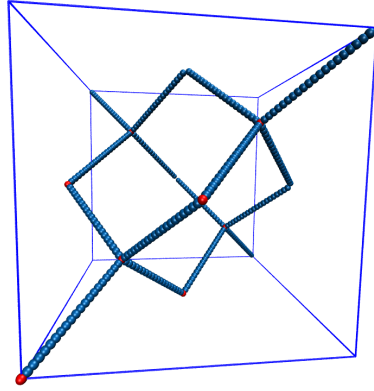


Figure 4.1.: Diamond-like polymer network with *monomers\_per\_chain*=15.

#### 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* Sets the number of chain monomers between the functional nodes.
- [*counterions*  $N_{CI}$ ] Adds  $N_{CI}$  counterions to the system.
- [*charges*  $val_{node}$   $val_{monomer}$   $val_{CI}$ ] Sets the charge of the nodes to  $val_{node}$ , the charge of the connecting monomers to  $val_{monomer}$ , and the charge of the counterions to  $val_{CI}$ .
- [*distance*  $d_{charged}$ ] Specifies the distance between charged monomers along the interconnecting chains. If  $d_{charged} > 1$  the remaining chain monomers are uncharged.
- [*nonet*] Do not create bonds between the chains.

#### 4.2.5. icosaeader: Setting up an icosaeader

##### Syntax

```
icosaeader a monomers_per_chain [counterions  $N_{\text{CI}}$ ]  
          [charges  $val_{\text{monomers}}$   $val_{\text{CI}}$ ]1 [distance  $d_{\text{charged}}$ ]1  
Required features: 1ELECTROSTATICS
```

##### Description

Creates a modified icosaeader to model a fullerene (or soccer ball). The edges are modeled by polymer chains connected at the corners of the icosaeader. 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.

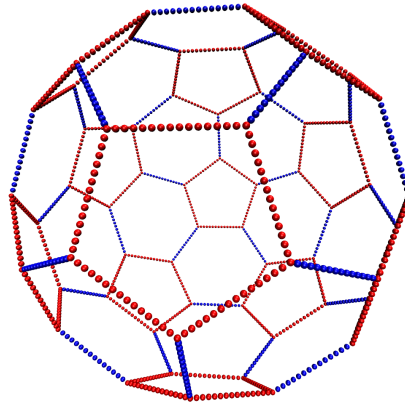


Figure 4.2.: Icosaeder with *monomers\_per\_chain*=15.

##### Arguments

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

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

##### 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 successfully 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\_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

##### 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. 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.

### 4.3. constraint: Setting up constraints

#### Syntax

- (1) `constraint wall normal  $n_x$   $n_y$   $n_z$  dist  $d$  type  $id$  [penetrable  $flag$ ]  
[reflecting  $flag$ ]`
  - (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  $rad$  length  
 $length$  type  $id$`
  - (7) `constraint rod center  $c_x$   $c_y$  lambda  $lambda$  1`
  - (8) `constraint plate height  $h$  sigma  $sigma$  1`
  - (9) `constraint ext_magn_field  $f_x$   $f_y$   $f_z$  2,3`
  - (10) `constraint plane cell  $x$   $y$   $z$  type  $id$`
  - (11) `constraint mindist_position  $x$   $y$   $z$`
- Required features: CONSTRAINTS <sup>1</sup>ELECTROSTATICS <sup>2</sup>ROTATION <sup>3</sup>DIPOLES

#### Description

The `constraint` command offers a variety of surfaces that can be defined to interact with desired particles. Variants (1) to (6) 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 (5), 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) 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.

Variants (7) and (8) 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 (9) does not define a surface but is based on magnetic dipolar interaction with an external magnetic field. It applies to all particles with a dipol moment.

Variant (10) is essential for the use of tunable-slip boundary interactions for microchannel flows like the Plane Poiseuille or Plane Couette Flow.

Variant (11) calculates the smallest distance to all non-penetrable constraints, that can be repulsive (wall, cylinder, sphere, rhomboid, maze, pore). Negative distances mean that the position is “within” the area that particles should not access. Helpful to find initial configurations.)

**Note that constraints are not saved to checkpoints and that they have to be reset upon restarting a simulation.**

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. 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, 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. 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.

Variant (7) specifies an electrostatic interaction between the charged particles in the system to an infinitely long rod with a line charge of  $lambda$  which is aligned along the z-axis and centered at  $c_x$  and  $c_y$ .

Variant (8) specifies the electrostatic interactions between the charged particles in the system and an infinitely large plate in the x-y-plane at height  $h$ . The plate carries a

charge density of *sigma*.

Variant (9) specifies the dipolar coupling of particles with a dipolar moment to an external field  $f_x f_y f_z$ .

Variant (10) 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.

#### *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

#### *Syntax*

```
| constraint delete [num]
```

#### *Description*

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

### 4.3.2. Getting the force on a constraint

#### *Syntax*

```
| constraint force n
```

#### *Description*

Returns the force acting on the  $n$ th constraint.

### 4.3.3. Getting the currently defined constraints

#### *Syntax*

```
| constraint [num]
```

#### *Description*

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

## 4.4. 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` in `myconfig.h` (sec. 3.4). 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 {molid {listofparticleids..} ...}
```

The lists of particles in a molecule comprise the non-virtual particles and the virtual site.

4. Assign to all particles that belong to the same molecule a common molecule id

```
| part pid mol molid
```

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

```
| integrate 0
```

#### 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

$$\vec{x}_v = \vec{x}_n + O_n(O_v \vec{E}_z)d, \quad (4.1)$$

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` in `myconfig.h` (see sec. 3.4). 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. The first two are saved in the virtual site's `vs_relative`-attribute, while the latter is saved in the quaternion 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 quat q vs_relative n d
```

 where `n` is the id of the non-virtual particle and `d` is its distance from the virtual site.
- 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.



#### 4.4.3. Additional features

The behaviour of virtual sites can be fine-tuned with the following switches in `myconfig.h` (sec.3.4)

- `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

## 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.

### 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

#### 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 5.8.5 for details.

### 5.1.1. Tabulated interaction

#### 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 tabforcecap**, see section 5.8.5.

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

#### 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

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

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

If  $c_{\text{shift}}$  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  $r_{\text{off}}$  is not set, it is set to 0.

The total force on a particle can be capped by using the command **inter ljforcecap**, see section 5.8.5, or on an individual level using the  $r_{\text{cap}}$  variable. When  $r_{\text{cap}}$  is set *and* **inter ljforcecap individual** has been issued before, the maximal force that is generated by this potential is the force at  $r_{\text{cap}}$ . 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  $r_{\text{min}}$ . 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  $r_{\text{cut}} = 2^{\frac{1}{6}}\sigma$ . The WCA potential is purely repulsive, and is often used to mimic hard sphere repulsion.

### 5.1.3. Generic Lennard-Jones interaction

#### Syntax

```
| 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 )]  
| 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

$$V_{\text{LJ}}(r) = \begin{cases} \epsilon(b_1(\frac{\sigma}{r-r_{\text{off}}})^{e_1} - b_2(\frac{\sigma}{r-r_{\text{off}}})^{e_2} + c_{\text{shift}}) & , \text{ if } r_{\text{min}} + r_{\text{off}} < r < r_{\text{cut}} + r_{\text{off}} \\ 0 & , \text{ otherwise} \end{cases} \quad (5.2)$$

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 ljforcecap**, see section 5.8.5, or on an individual level using the  $r_{\text{cap}}$  variable. When  $r_{\text{cap}}$  is set *and* **inter ljforcecap individual** has been issued before, the maximal force that is generated by this potential is the force at  $r_{\text{cap}}$ . By default, force capping is off, *i.e.* the cap radius is set to 0.

### 5.1.4. Lennard-Jones cosine interaction

#### Syntax

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

### Description

specifies a Lennard-Jones interaction with cosine tail [46] between particles of the types *type1* and *type2*. The first variant behaves as follows: Until the minimum of the Lennard-Jones potential at  $r_{\min} = r_{\text{off}} + 2^{\frac{1}{6}}\sigma$ , it behaves identical to the unshifted Lennard-Jones potential ( $c_{\text{shift}} = 0$ ). Between  $r_{\min}$  and  $r_{\text{cut}}$ , a cosine is used to smoothly connect the potential to 0, *i.e.*

$$V(r) = \frac{1}{2}\epsilon \left( \cos \left[ \alpha (r - r_{\text{off}})^2 + \beta \right] - 1 \right), \quad (5.3)$$

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

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

$$V(r) = \epsilon \cos^2 \left[ \frac{\pi}{2\omega} (r - r_{\min}) \right]. \quad (5.4)$$

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

Only the second variant allows capping the force using `inter ljforcecap`, see section 5.8.5.

### 5.1.5. Smooth step interaction

#### 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

$$V(r) = (\sigma_1/d)^n + \epsilon / (1 + \exp[2k_0(r - \sigma_2)]) \quad (5.5)$$

for  $r < r_{\text{cut}}$ , 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

#### 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:

$$V(r) = A \exp[B(\sigma - r)] - Cr^{-6} - Dr^{-8} + \epsilon_{\text{shift}}, \quad (5.6)$$

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)	$B$ ( $\text{\AA}^{-1}$ )	$C$ ( $\text{\AA}^6\text{kJ/mol}$ )	$D$ ( $\text{\AA}^8\text{kJ/mol}$ )	$\sigma$ ( $\text{\AA}$ )
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  $\approx 1389.3549 \text{ kJ/mol}$ .

### 5.1.7. Morse interaction

#### Syntax

```
| inter type1 type2 morse  $\epsilon$   $\alpha$   $r_{\text{min}}$   $r_{\text{cut}}$ 
| 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 morseforcecap`, see section 5.8.5.

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

$$V(r) = \epsilon (\exp[-2\alpha(r - r_{\text{min}})] - 2 \exp[-\alpha(r - r_{\text{min}})]) - \epsilon_{\text{shift}}, \quad (5.7)$$

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

#### 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

$$V(r) = A \exp(-Br) - Cr^{-6} - Dr^{-4} + \epsilon_{\text{shift}} \quad (5.8)$$

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 buckforcecap`, see section 5.8.5.

### 5.1.9. Soft-sphere interaction

#### Syntax

```
| inter type1 type2 soft-sphere a n r_cut r_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:

$$V(r) = a (r - r_{\text{offset}})^{-n} \quad (5.9)$$

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. Hertzian interaction

#### Syntax

```
| inter type1 type2 hertzian σ ε  
| 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

$$V(r) = \begin{cases} \epsilon \left(1 - \frac{r}{\sigma}\right)^{5/2} & r < \sigma \\ 0 & r \geq \sigma. \end{cases} \quad (5.10)$$

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

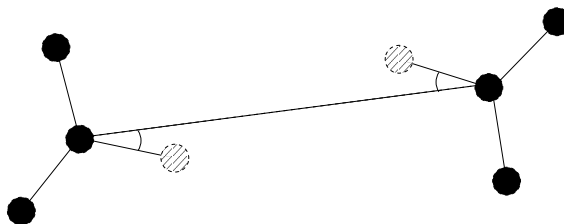
## 5.2. Anisotropic non-bonded interactions

### 5.2.1. Directional Lennard-Jones interaction

#### Syntax

```
| inter type1 type2 lj-angle ε σ r_cut b1_a b1_b b2_a b2_b [r_cap z0 δz κ ε']  
| 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

$$U(r_{ik}, \theta_{jik}, \theta_{ikn}) = \epsilon \left[ 5 \left( \frac{\sigma}{r} \right)^{12} - 6 \left( \frac{\sigma}{r} \right)^{10} \right] \cos^2 \theta_{jik} \cos^2 \theta_{ikn}, \quad (5.11)$$

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 seeked 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 ljangleforcecap`. 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

#### Syntax

```
| inter type1 type2 gay-berne  $\epsilon_0$   $\sigma_0$   $r_{\text{cutoff}}$   $k_1$   $k_2$   $\mu$   $\nu$ 
| 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 inter-molecular vector  $\mathbf{r} = r\hat{\mathbf{r}}$ . If  $r < r_{\text{cut}}$ , then the interaction between these two particles is given by

$$V(\mathbf{r}_{ij}, \hat{\mathbf{u}}_i, \hat{\mathbf{u}}_j) = 4\epsilon(\hat{\mathbf{r}}_{ij}, \hat{\mathbf{u}}_i, \hat{\mathbf{u}}_j) \left( \tilde{r}_{ij}^{-12} - \tilde{r}_{ij}^{-6} \right), \quad (5.12)$$

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

$$\tilde{r} = \frac{r - \sigma(\hat{\mathbf{r}}, \hat{\mathbf{u}}_i, \hat{\mathbf{u}}_j) + \sigma_0}{\sigma_0}, \quad (5.13)$$

$$\sigma(\hat{\mathbf{r}}, \hat{\mathbf{u}}_i, \hat{\mathbf{u}}_j) = \sigma_0 \left\{ 1 - \frac{1}{2}\chi \left[ \frac{(\hat{\mathbf{r}} \cdot \hat{\mathbf{u}}_i + \hat{\mathbf{r}} \cdot \hat{\mathbf{u}}_j)^2}{1 + \chi \hat{\mathbf{u}}_i \cdot \hat{\mathbf{u}}_j} + \frac{(\hat{\mathbf{r}} \cdot \hat{\mathbf{u}}_i - \hat{\mathbf{r}} \cdot \hat{\mathbf{u}}_j)^2}{1 - \chi \hat{\mathbf{u}}_i \cdot \hat{\mathbf{u}}_j} \right] \right\}^{-\frac{1}{2}} \quad (5.14)$$

and

$$\epsilon(\hat{\mathbf{r}}, \hat{\mathbf{u}}_i, \hat{\mathbf{u}}_j) = \epsilon_0 (1 - \chi^2 (\hat{\mathbf{u}}_i \cdot \hat{\mathbf{u}}_j))^{-\frac{\nu}{2}} \left[ 1 - \frac{\chi'}{2} \left( \frac{(\hat{\mathbf{r}} \cdot \hat{\mathbf{u}}_i + \hat{\mathbf{r}} \cdot \hat{\mathbf{u}}_j)^2}{1 + \chi' \hat{\mathbf{u}}_i \cdot \hat{\mathbf{u}}_j} + \frac{(\hat{\mathbf{r}} \cdot \hat{\mathbf{u}}_i - \hat{\mathbf{r}} \cdot \hat{\mathbf{u}}_j)^2}{1 - \chi' \hat{\mathbf{u}}_i \cdot \hat{\mathbf{u}}_j} \right) \right]^\mu. \quad (5.15)$$

The parameters  $\chi = (k_1^2 - 1) / (k_1^2 + 1)$  and  $\chi' = (k_2^{1/\mu} - 1) / (k_2^{1/\mu} + 1)$  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  $\nu = 1$ .

## 5.3. Bonded interactions

### Syntax

```
| inter bondid [interaction] [parameters]
```

### Description

Bonded interactions are identified by their *bonded interaction type identifier* *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 26). 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.

### 5.3.1. FENE bond

#### Syntax

```
| inter bondid fene K  $\Delta r_{\max}$  [r0]
```

#### Description

This creates a bond type with identifier *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

$$V(r) = -\frac{1}{2}K\Delta r_{\max}^2 \ln \left[ 1 - \left( \frac{r - r_0}{\Delta r_{\max}} \right)^2 \right]. \quad (5.16)$$

### 5.3.2. Harmonic bond

#### Syntax

```
| inter bondid harmonic K R [rcut]
```

#### Description

This creates a bond type with identifier *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

$$V(r) = \frac{1}{2}K (r - R)^2 \quad (5.17)$$

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

### 5.3.3. Subtracted Lennard-Jones bond

#### Syntax

```
| inter bondid subt_lj reserved R
```

#### Description

This creates a “bond” type with identifier *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 is often the case for force fields or in general tabulated potentials.

### 5.3.4. Rigid bonds

#### Syntax

```
| inter bondid rigid_bond constrained_bond_distance positional_tolerance  
      velocity_tolerance
```

#### 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]

### 5.3.5. Tabulated bond interactions

#### 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 identifier *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 5.4). 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 5.4) 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 5.5). 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.*

### 5.3.6. Virtual bonds

#### Syntax

```
| inter bondid virtual_bond
```

#### Description

This creates a virtual bond type with identifier *bondid*, *i.e.* a pair bond without associated potential or force. It can be 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. Bond-angle interactions

### Syntax

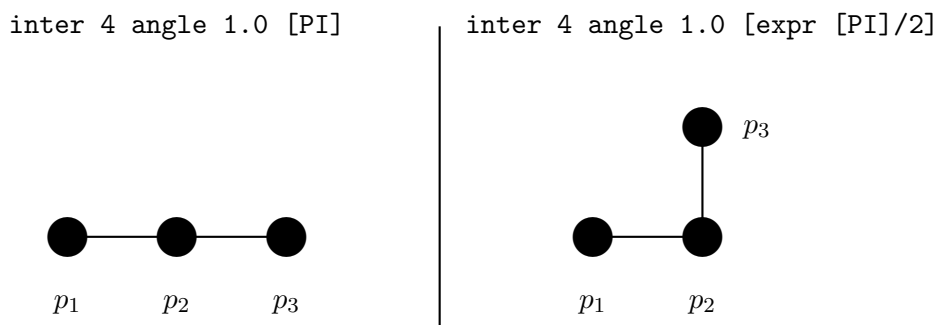
```
inter bondid angle K [ $\phi_0$ ]
Required features: BOND_ANGLE_HARMONIC , BOND_ANGLE_COSINE or BOND_ANGLE_
COSSQUARE
```

### Description

This creates a bond type with identifier *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 equilibrium 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, different choices are possible, which have to be activated in `myconfig.h`

- Harmonic bond angle potential (requires feature BOND\_ANGLE\_HARMONIC):  
A classical harmonic potential,

$$V(\phi) = \frac{K}{2} (\phi - \phi_0)^2. \quad (5.18)$$

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 (requires feature BOND\_ANGLE\_COSINE):

$$V(\alpha) = K [1 - \cos(\phi - \phi_0)] \quad (5.19)$$

Around  $\phi_0$ , this potential 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 (requires feature BOND\_ANGLE\_COSSQUARE):

$$V(\alpha) = \frac{K}{2} [\cos(\phi) - \cos(\phi_0)]^2 \quad (5.20)$$

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.

## 5.5. Dihedral interactions

### Syntax

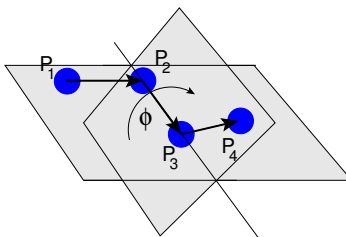
```
|inter bondid dihedral n K p
```

### Description

This creates a bond type with identifier *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 p2 bond bondid p1 p3 p4`. Then, the dihedral potential is given by

$$V(\phi) = K [1 - \cos(n\phi - p)], \quad (5.21)$$

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:

$$V(\phi) = K [1 + p \cos(n\phi)], \quad (5.22)$$

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

## 5.6. Coulomb interaction

### 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

$$U^C(r) = l_B k_B T \frac{q_1 q_2}{r}. \quad (5.23)$$

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
```

### 5.6.1. Coulomb P3M

#### Syntax

```
inter coulomb l_B p3m r_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. The different parameters are described in more detail in [15].

*r<sub>cut</sub>* The real space cutoff as a positive floating point number.

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

*mesh<sub>x,y,z</sub>* 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 [18, 22, 28, 14, 15, 16, 13, 12].

## Tuning Coulomb P3M

### Syntax

```
inter coulomb lB p3m ( tune | tunev2 ) accuracy accuracy
      [rcut rcut] [mesh mesh] [cao cao] [alpha α]
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 [22] and its real space error [28] 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

### 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<sub>i</sub>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<sub>off</sub>* 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.

## 5.6.2. Debye-Hückel potential

### Syntax

```
| inter coulomb l_B dh kappa r_cut
| Required features: ELECTROSTATICS
```

### Description

Defines the electrostatic potential by

$$U^{C-DH} = l_B k_B T \frac{q_1 q_2 \exp(-\kappa r)}{r} \quad \text{for } r < r_{\text{cut}} \quad (5.24)$$

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  $r > r_{\text{cut}}$  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.

### 5.6.3. MMM2D

#### Syntax

```
| inter coulomb  $l_B$  mmm2d maximal_pairwise_error [fixed_far_cutoff]
| [dielectric  $\epsilon_t$   $\epsilon_m$   $\epsilon_b$ ] [dielectric-contrasts  $\Delta_t$   $\Delta_b$ ]
| Required features: ELECTROSTATICS
```

#### Description

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 195.

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.

### 5.6.4. MMM1D

#### Syntax

```
| (1) inter coulomb  $l_B$  mmm1d switch_radius [bessel_cutoff]
| maximal_pairwise_error
| (2) inter coulomb  $l_B$  mmm1d tune maximal_pairwise_error
| Required features: ELECTROSTATICS
```

#### Description

MMM1D coulomb method for systems with periodicity 0 0 1. Needs the nsquared cell system (see section 6.4 on page 74). 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. If this takes too long, you can change the value of the setmd variable `timings`, which controls the number of test force calculations. For details on the MMM family of algorithms, refer to appendix E on page 195.

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

#### Syntax

```
| inter coulomb l_B memd f_mass mesh [epsilon ε∞]  
| Required features:  ELECTROSTATICS
```

#### Description

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

$$U = l_B k_B T \frac{q_1 q_2}{r} \quad (5.25)$$

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.
- `ε∞` 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:  $\text{mesh} \approx \text{box\_l}/\text{lj\_sigma}$
- 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  $\text{f\_mass} \gg dt^2/a^2$ .

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 algorithmic error is proportional to  $1/c^2$ , where  $c$  is the adjustable speed of light. From the stability criterion, this yields

$$\Delta_{\text{maggs}} = A \cdot a^3 + B \cdot dt^2/a^2 \quad (5.26)$$

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 189 or the publications [32, 36].

### 5.6.6. Electrostatic Layer Correction (ELC)

#### Syntax

```
inter coulomb elc maximal_pairwise_error gap_size
    [far_cutoff] [noneutralization] [dielectric  $\epsilon_t$   $\epsilon_m$   $\epsilon_b$ ]
    [dielectric-contrasts  $\Delta_t$   $\Delta_b$ ]
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 195.

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 [9]. 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 [9]. Also, if you add neutralizing walls explicitly as constraints, you have to disable the neutralization.

The dielectric contrast features work exactly the same as for MMM2D, see the documentation above.

Make sure that you read the papers on ELC ([6, 53]) before using it.

### 5.6.7. Dielectric interfaces with the ICC★ algorithm

#### 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 [8, 51, 27] before using it.

The particles with ids 0 to  $n\_induced\_particles - 1$  are treated as iterated particles by ICC★. They 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 element's 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

### 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`

### 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 geometry is supported. 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_epsilon`s 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_epsilon normals  
$icc_normals areas $icc_areas sigmas $icc_sigmas
```

## 5.7. Dipolar interaction

### 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:

$$U^{D-P3M}(\vec{r}) = l_B k_B T \left( \frac{(\vec{\mu}_i \cdot \vec{\mu}_j)}{r^3} - \frac{3(\vec{\mu}_i \cdot \vec{r})(\vec{\mu}_j \cdot \vec{r})}{r^5} \right) \quad (5.27)$$

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. ESPReso 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 counterparts for the electrostatic interactions (see section 5.6.1 on page 55). 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
```

### 5.7.1. Dipolar P3M

#### 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 [18, 22, 28, 14, 15, 16, 13].

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

### Tuning dipolar P3M

#### 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 algorithm, refer to the documentation of Coulomb P3M (see section 5.6.1 on page 56).

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

### 5.7.2. Dipolar Layer Correction (DLC)

#### 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 large as the largest of the dipoles in the system, the error for the energy would be smaller than 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].

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

#### 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.

### 5.7.4. Magnetic Dipolar Direct Sum (MDDS)

#### 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. Special interaction commands

### 5.8.1. Tunable-slip boundary interaction

#### 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** plane cell 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 [44] before using this interaction.

### 5.8.2. DPD interaction

#### 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 6.2.2 when the INTER\_DPD implementation is used. The parameters correspond to the parameters of the DPD thermostat 5.8.2, but can be set individually for the different interactions.

### 5.8.3. Fixing the center of mass

#### 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 on the particles. This force is divided equally on all the particles of type *typeid1*, since currently there is no mass concept in ESPResSo. 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).

### 5.8.4. Pulling particles apart

#### 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<sub>flag</sub>* 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.

### 5.8.5. Capping the force during warmup

#### Syntax

```
(1) inter ljforcecap (  $F_{\max}$  | individual )
(2) inter morseforcecap (  $F_{\max}$  | individual )
(3) inter buckforcecap (  $F_{\max}$  | individual )
(4) inter tabforcecap (  $F_{\max}$  | individual )
Required features: (1) Lennard-Jones (2) Morse (3) Buckingham (4) Tabulated
```

#### 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 interactions of the corresponding type (*e.g.* all Lennard-Jones 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.

#### 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:

`box_l` (double[3]) Simulation box length.

`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.

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

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

`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 74).

**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*)

**nptiso\_gammav** (double, *read-only*)

**npt\_p\_ext** (double, *read-only*) Pressure for NPT simulations.

**npt\_p\_inst** (double) Pressure calculated during an NPT\_isotropic integration.

**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, this variable can be set to (1,1,1) or (0,0,0) at the moment. If not it is readonly and gives the default setting (1,1,1).

**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.

## 6.2. thermostat: Setting up the thermostat

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.

### Syntax

- | (1) **thermostat**
- | (2) **thermostat off**
- | (3) **thermostat parameters**

### Description

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 5.6).

Variant (3) sets up one of the thermostats described below.

#### 6.2.1. Langevin thermostat

### Syntax

- | **thermostat langevin temperature gamma**

### Description

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. For a more detailed explanation, refer to [10, 20].

If the feature **ROTATION** is compiled in, the rotational degrees of freedom are also coupled to the thermostat.

Using the Langevin thermostat, it is possible to set a temperature and a friction coefficient for every particle individually. Consult the **part** commands reference (chapter 4) for information on how to achieve this.

### 6.2.2. 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

##### 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 [47]. 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 [35]. 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

$$\vec{F}_{ij}^D = -\zeta w^D(r_{ij})(\hat{r}_{ij} \cdot \vec{v}_{ij})\hat{r}_{ij}$$

The random force by

$$\vec{F}_{ij}^R = \sigma w^R(r_{ij})\Theta_{ij}\hat{r}_{ij}$$

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:

$$(\sigma w^R(r_{ij}))^2 = \zeta w^D(r_{ij})k_B T$$

The parameters *gamma* *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 (  $r_{ij} < r_{cut}$  ) :

$$w^D(r_{ij}) = (w^R(r_{ij}))^2 = \begin{cases} (1 - \frac{r_{ij}}{r_c})^2 & , WF = 0 \\ 1 & , WF = 1 \end{cases}$$

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 probably 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 `galileiTransformParticles` 11.1. This may be repeated once in a while for long runs due to round off errors (check this with the command `system_com_vel`) 11.1.

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 translational degrees of freedom.

**Transverse DPD thermostat** This is an extension of the above standard DPD thermostat [26], 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

$$\vec{F}_{ij}^D = -\zeta w^D(r_{ij})(I - \hat{r}_{ij} \otimes \hat{r}_{ij}) \cdot \vec{v}_{ij}$$

The random force by

$$\vec{F}_{ij}^R = \sigma w^R(r_{ij})(I - \hat{r}_{ij} \otimes \hat{r}_{ij}) \cdot \vec{\Theta}_{ij}$$

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

### 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 5.8.2 on page 66). This allows to set the friction on a per-interaction basis.

### Other DPD extensions

The features `DPD_MASS_RED` or `DPD_MASS_LIN` make the friction constant mass dependent:

$$\zeta \rightarrow \zeta M_{ij}$$

and

$$\zeta \rightarrow \zeta M_{ij}$$

There are two implemented cases. `DPD_MASS_RED` uses the reduced mass:

$$M_{ij} = 2 \frac{m_i m_j}{m_i + m_j}$$

while `DPD_MASS_LIN` uses the real mass mass:

$$M_{ij} = \frac{m_i + m_j}{2}$$

The prefactors are such that equal masses result in a factor 1.

### 6.2.3. Isotropic NPT thermostat

#### Syntax

```
thermostat npt_isotropic temperature gamma0 gammaV
Required features: NPT
```

#### Description

This theormstat is based on the Anderson thermostat (see [1, 33]) 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.

## 6.3. nemd: Setting up non-equilibrium MD

#### 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 [46].

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

$$target\_velocity = \pm shearrate \frac{L_z}{4} \quad (6.1)$$

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

$$\eta = \frac{F}{\dot{\gamma} L_x L_y} \quad (6.2)$$

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.

## 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 14.1 on page 167.

### 6.4.1. Domain decomposition

#### 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 cell system 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

### Syntax

```
| cellsystem nsquare
```

### Description

This selects the very primitive nsquared cell system, 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 cell system 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

#### 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  $N \cdot n\_layers$  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. AdResS

#### Syntax

```
| address set topo kind width width hybrid_width center x R_x wf wf  
| Required features:  ADRESSO
```

#### Description

where *kind* determines the type of AdResS simulation:

- 0 disabled
- 1 constant weight function
- 2 one dimensional splitting
- 3 spherical splitting

*wf* the type of weighting function:

- 0 standard
- 1 user defined

*width* and *hybrid\_width* are the widths of the explicit and hybrid regions respectively and *R\_x* is the *x* position of the center of the explicit zone. For more details on the method itself see [38, 39, 37]. And for further information about the technical implementation see [25].

## 6.6. CUDA

#### 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.

## 6.7. Creating bonds when particles collide

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: In the “bind centers” method, a bonded interaction is added between the colliding particles. This leads to the distance between the particles being fixed, the particles can, however still slide around each other. The second method, called “bind at point of collision”, aims to prevent this sliding 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 5.4). 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 5.3.4.

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

### Syntax

- (1) `collision_detection`
- (1) `collision_detection off`
- (2) `collision_detection bind_centers d bond1`
- (3) `collision_detection bind_at_point_of_collision d bond1 bond2 type`

### Description

Here, *d* is the distance at which the bond is created, *bond1* denotes the type of the bond created between the colliding particles, and *bond2* determines the type of the bond created between the virtual sites (if applicable). *bond1* needs to denote a pair bond, like FENE or harmonic, *bond2* 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 Lennard-Jones interaction for this particle type, as two particles will be generated in the same place.

## 7. Running the simulation

### 7.1. `integrate`: Running the simulation

#### Syntax

```
(1) integrate steps  
(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.

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. `change_volume`: Changing the box volume

#### Syntax

```
(1) change_volume  $V_{new}$   
(2) change_volume  $L_{new}$  ( x | y | z | xyz )
```

#### Description

Changes the volume of either a cubic simulation box to the new volume  $V_{new}$  or its given x-/y-/z-/xyz-extension to the new box-length  $L_{new}$ , and isotropically adjusts the particles coordinates as well. The function returns the new volume of the deformed simulation box.

## 7.3. Stopping particles

### Syntax

```
| (1) stopParticles  
| (2) stop_particles
```

### Description

Halts all particles in the current simulation, setting their velocities and forces to zero. Variant (2) does not provide feedback on the execution status.

## 7.4. velocities: Setting the velocities

### 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.

## 7.5. invalidate\_system

### Syntax

```
| invalidate_system
```

### Description

Forces a system re-init which, among others, causes the integrator to also update the forces at its beginning (instead of re-using the values from the previous integration step). This is particularly necessary to ensure continuity after setting a checkpoint: **integrate - set\_checkpoint - integrate** has only one call to the force calculation routine, while **read\_checkpoint - integrate** has two at the beginning of the **integrate** command (because loading a new system from disk typically requires re-initializing the system), and since the forces routine also uses the thermostat which in turn draws random numbers, the two situations do not end up at the same segment of the random number sequence, all random events will therefore slightly differ. To prevent this, simply include a call to **invalidate\_system** upon setting the checkpoint, because in that case both scenarios will call the forces routine twice at the beginning of the second integration phase thus having their random number sequences in total sync.

Without applying this command directly before or after writing a checkpoint, you will run into a different state of the random number generator when reading the checkpoint to start again later!



## 7.6. Parallel tempering

### Syntax

```
parallel_tempering::main -rounds N -swap swap -perform perform  
    [-init init] [-values {Ti}] [-connect master] [-port port]  
    [-load jnode] [-resrate Nreset] [-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 initial 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 alternately, 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<sub>node</sub>* 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<sub>reset</sub>* 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  $T_1 < T_2 < \dots < T_N$ , 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

$$\min(1, \exp - [\beta(T_2)U_A(T_2) + \beta(T_1)U_B(T_1) - \beta(T_1)U_A(T_1) - \beta(T_2)U_B(T_2)]), \quad (7.1)$$

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 independent of  $T$ , and  $\beta(T) = 1/(k_B T)$ . In this case, the swap probability reduces to the textbook result

$$\min(1, \exp - [(1/T_2 - 1/T_1)(U_A - U_B)/k_B]). \quad (7.2)$$

However,  $T$  can also be chosen to be any other parameter, for example the Bjerrum length, *i.e.* 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 (7.1). 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.

The 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 explicitly 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 time 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 probability 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. (7.1).

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

### 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`.

## 7.7. Metadynamics

### Syntax

```
(1) metadynamics
(2) metadynamics set off
(3) metadynamics set distance pid1 pid2 dmin dmax bheight bwidth fbound
    dbins
(4) metadynamics set relative_z pid1 pid2 zmin zmax bheight bwidth fbound
    zbins
(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*<sub>1</sub> and *pid*<sub>2</sub>). As the system samples values along this reaction coordinate (here the distance between *pid*<sub>1</sub> and *pid*<sub>2</sub>), 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*<sub>min</sub> and *d*<sub>max</sub>). The potential of mean force (or free energy profile) can be extracted by reading the **profile**.

### Arguments

- *pid*<sub>1</sub> ID of the first particle involved in the metadynamics scheme.
- *pid*<sub>2</sub> ID of the second particle involved in the metadynamics scheme.
- *d*<sub>min</sub>, *z*<sub>min</sub> : minimum value of the reaction coordinate. While *d*<sub>min</sub> must be positive (it's a distance), *z*<sub>min</sub> can be negative since it's the relative height of *pid*<sub>1</sub> with respect to *pid*<sub>2</sub>.
- *d*<sub>max</sub>, *z*<sub>max</sub> : maximum value of the reaction coordinate.
- *b*<sub>height</sub> height of the bias function.
- *b*<sub>width</sub> width of the bias function.
- *f*<sub>bound</sub> strength of the ramping force at the boundaries of the reaction coordinate interval.
- *d*<sub>bins</sub>, *z*<sub>bins</sub> : number of bins of the reaction coordinate.
- *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 *pid<sub>1</sub>* with respect to *pid<sub>2</sub>* (*e.g.* calculate the potential of mean force of inserting one particle *pid<sub>1</sub>* through an interface with center of mass *pid<sub>2</sub>*). 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.*  $d_{\text{bins}}$  values between  $d_{\text{min}}$  and  $d_{\text{max}}$ ). 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 [34] for more details. These avoid the calculation of exponentials.

## 8. Analysis in Tcl

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 they 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

##### 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

#### Syntax

```
| (1) analyze nbhood pid r_catch  
| (2) analyze nbhood x y z r_catch
```

#### 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

#### 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 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* ≥ 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 10.1 on page 115):

```
{ parameters }  
{  
  { r dist(r) }  
  :  
}
```

### 8.1.4. Radial density map

#### 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.

### 8.1.5. Modes

*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:

$$ht_{RE} \quad ht_{IM} \quad \theta_{RE} \quad \theta_{IM}$$

### 8.1.6. Lipid orientation

*Syntax*

```
| (1) analyze get_lipid_orients
| (2) analyze lipid_orient_order
```

*Description*

### 8.1.7. Bilayers

*Syntax*

```
| (1) analyze bilayer_set
| (2) analyze bilayer_density_profile
```

*Description*

### 8.1.8. GPB

*Syntax*

```
| analyze cell_gpb Manningparameter outercellradius innercellradius
| [accuracy [numberofinteractions]]
```

*Description*

### 8.1.9. Get folded positions

*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.

## 8.1.10. Vkappa

### Syntax

```
|analyze Vkappa [( reset | read | set  $V_{\kappa,1}$   $V_{\kappa,2}$   $avk$  ) ]
```

### Description

## 8.1.11. Radial distribution function

### 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 10.1 on page 115):

```
{ parameters }  
{  
  { r rdf(r) }  
  :  
}
```

## 8.1.12. Structure factor

### Syntax

```
|analyze structurefactor type order
```

### Description

Returns the spherically averaged structure factor  $S(q)$  for particles of a given type *type*. The  $S(q)$  is calculated for all possible wave vectors,  $\frac{2\pi}{L} \leq q \leq \frac{2\pi}{L} order$ . Do not chose parameter *order* too large, because the number of calculations grows as  $order^3$ .

### Output format

The output corresponds to the blockfile format (see section 10.1 on page 115):

```
{ q_value S(q)-value }  
:  
:
```

### 8.1.13. Van-Hove autocorrelation function $G(r, t)$

#### 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 101) 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 10.1 on page 115):

```
{ msd { msd(0) msd(1) ... } }
{ vanhove { { G(0,0) G(1,0) ... }
            { G(0,1) G(1,1) ... }
          }
:
      }
}
```

The  $G(r, t)$  are normalized such that the integral over space always yields 1.

### 8.1.14. Center of mass

#### Syntax

```
| analyze centermass part_type
```

#### Description

Returns the center of mass of particles of the given type.

### 8.1.15. Moment of inertia matrix

#### 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.

### 8.1.16. Gyration tensor

#### 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.

### 8.1.17. Aggregation

#### 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.

### 8.1.18. Identifying pearl-necklace structures

#### Syntax

```
| analyze necklace pearl_threshold back_dist space_dist first length
```

#### Description

Algorithm for identifying pearl necklace structures for polyelectrolytes in poor solvent [30]. 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.

### 8.1.19. Finding holes

#### Syntax

```
| analyze holes typeidprobe mesh_size
```

#### 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 [43]. 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. Requires the feature `LENNARD_JONES`. .

### 8.1.20. Energies

#### Syntax

- (1) `analyze energy`
- (2) `analyze energy ( total | kinetic | coulomb | magnetic )`
- (3) `analyze energy bonded bondid`
- (4) `analyze energy nonbonded typeid1 typeid2`

#### Description

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* } ...

### 8.1.21. Pressure

#### Syntax

- (1) `analyze pressure`
- (2) `analyze pressure total`
- (3) `analyze pressure ( totals | ideal | coulomb |  
tot_nonbonded_inter | tot_nonbonded_intra )`
- (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!

The pressure is calculated (if there are no electrostatic interactions) by

$$p = \frac{2E_{kinetic}}{Vf} + \frac{\sum_{j>i} F_{ij}r_{ij}}{3V} \quad (8.1)$$

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

$$\frac{2E_{kinetic}}{f} = \frac{1}{3} \sum_i m_i v_i^2. \quad (8.2)$$

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. [50]. 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. 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.

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.

### 8.1.22. Stress Tensor

*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 8.1.21 on page 94. 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 8.1.21).

The stress tensor is calculated by

$$p^{(kl)} = \frac{\sum_i m_i v_i^{(k)} v_i^{(l)}}{V} + \frac{\sum_{j>i} F_{ij}^{(k)} r_{ij}^{(l)}}{V} \quad (8.3)$$

where the notation is the same as for **analyze pressure** in 8.1.21 on page 94 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.

## 8.1.23. Local Stress Tensor

### 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 is 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)

The following 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.

*Syntax*

```
(1) analyze set chains [chain_start n_chains chain_length]
(2) analyze set topo_part_sync
(3) analyze set
```

*Description*

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

##### 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 101).

##### Output format

```
{ re error_of_re re2 error_of_re2 }
```

#### Radius of gyration

##### 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

$$R_G^2 = \frac{1}{N} \sum_{i=1}^N (\vec{r}_i - \vec{r}_{\text{cm}})^2, \quad (8.4)$$

where  $\vec{r}_i$  are position vectors of individual particles constituting a molecule and  $\vec{r}_{\text{cm}}$  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. [41]. If <rg> is used, the radius of gyration is averaged over all stored configurations (see section 8.3 on page 101).

##### Output format

```
{ rg error_of_rg rg2 error_of_rg2 }
```

#### Hydrodynamic radius

##### 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 101). The following formula is used for the computation:

$$\frac{1}{R_H} = \frac{2}{N^2} \sum_{i=1}^N \sum_{j=i}^N \frac{1}{|\vec{r}_i - \vec{r}_j|}, \quad (8.5)$$

The above-mentioned formula is only valid under certain assumptions. For more information, see Chapter 4 and equation 4.102 in [17].

### Output format

{ *rh error\_of\_rh* }

## Internal distances

### 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 101).

### 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)

### 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 101).

### Output format

{ *bdf(0) bdf(1) ... bdf(chain\_length - 1 - index)* }

## Bond lengths

### Syntax

```
| analyze ( bond_1 | <bond_1> ) [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_1>` is used, the value will be averaged over all stored configurations (see section 8.3 on the next page). This function assumes linear chain topology and does not check if the bonds really exist!

### Output format

```
{ mean stddev max min }
```

## Form factor

### Syntax

```
| analyze ( formfactor | <formfactor> ) qmin qmax qbins  
[chain_start n_chains chain_length]
```

### Description

Computes the spherically averaged form factor of a single chain, which is defined by

$$S(q) = \frac{1}{chain\_length} \sum_{i,j=1}^{chain\_length} \frac{\sin(qr_{ij})}{qr_{ij}} \quad (8.6)$$

of a single chain, averaged over all chains for `qbin + 1` logarithmically spaced q-vectors `qmin, ..., 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 facing page).

### Output format

```
{  
  { q S(q) }  
  :  
}
```

with  $q \in \{qmin, \dots, qmax\}$ .

## Chain radial distribution function

### 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

### 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 in 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).

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 10.1 on page 115).

### 8.3.1. Storing and removing configurations

#### 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

#### 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

### 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 [54].

### Arguments

- *data* is a matrix filled with the primary estimates  $a_{\alpha}^{i,r}$  from  $R$  replica with  $N_1, N_2, \dots, N_R$  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.

$$nrep = (N_1, N_2, \dots, N_R)$$

- *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 [54]. 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  $\tau_{\text{int}}$  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 too 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 `telcommand` 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. It does not refer to the numbers itself.

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.

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

#### 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 9.1.6 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.

- **particle\_positions** *particle\_specifications*  
Positions of the particles, in the format  $x_1, y_1, z_1, x_2, y_2, z_2, \dots, x_n, y_n, z_n$ . The particles are ordered ascending according to their ids.
- **particle\_velocities** *particle\_specifications*  
Velocities of the particles, in the format  $v_1^x, v_1^y, v_1^z, v_2^x, v_2^y, v_2^z, \dots, v_n^x, v_n^y, v_n^z$ . The particles are ordered ascending according to their ids.
- **particle\_forces** *particle\_specifications*  
Forces on the particles, in the format  $f_1^x, f_1^y, f_1^z, f_2^x, f_2^y, f_2^z, \dots, f_n^x, f_n^y, f_n^z$ . The particles are ordered ascending according to their ids.
- **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.
- **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
- **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\_specifications1*
- **density\_profile** *particle\_specifications profile\_specifications*  
Compute the density profile within the specified cube. For profile specifications, see section 9.1.7.
- **lb\_velocity\_profile** *particle\_specifications profile\_specifications*  
Compute the Lattice-Boltzmann velocity profile within the specified cube. For profile specifications, see section 9.1.7.
- **flux\_density\_profile** *particle\_specifications profile\_specifications*  
Compute the flux density within the specified cube. For profile specifications, see section 9.1.7.
- **radial\_density\_profile** Compute the density profile in cylindrical coordinates. For profile specifications, see section 9.1.7.
- **radial\_flux\_density\_profile**  
Compute the flux density profile in cylindrical coordinates. For profile specifications, see section 9.1.7.

- **lb\_radial\_velocity\_profile**  
Compute the Lattice-Boltzmann velocity profile in cylindrical coordinates. For profile specifications, see section 9.1.7.
- **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.

### 9.1.3. Printing an observable

#### Syntax

```
| observable id print [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.

### 9.1.4. Passing an observable to an analysis function

Currently the only analysis function which uses the core observables is the correlator (section 9.2).

### 9.1.5. Deleting an observable to an analysis function

#### Syntax

```
| observable id delete
```

#### Description

Deletes the observable, *i.e.* frees the allocated memory and makes the *id* free for a new observable.

### 9.1.6. 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 tel list of existing particle ids.

### 9.1.7. 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:

#### 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:

#### Syntax

```
observable new needs_radial_profile_specs [other_parameters]
[ center <cx> <cy> <cz> ] [ 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

$$C(\tau) = \langle A(t) \otimes B(t + \tau) \rangle , \quad (9.1)$$

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  $C(\tau)$ .

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.

#### 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 be 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 available:

- **scalar\_product**  
Scalar product of  $A$  and  $B$ , *i.e.*  $C = \sum_i A_i B_i$
- **componentwise\_product**  
Componentwise 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).
- **complex\_conjugate\_product**
- **fcs\_acf**  $w_x w_y w_z$   
Fluorescence Correlation Spectroscopy (FCS) autocorrelation function, *i.e.*

$$G_i(\tau) = \frac{1}{N} \left\langle \exp \left( -\frac{\Delta x_i^2(\tau)}{w_x^2} - \frac{\Delta y_i^2(\tau)}{w_y^2} - \frac{\Delta z_i^2(\tau)}{w_z^2} \right) \right\rangle, \quad (9.2)$$

where  $\Delta x_i^2(\tau) = (x_i(0) - x_i(\tau))^2$  is the square displacement of particle  $i$  in the  $x$  direction, and  $w_x$  is the beam waist of the intensity profile of the exciting laser beam,

$$W(x, y, z) = I_0 \exp \left( -\frac{2x^2}{w_x^2} - \frac{2y^2}{w_y^2} - \frac{2z^2}{w_z^2} \right). \quad (9.3)$$

Equation 9.2 is a generalization of the formula presented by Höfling *et al.* [23]. 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*  $\neq$  *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. [40] 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. [40] or to perform your own tests.

### 9.2.3. Inquiring about already existing correlations

#### Syntax

```
| (1) correlation
| (2) correlation n_corr
```

#### Description

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

#### 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.

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

#### 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 [54] (same as used by the **uerr** 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 [42]. About a decade later it found its way to the Fluorescence Correlation Spectroscopy (FCS) [31]. The book of Frenkel and Smit [19] describes its application for the special case of the velocity autocorrelation function.

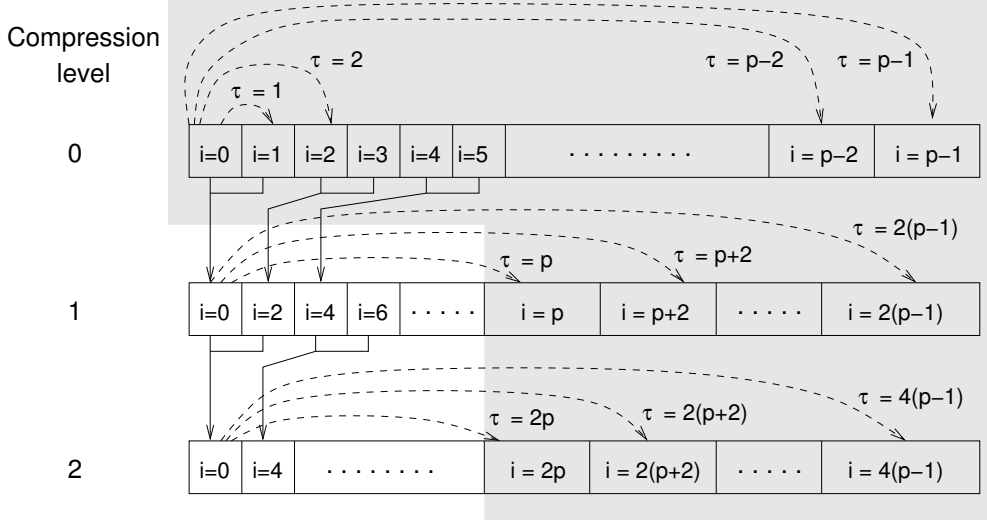


Figure 9.1.: Schematic representation of buffers in the correlator.

Let us consider a set of  $N$  observable values as schematically shown in Figures 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 ?? 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  $m^2$ . 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  $\mathcal{O}(p^2 \log(\tau_{\max}))$ . 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. [40].

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

$$\frac{1}{2}(A_i \otimes B_{i+p} + A_{i+1} \otimes B_{i+p+1}) - \frac{1}{2}(A_i + A_{i+1}) \otimes \frac{1}{2}(B_{i+p} + B_{i+p+1}) \quad (9.4)$$

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 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. [40].

# 10. Input / Output

## 10.1. 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 ??).

### 10.1.1. Writing ESPResSo's global variables

#### 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 68) 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.

### 10.1.2. Writing Tcl variables

#### Syntax

```
(1) blockfile channel write tclvariable { varname1 varname2 ... }  
(2) blockfile channel write tclvariable all  
(2) 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.

### 10.1.3. Writing particles, bonds and interactions

#### 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 parti 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. The keyword `all` denotes all known particles.

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*.

### 10.1.4. Writing the random number generator states

#### Syntax

```
(1) blockfile channel write random  
(2) blockfile channel write bit_random  
(3) blockfile channel write seed  
(4) blockfile channel write bitseed
```

### *Description*

Variants (1) and (2) write the full information on the current states of the respective random number generators (see sections 11.3.1 on page 134 and 11.3.2 on page 135) on any node to *channel*. Using this information, it is possible to recover the exact states of the generators.

Variants (3) and (4) 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.

## **10.1.5. Writing all stored configurations**

### *Syntax*

```
| blockfile channel write configs
```

### *Description*

This command writes all configurations currently stored for off-line analysis (see section 8.3 on page 101) to *channel*.

## **10.1.6. Writing arbitrary blocks**

### *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`.

### 10.1.7. Reading blocks

#### Syntax

```
(1) blockfile channel read start
(2) blockfile channel read toend
(3) blockfile channel read ( particles | interactions | bonds |
    variable | seed | random | bitrandom | configs )
(4) blockfile channel read auto
```

#### 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 the beginning of one block, checks whether it contains data of the given type and reads it. Variant (4) 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 (3) 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.

## 10.2. Checkpointing (deprecated)

**Warning:** The functions in this section are deprecated and will be removed in some future version. They might still be usable in your case, but be aware that if you use newer features in your simulation, not all necessary parameters might be saved and restored when writing a standard checkpoint. Try to use blockfiles instead and think about what variables need to be stored.

The following procedures may be used to save and restore checkpoints to minimize the hassle involved when your simulation crashes after long runs.

### 10.2.1. Creating a checkpoint (deprecated)

#### Syntax

```
| checkpoint_set destination [num_configs [tclvar [iaflag [varflag [ranflag]]]]]
```

#### Description

Creates a checkpoint with path/filename *destination* (compressed if *destination* ends with '.gz'), saving the last *#ofconfigs* which have been appended using `analyze_append` (defaults to 'all'), adds all tcl-embedded variables specified in the tcl-list *tclvar* (defaults

to '-'), all interactions (The `inter` command) / ESPResSo-variables (The `setmd` command) / random-number-generator informations (The `t_random` command etc.) unless their respective flags *iaflag* / *varflag* / *ranflag* are set to '-'; you may however choose to only include certain ESPResSo-variables (The `setmd` command) by providing their names as a tcl-list in place of *varflag*. When you're reading this, `tcl_checkpoint_set` will be using the `invalidate.system` command automatically; therefore continuing an integration after setting a checkpoint or restarting it there by reading one should make absolutely no difference anymore, since the current state of the random number generator(s) is/are completely (re)stored to (from) the checkpoint and the integrator is forced to re-init the forces (incl. thermostat) no matter what. It may be a good choice to use filenames such as `'kremer_checkpoint.[eval format 05 $integration_step]'` or `'kremer_checkpoint.029.gz'` for *destination* because the command stores all the names of checkpoints set to a file derived from *destination* by replacing the very last suffix plus maybe '.gz' with '.chk' (in the above examples: `'kremer_checkpoint.chk'`) which is used by `tcl_checkpoint_read` to restore all checkpoints. Although `'checkpoint_set destination'` without the optional parameters will store a complete checkpoint sufficient for re-starting the simulation later on, you may run out of memory while trying to save a huge number of timesteps appended (`analyze_append`). Hence one should rather only save those configurations newly added since the last checkpoint, i.e. if a checkpoint is created every 100,000 steps while a configuration is appended every 500 steps you may want to use `'checkpoint_set destination 200'` which saves the current configuration, all interactions, all bonds, the precise state of the random number generator(s), and the last 200 entries appended to configs since the last checkpoint was created. Since `tcl_checkpoint_read` reads in successively the checkpoints given in the '.chk'-file, the configs-array will nevertheless be completely restored to its original state although each checkpoint-file contains only a fraction of the whole array.

### 10.2.2. Reading a checkpoint (deprecated)

#### Syntax

```
| checkpoint_read source
```

#### Description

Restores all the checkpoints whose filenames are listed in *source* in the order given therein, consequently putting the simulation into the state it was in when `checkpoint_set` was called. If parts of the configs array are given in the files listed in *source*, it is assumed that they represent a fraction of the whole array.

### 10.2.3. Writing a checkpoint 2 (deprecated)

#### Syntax

```
| (1) polyBlockWrite path ( param_list | all ) part_list
```



### Description

Variant (1) writes out the current ESPResSo-configuration as a blockfile, including parameters, interactions, particles, and bonds. *path* should contain the filename including the full path to it. *paramlist* gives a tcl-list of the ESPResSo-parameters to be saved; if an empty list {} is supplied, no parameters are written. If **all**, all global variables are written. This defaults to **all**. *partlist* gives a list of the particle-properties (out of **pos**, **type**, **q**, **v**, **f**) to be saved to disk; if an empty list {} is provided, no particles, no bonds, and no interactions are written. Defaults to all particle properties. If the suffix of *path* is **.gz**, the output will be compressed.

## 10.2.4. Writing a checkpoint 3 (deprecated)

### Syntax

```
| (2) polyBlockWriteAll destination [( tclvar | all ) [( whatever |- )  
| [( state | seed |- )]]]
```

### Description

Variant (2) saves all current interactions, particles, bonds, and global variables to *destination*, but in addition it also saves the tcl-variables specified by *tclvar* (if **all**, then all the variables in the active script are stored), it saves all the stored configurations if *whatever* is **whatever**, but **-**. Furthermore, it saves the state (**state**) or the seed (**seed**) of the random number generator.

## 10.3. Writing VTF files

The formats VTF (**VTF Trajectory Format**), VSF (**VTF Structure Format**) and VCF (**VTF Coordinate Format**) are formats for the visualization software VMD[24]<sup>1</sup>. 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<sup>2</sup>.

Creating files in these formats from within ESPResSo is supported by the commands **writevsvf** and **writevcf**, that write a structure respectively a coordinate block to the given Tcl channel. To create a VTF file, first use **writevsvf** at the beginning of the

<sup>1</sup><http://www.ks.uiuc.edu/Research/vmd/>

<sup>2</sup><https://github.com/olenz/vtfplugin/wiki/VTF-format>

simulation, and then `writvcf` 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 `writvcf`, 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.

### 10.3.1. `writevsf`: Writing the topology

#### Syntax

```
writevsf channelId [( short | verbose )] [radius ( radii | auto )]
               [typedesc typedesc] [ignore_charges]
```

#### 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

```
writvtsf $file typedesc {0 "name colloid" 1 "name pe"}
```

- `[ignore_charges]` this is a temporary workaround for a bug in the VTF reader of VMD, which cannot handle multiple atom sections for the same atom. However, charges are written out per-atom, so that VMD forgets about atom types and radii. If you don't need the charges, you can switch them off by this flag, making the other settings work again.

### 10.3.2. writvcf: Writing the coordinates

#### Syntax

```
writvcf 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"}
```

### 10.3.3. vtfpid: Translating ESPResSo particles ids to VMD particle ids

#### 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.

## 10.4. 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 10.3), 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.

### 10.4.1. writepsf: Writing the topology

#### Syntax

```
| writepsf file [-molecule] NP MPC NCI NpS NnS
```

#### Description

Writes the current topology to the file *file* (here, *file* is not a channel, since additional information cannot be written anyway). *N<sub>P</sub>*, *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`.

### 10.4.2. writepdb: Writing the coordinates

#### 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).

## 10.5. 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.

### 10.5.1. imd: Using IMD in the script

#### 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 10.5.3 on the next page) 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.

### 10.5.2. 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.

### 10.5.3. Automatically setting up a VMD connection

#### Syntax

```
|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 *constraints* is not 0, then the command will create graphics primitives in VMD that represent some of the spatial constraints (sphere, rhomboid and cylinder at present).

When *wait* is provided, ESPResSo will wait for *wait* seconds for VMD to connect, and only continue after this time has passed or VMD has connected. If *start* is 1 (the default), it will automatically try to start VMD and connect to the ESPResSo simulation, otherwise it writes a corresponding script to the file *filename.vmd\_start.script*, that can be executed via VMD, either from the command line

```
vmd -e vmd_start.script
```

or from the Tcl console of VMD with the command

```
play "vmd_start.script"
```

## 10.6. Errorhandling

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.

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

## 11.1. Center of mass motion

### 11.1.1. `system_com`

#### *Syntax*

| `system_com`

#### *Description*

Returns the center of mass of the whole system.

### 11.1.2. `system_com_vel`

#### *Syntax*

| `system_com_vel`

#### *Description*

Returns the velocity of the center of mass of the whole system.

### 11.1.3. `galileiTransformParticles`

#### *Syntax*

| `galileiTransformParticles`

#### *Description*

Subtracts 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 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.

## 11.2. Finding particles and bonds

### 11.2.1. `countBonds`

#### *Syntax*

| `countBonds` *particle<sub>i</sub>ist*



### 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 26).

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.

## 11.2.2. findPropPos

### 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]**.

## 11.2.3. findBondPos

### Syntax

```
| findBondPos particle_property_list
```

### Description

Returns the index of the bonds within *particle\_property\_list*, which is expected to have the same format as **[part particle\_number]**; hence its output is the same as **[findPropPos particle\_property\_list bonds]**. If the particle does not have any bonds, -1 is returned.

#### 11.2.4. timeStamp

##### 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.

### 11.3. 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  $\text{Coulomb}^2/(\text{Joule meter})$

- ATOMICMASS

- Returns the atomic mass unit  $u$  in kilograms

- 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 whether 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 choosing 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 success. On average the procedure needs 5.739 random numbers per vector. (This is probably not the most efficient way, but it works!) Ask your favorite 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) or the particle position directly When the optional *box* parameter for minimum image conventions is omitted the functions use the `setmd box_1` 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 radian. 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 radian. 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 =  $(a \times b)$
- `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 omitted or set to rnd then  $phi$  is assigned a random value between 0 and 2 Pi.

#### • TCL LIST OPERATIONS

- `average <list>`  
Returns the average 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 whether *list* contains *val*. returns the number of occurrences of *val* in *list*.

- REGRESSION

- `LinRegression <l>`

where  $l$  is a list of pairs of points  $\{ \{x_1 \ y_1\} \{x_2 \ y_2\} \dots \}$ . `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  $\{ \{x_1 \ y_1 \ s_1\} \{x_2 \ y_2 \ s_2\} \dots \}$  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  $\text{cov}(a, b)$  and  $\chi$ .

### 11.3.1. `t_random`

- Without further arguments,

`t_random`

returns a random double between 0 and 1 using the 'ran1' random number generator from Numerical Recipes.

- `t_random int <n>`

returns a random integer between 0 and  $n-1$ .

- `t_random seed`

returns a tcl-list with the seeds of the random number generators on each of the 'n\_nodes' nodes, while

`t_random 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 use this tcl-command to reset the sequences' seeds.

- Since internally the random number generators' random sequences are not based on mere seeds but rather on whole random number tables, to recover the exact state of the random number generators at a given time during the simulation run (e. g. for saving a checkpoint) requires knowledge of all these values. They can be accessed by

`t_random stat`

which returns a tcl-list with all status informations for any node (e. g. 8 nodes => approx. 350 parameters). To overwrite those internally in Espresso (e. g. upon restoring a checkpoint) submit the whole list back using

`t_random stat <status-list>`

with *status* – *list* being the tcl-list mentioned above without any braces. Be careful! A complete recovery of the current state of the simulation is only possible if you make sure to include a call to `The invalidate_system` command after you saved the checkpoint (`tcl_checkpoint_set` will do this automatically for you), because the integration algorithm re-uses the old forces calculated in the previous time-step; if something has changed in the system (or if it has just been read from a file) the forces are re-derived (including application of the thermostat and its random numbers) leading to slightly different results compared to the uninterrupted run (see `The invalidate_system` command for details)!

The C implementation is `t_random`

### 11.3.2. The `bit_random` command

- Without further arguments,

`bit_random`

returns a random double between 0 and 1 using the R250 generator XOR-ing a table of 250 linear independent integers.

- `bit_random seed`

returns a tcl-list with the seeds of the random number generators on each of the '`n_nodes`' nodes, while

`bit_random 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 use this tcl-command to reset the sequences' seeds.

- Since internally the random number generators' random sequences are not based on mere seeds but an array of 250 linear independent integers whose bits are used as matrix elements which are XOR-ed, to recover the exact state of the random number generators at a given time during the simulation run (e. g. for saving a checkpoint) requires knowledge of all these values. They can be accessed by

`bit_random stat`

which returns a tcl-list with all status informations for any node (e. g. 8 nodes => approx. 2016 parameters). To overwrite those internally in Espresso (e. g. upon restoring a checkpoint) submit the whole list back using

`bit_random stat <status-list>`

with `jstatus-listj` being the tcl-list mentioned above without any braces. Be careful! A complete recovery of the current state of the simulation is only possible if you make sure to include a call to `The invalidate_system` command after you saved

the checkpoint (`tcl.checkpoint.set` will do this automatically for you), because the integration algorithm re-uses the old forces calculated in the previous time-step; if something has changed in the system (or if it has just been read from a file) the forces are re-derived (including application of the thermostat and its random numbers) leading to slightly different results compared to the uninterrupted run (see `The invalidate.system` command for details)!

- Note further that the bit-wise display of integers, as it is used by this random number generator, is platform dependent. As long as you stay on the same architecture this doesn't matter at all; however, it wouldn't be wise to use a checkpoint including the state of the R250 to restart the simulation on a different platform - most likely, the integers will have a different bit-muster leading to a completely different random matrix. So, if you're using this random number generator, always remain on the same platform!

## 11.4. 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

#### Syntax

```
lbfluid [gpu]2 [agrid agrid]1 or 2 [dens density]1 or 2 [visc viscosity]1 or 2  
      [tau lb_timestep]1 or 2 [bulk_visc bulk_viscosity]1 or 2  
      [ext_force f_x f_y f_z]1 or 2 [friction gamma]1 or 2  
      [gamma_odd gamma_odd]1 or 2 [gamma_even gamma_even]1 or 2  
Required features: 1LB 2LB_GPU
```

#### 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,  $f_x = f_y = f_z = 0$ .

#### Syntax

```
| lbfluid print_interpolated_velocity x y z
```

#### Description

This variant returns the velocity at point in continuous space. This can make it easier to calculate flow profiles independent of the lattice constant.

#### Syntax

```
| lbfluid print [vtk] property filename
```

#### Description

The print parameter 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. The additional option `vtk` enables export in the vtk format which is readable by visualization software such as paraview or mayavi. Otherwise gnuplot readable data will be exported.

#### 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.

This is currently only implemented for the cpu version of LB.

## 12.2. LB as a thermostat

### Syntax

```
| thermostat lb1 or 2 T  
| Required features: 1LB 2LB_GPU
```

### Description

The LBM implementation in ESPResSo uses Duenweg's point coupling method to couple MD particles the LB fluid. This coupling consists in a frictional force and a random force:

$$\vec{F} = -\gamma (\vec{v} - \vec{u}) + \vec{F}_R.$$

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 thermostating mechanism is necessary then. Please any of these off before starting the LB thermostating 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.

## 12.3. Reading and setting single lattice nodes

### Syntax

```
| lbnode x y z ( print | set ) args  
| Required features: LB
```

### 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:

<b>rho</b>	the density (scalar).
<b>u</b>	the fluid velocity (three floats: $u_x$ , $u_y$ , $u_z$ )
<b>pi</b>	the fluid velocity (six floats: $\Pi_{xx}$ , $\Pi_{xy}$ , $\Pi_{yy}$ , $\Pi_{xz}$ , $\Pi_{yz}$ , $\Pi_{zz}$ )
<b>pi_neq</b>	the nonequilibrium part of the pressure tensor, components as above.
<b>pop</b>	the 19 population (check the order from the source code please).
<b>boundary</b>	the flag indicating whether the node is a fluid node ( <b>boundary</b> = 0) or a boundary node ( <b>boundary</b> $\neq$ 0). Does not support <b>set</b> . Refer to the <b>lbboundary</b> command for this functionality.

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.]
```

## 12.4. Setting up boundary conditions

### Syntax

```
| lbboundary shape shape_args [velocity vx vy vz]
| Required features: LB_BOUNDARIES
```

### Description

If nothing else is specified periodic boundary conditions are assumed for the LB fluid. The **lbboundary** command 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 similarly 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 independent of the viscosity. This can *e.g.* be seen when using the sample script `poisseuille.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 [49] eq. 12.58. Using this implementation as a blueprint for the boundary treatment an implementation of the Ladd-Coupling should be relatively straightforward.

#### *Syntax*

```
| lbboundary force [ nboundary]  
| Required features: LB_BOUNDARIES
```

#### *Description*

This variant prints out the force on boundary number `n_boundary`.

## 12.5. Choosing between the GPU and CPU implementations

#### *Syntax*

```
| (1) lbfluid cpu  
| (2) lbfluid gpu  
| Required features: 1LB 2LB_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 `ESPReso-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 6.6.

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.

## 12.6. Electrohydrodynamics

### *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 implicitly 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 [21].

## 13. External package: mbtools

mbtools<sup>1</sup> 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

### 13.1. 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 procedures. 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.

---

<sup>1</sup>This documentation was written by Ira R. Cooke and published on his website. It has been transcribed by Tristan Bereau.

## 13.2. 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_box1_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
```

### 13.3. 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.

#### 13.3.1. 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 13.5)
- *\*moltypes* A list of molecule types (see documentation in 13.5)
- *\*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

## 13.4. 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.

### 13.4.1. 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.

### 13.4.2. 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 -nograd  
-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`.

### 13.4.3. 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_myanalysis
}

```

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.

## 13.5. System generation

Package for setting up lipid membrane systems in a variety of geometrical shapes.

### 13.5.1. Basic commands

```
::mbtools::system_generation::setup_system : [system_specs]  
[ibox1] [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_specs`

```

set geometry { geometry "flat -fixz" }
set n_molslist { n_molslist { { 1 3000 } } }
lappend bilayerspec $geometry
lappend bilayerspec $n_molslist

```

Now group together the *system\_specs* 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*.

### 13.5.2. 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 is defined, then no molecules will be placed 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\_molslist* 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.

### 13.5.3. 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 procedure `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` procedure within your new namespace as follows

```
::mbtools::system_generation::mygeom::create_mygeom args
```

### 13.5.4. 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 *bondtypeids*. 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 : { multypeid "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 : { multypeid "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 *bondtypeids* with the same meaning as explained above for standard lipids.

Create a lipid which spans across the bilayer.

### 13.5.5. Adding a new molecule type

To add a new molecule type you need to define a procedure 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.

## 13.6. Utils

Useful utilities routines for various types. Includes file management, basic geometry and math procedures.

### 13.6.1. 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\_parms* 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 **writpdb** 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 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.



### 13.6.2. 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.

### 13.6.3. 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.

#### 13.6.4. 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 position 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.

### 13.6.5. 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.

## 13.7. 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.

### 13.7.1. 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.

### 13.7.2. 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"
```

## 14. Under the hood

- Implementation issues that are interesting for the user
- Main loop in pseudo code (for comparison)

### 14.1. 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.<sup>a</sup>

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 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 resp. 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.



## 15. 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

### 15.1. 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.

## 15.2. 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.

## 15.3. 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>.

## 15.4. 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 L<sup>A</sup>T<sub>E</sub>X package `todonotes`.

You can then build the user guide by typing

```
make ug
```

## A. ESPResSo quick reference

<hr/> <code>part <i>pid</i> [pos <i>x y z</i>] [type <i>typeid</i>] [v <i>vx vy vz</i>] [f <i>fx fy fz</i>]</code>	26
<code>  [bond <i>bondid pid2 ...</i>] [temp <i>T</i>]<sup>9</sup> [gamma <i>g</i>]<sup>9</sup> [q <i>charge</i>]<sup>1</sup></code> <code>  [quat <i>q1 q2 q3 q4</i>]<sup>2</sup> [omega <i>x y z</i>]<sup>2</sup> [torque <i>x y z</i>]<sup>2</sup></code> <code>  [rinertia <i>x y z</i>]<sup>2</sup> [[un]fix <i>x y z</i>]<sup>3</sup> [ext_force <i>x y z</i>]<sup>3</sup></code> <code>  [exclude <i>pid2...</i>]<sup>4</sup> [exclude delete <i>pid2...</i>]<sup>4</sup> [mass <i>mass</i>]<sup>5</sup></code> <code>  [dipm <i>moment</i>]<sup>6</sup> [dip <i>dx dy dz</i>]<sup>6</sup> [virtual <i>v</i>]<sup>7,8</sup></code> <code>  [vs_relative <i>pid distance</i>]<sup>8</sup> [vs_auto_relate_to <i>pid</i>]<sup>8</sup></code>	
Required features: <sup>1</sup> ELECTROSTATICS <sup>2</sup> ROTATION <sup>3</sup> EXTERNAL_FORCES <sup>4</sup> EXCLUSION	
<sup>5</sup> MASS <sup>6</sup> DIPOLES <sup>7</sup> VIRTUAL_SITES_COM <sup>8</sup> VIRTUAL_SITES_RELATIVE	
<sup>9</sup> LANGEVIN_PER_PARTICLE	
<hr/> <code>part <i>pid</i> print [( id   pos   type   folded_position   type   q   v   f  </code>	28
<code>  fix   ext_force   bond   connections [<i>range</i>] )]...</code> <code>part</code>	
<hr/> <code>part <i>pid</i> delete</code>	29
<code>part deleteall</code>	
<hr/> <code>part auto_exclusions [<i>range</i>]</code>	29
<code>part delete_exclusions</code>	
Required features: EXCLUSIONS	
<hr/> <code>polymer <i>num_polymers monomers_per_chain bond_length</i></code>	30
<code>  [start <i>pid</i>] [pos <i>x y z</i>] [mode ( RW   SAW   PSAW ) [<i>shield [try_max]</i>]]</code> <code>  [charge <i>q</i>]<sup>1</sup> [distance <i>d_charged</i>]<sup>1</sup> [types <i>typeid_neutral [typeid_charged]</i>]</code> <code>  [bond <i>bondid</i>] [angle <math>\phi</math> [<math>\theta</math> [<i>x y z</i>]]] [constraints]<sup>2</sup></code>	
Required features: <sup>1</sup> ELECTROSTATICS <sup>2</sup> CONSTRAINTS	
<hr/> <code>counterions <i>N</i> [start <i>pid</i>] [mode ( SAW   RW ) [<i>shield [try_max]</i> ]]</code>	31
<code>  [charge <i>val</i>]<sup>1</sup> [type <i>typeid</i>]</code>	
Required features: <sup>1</sup> ELECTROSTATICS	
<hr/> <code>salt <i>N<sub>+</sub> N<sub>-</sub></i> [start <i>pid</i>] [mode ( SAW   RW ) [<i>shield [try_max]</i>]]</code>	32
<code>  [charges <i>val<sub>+</sub> [val<sub>-</sub>]]<sup>1</sup> [types <i>typeid<sub>+</sub> [typeid<sub>-</sub>]</i>] [rad <i>r</i>]</i></code>	
Required features: <sup>1</sup> ELECTROSTATICS	
<hr/> <code>diamond <i>a bond_length monomers_per_chain</i> [counterions <i>N<sub>CI</sub></i>]</code>	32
<code>  [charges <i>val<sub>node val<sub>monomer val<sub>CI</sub></sub></sub></i>]<sup>1</sup> [distance <i>d_charged</i>]<sup>1</sup> [nonet]</code>	
Required features: <sup>1</sup> ELECTROSTATICS	

<code>icosaeder</code> <i>a</i> <i>monomers_per_chain</i> [counterions <i>N<sub>CI</sub></i> ] [charges <i>val<sub>monomers</sub></i> <i>val<sub>CI</sub></i> ] <sup>1</sup> [distance <i>d<sub>charged</sub></i> ] <sup>1</sup>	34
Required features: <sup>1</sup> ELECTROSTATICS	
<code>crosslink</code> <i>num_polymer</i> <i>monomers_per_chain</i> [start <i>pid</i> ] [catch <i>r<sub>catch</sub></i> ] [distLink <i>link_dist</i> ] [distChain <i>chain_dist</i> ] [FENE <i>bondid</i> ] [trials <i>try<sub>max</sub></i> ]	35
<code>copy_particles</code> [set <i>id1 id2 ...</i> ] range from to ... [shift <i>s_x s_y s_z</i> ]	35
<code>constraint wall</code> normal <i>n<sub>x</sub> n<sub>y</sub> n<sub>z</sub></i> dist <i>d</i> type <i>id</i> [penetrable <i>flag</i> ] [reflecting <i>flag</i> ]	36
<code>constraint sphere</code> center <i>c<sub>x</sub> c<sub>y</sub> c<sub>z</sub></i> radius <i>rad</i> direction <i>direction</i> type <i>id</i> [penetrable <i>flag</i> ] [reflecting <i>flag</i> ]	
<code>constraint cylinder</code> center <i>c<sub>x</sub> c<sub>y</sub> c<sub>z</sub></i> axis <i>n<sub>x</sub> n<sub>y</sub> n<sub>z</sub></i> radius <i>rad</i> length <i>length</i> direction <i>direction</i> type <i>id</i> [penetrable <i>flag</i> ] [reflecting <i>flag</i> ]	
<code>constraint rhomboid</code> corner <i>p<sub>x</sub> p<sub>y</sub> p<sub>z</sub></i> a <i>a<sub>x</sub> a<sub>y</sub> a<sub>z</sub></i> b <i>b<sub>x</sub> b<sub>y</sub> b<sub>z</sub></i> c <i>c<sub>x</sub> c<sub>y</sub> c<sub>z</sub></i> direction <i>direction</i> type <i>id</i> [penetrable <i>flag</i> ] [reflecting <i>flag</i> ]	
<code>constraint maze</code> nsphere <i>n</i> dim <i>d</i> sphrad <i>r<sub>s</sub></i> cylrad <i>r<sub>c</sub></i> type <i>id</i> [penetrable <i>flag</i> ]	
<code>constraint pore</code> center <i>c<sub>x</sub> c<sub>y</sub> c<sub>z</sub></i> axis <i>n<sub>x</sub> n<sub>y</sub> n<sub>z</sub></i> radius <i>rad</i> length <i>length</i> type <i>id</i>	
<code>constraint rod</code> center <i>c<sub>x</sub> c<sub>y</sub></i> lambda <i>lambda</i> <sup>1</sup>	
<code>constraint plate</code> height <i>h</i> sigma <i>sigma</i> <sup>1</sup>	
<code>constraint ext_magn_field</code> <i>f<sub>x</sub> f<sub>y</sub> f<sub>z</sub></i> <sup>2,3</sup>	
<code>constraint plane</code> cell <i>x y z</i> type <i>id</i>	
<code>constraint mindist_position</code> <i>x y z</i>	
Required features: CONSTRAINTS <sup>1</sup> ELECTROSTATICS <sup>2</sup> ROTATION <sup>3</sup> DIPOLES	
<code>constraint delete</code> [ <i>num</i> ]	38
<code>constraint force</code> <i>n</i>	38
<code>constraint</code> [ <i>num</i> ]	38
<code>inter</code>	42
<code>inter</code> <i>type1 type2</i> tabulated <i>filename</i>	42
Required features: TABULATED	
<code>inter</code> <i>type1 type2</i> lennard-jones $\epsilon$ $\sigma$ <i>r<sub>cut</sub></i> [( <i>c<sub>shift</sub></i>  auto ) [ <i>r<sub>off</sub></i> [ <i>r<sub>cap</sub></i> [ <i>r<sub>min</sub></i> ]]]]	43
Required features: LENNARD_JONES	
<code>inter</code> <i>type1 type2</i> lj-gen $\epsilon$ $\sigma$ <i>r<sub>cut</sub></i> <i>c<sub>shift</sub></i> <i>r<sub>off</sub></i> <i>e<sub>1</sub> e<sub>2</sub></i> <i>b<sub>1</sub> b<sub>2</sub></i> [( <i>r<sub>cap</sub></i>  auto )]	43
Required features: LENNARD_JONES_GENERIC	
<code>inter</code> <i>type1 type2</i> lj-cos $\epsilon$ $\sigma$ <i>r<sub>cut</sub></i> <i>r<sub>off</sub></i>	44
<code>inter</code> <i>type1 type2</i> lj-cos2 $\epsilon$ $\sigma$ <i>r<sub>off</sub></i> $\omega$	
Required features: LJCOS LJCOS2	

<code>inter type1 type2 smooth-step <math>\sigma_1</math> <math>n</math> <math>\epsilon</math> <math>k_0</math> <math>\sigma_2</math> <math>r_{\text{cut}}</math></code>	44
Required features: SMOOTH_STEP	
<code>inter type1 type2 bmhtf-nacl <math>A</math> <math>B</math> <math>C</math> <math>D</math> <math>\sigma</math> <math>r_{\text{cut}}</math></code>	45
Required features: BMHTF_NACL	
<code>inter type1 type2 morse <math>\epsilon</math> <math>\alpha</math> <math>r_{\text{min}}</math> <math>r_{\text{cut}}</math></code>	45
Required features: MORSE	
<code>inter type1 type2 buckingham <math>A</math> <math>B</math> <math>C</math> <math>D</math> <math>r_{\text{cut}}</math> <math>r_{\text{discont}}</math> <math>\epsilon_{\text{shift}}</math></code>	46
Required features: BUCKINGHAM	
<code>inter type1 type2 soft-sphere <math>a</math> <math>n</math> <math>r_{\text{cut}}</math> <math>r_{\text{offset}}</math></code>	46
Required features: SOFT_SPHERE	
<code>inter type1 type2 hertzian <math>\sigma</math> <math>\epsilon</math></code>	47
Required features: HERTZIAN	
<code>inter type1 type2 lj-angle <math>\epsilon</math> <math>\sigma</math> <math>r_{\text{cut}}</math> <math>b1_a</math> <math>b1_b</math> <math>b2_a</math> <math>b2_b</math> [<math>r_{\text{cap}}</math> <math>z_0</math> <math>\delta z</math> <math>\kappa</math> <math>\epsilon'</math>]</code>	47
Required features: LJ_ANGLE	
<code>inter type1 type2 gay-berne <math>\epsilon_0</math> <math>\sigma_0</math> <math>r_{\text{cutoff}}</math> <math>k1</math> <math>k2</math> <math>\mu</math> <math>\nu</math></code>	47
Required features: ROTATION GAY_BERNE	
<code>inter bondid fene <math>K</math> <math>\Delta r_{\text{max}}</math> [<math>r_0</math>]</code>	49
<code>inter bondid harmonic <math>K</math> <math>R</math> [<math>r_{\text{cut}}</math>]</code>	50
<code>inter bondid subt_lj reserved <math>R</math></code>	50
<code>inter bondid rigid_bond constrained_bond_distance positional_tolerance velocity_tolerance</code>	51
<code>inter bondid tabulated bond filename</code>	51
<code>inter bondid tabulated angle filename</code>	
<code>inter bondid tabulated dihedral filename</code>	
<code>inter bondid virtual_bond</code>	51
<code>inter bondid angle <math>K</math> [<math>\phi_0</math>]</code>	52
Required features: BOND_ANGLE_HARMONIC , BOND_ANGLE_COSINE or BOND_ANGLE_- COSSQUARE	
<code>inter bondid dihedral <math>n</math> <math>K</math> <math>p</math></code>	52
<code>inter coulomb 0.0</code>	53
<code>inter coulomb</code>	
<code>inter coulomb parameters</code>	
<code>inter coulomb <math>l_B</math> p3m <math>r_{\text{cut}}</math> ( mesh   {<math>mesh_x</math> <math>mesh_y</math> <math>mesh_z</math>} ) cao alpha</code>	54
Required features: ELECTROSTATICS	
<code>inter coulomb <math>l_B</math> p3m ( tune   tunev2 ) accuracy accuracy [<math>r_{\text{cut}}</math> <math>r_{\text{cut}}</math>] [mesh mesh] [cao cao] [alpha <math>\alpha</math>]</code>	55
Required features: ELECTROSTATICS	
<code>inter coulomb [epsilon ( metallic   epsilon )] [n_interpol points] [mesh_off xoff yoff zoff]</code>	55

<code>inter coulomb <math>l_B</math> dh <math>\kappa</math> <math>r_{\text{cut}}</math></code>	56
Required features: ELECTROSTATICS	
<code>inter coulomb <math>l_B</math> mmm2d maximal_pairwise_error [fixed_far_cutoff] [dielectric <math>\epsilon_t</math> <math>\epsilon_m</math> <math>\epsilon_b</math>] [dielectric-contrasts <math>\Delta_t</math> <math>\Delta_b</math>]</code>	57
Required features: ELECTROSTATICS	
<code>inter coulomb <math>l_B</math> mmm1d switch_radius [bessel_cutoff] maximal_pairwise_error</code>	57
<code>inter coulomb <math>l_B</math> mmm1d tune maximal_pairwise_error</code>	
Required features: ELECTROSTATICS	
<code>inter coulomb <math>l_B</math> memd f_mass mesh [epsilon <math>\epsilon_\infty</math>]</code>	58
Required features: ELECTROSTATICS	
<code>inter coulomb elc maximal_pairwise_error gap_size [far_cutoff] [noneutralization] [dielectric <math>\epsilon_t</math> <math>\epsilon_m</math> <math>\epsilon_b</math>] [dielectric-contrasts <math>\Delta_t</math> <math>\Delta_b</math>]</code>	58
Required features: ELECTROSTATICS	
<code>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]</code>	59
Required features: ELECTROSTATICS	
<code>dielectric sphere center cx cy cz radius r res res</code>	60
<code>dielectric wall normal nx ny nz dist d res res</code>	
<code>dielectric cylinder center cx cy cz axis ax ay az radius r direction d</code>	
<code>dielectric pore center cx cy cz axis ax ay az radius r length l smoothing_radius rs res res</code>	
<code>inter magnetic 0.0</code>	61
<code>inter magnetic</code>	
<code>inter magnetic parameters</code>	
<code>inter magnetic <math>l_B</math> p3m <math>r_{\text{cut}}</math> mesh cao alpha</code>	62
Required features: DIPOLES	
<code>inter magnetic <math>l_B</math> p3m ( tune   tunev2 ) accuracy accuracy [r_cut <math>r_{\text{cut}}</math>] [mesh mesh] [cao cao] [alpha <math>\alpha</math>]</code>	62
Required features: DIPOLES	
<code>inter magnetic mdlc accuracy gap_size [far_cutoff]</code>	63
Required features: DIPOLES	
<code>inter magnetic <math>l_B</math> dawaanr</code>	63
Required features: DIPOLES	
<code>inter magnetic <math>l_B</math> mdds n_cut value_n_cut</code>	64
Required features: DIPOLES MAGNETIC_DIPOLAR_DIRECT_SUM	

<code>inter type1 type2 tunable_slip T <math>\gamma_L</math> r_cut <math>\delta t</math> v_x v_y v_z</code>	64
Required features: TUNABLE_SLIP	
<code>inter type1 type2 inter_dpd gamma r_cut wf tgamma tr_cut twf</code>	64
Required features: INTER_DPD	
<code>inter typeid1 typeid1 comfixed flag</code>	65
Required features: COMFIXED	
<code>inter typeid1 typeid2 comforce flag dir force fratio</code>	66
Required features: COMFORCE	
<code>inter ljforcecap ( <math>F_{\max}</math>   individual )</code>	66
<code>inter morseforcecap ( <math>F_{\max}</math>   individual )</code>	
<code>inter buckforcecap ( <math>F_{\max}</math>   individual )</code>	
<code>inter tabforcecap ( <math>F_{\max}</math>   individual )</code>	
Required features: LENNARD_JONES MORSE BUCKINGHAM TABULATED	
<code>setmd variable</code>	66
<code>setmd variable [value]+</code>	
<code>thermostat</code>	67
<code>thermostat off</code>	
<code>thermostat parameters</code>	
<code>thermostat langevin temperature gamma</code>	68
<code>thermostat dpd temperature gamma r_cut [ WF wf tgamma tr_cut TWF twf]</code>	70
Required features: DPD or TRANS_DPD	
<code>thermostat inter_dpd temperature</code>	70
Required features: INTER_DPD	
<code>thermostat npt_isotropic temperature gamma0 gammaV</code>	71
Required features: NPT	
<code>nemd exchange n_slabs n_exchange</code>	72
<code>nemd shearrate n_slabs shearrate</code>	
<code>nemd off</code>	
<code>nemd</code>	
<code>nemd profile</code>	
<code>nemd viscosity</code>	
Required features: NEMD	
<code>cellsystem domain_decomposition [-no_verlet_list]</code>	73
<code>cellsystem nsquare</code>	73
<code>cellsystem layered n_layers</code>	74
<code>adress set topo kind width width hybrid_width center x R_x wf wf</code>	75
Required features: ADRESSO	
<code>cuda list</code>	76
<code>cuda setdevice id</code>	
<code>cuda getdevice</code>	

collision_detection	76
collision_detection off	
collision_detection bind_centers <i>d bond1</i>	
collision_detection bind_at_point_of_collision <i>d bond1 bond2 type</i>	
integrate <i>steps</i>	76
integrate set [nvt]	
integrate set npt_isotropic <i>p_ext piston [x y z] [-cubic_box]</i>	
change_volume <i>V<sub>new</sub></i>	77
change_volume <i>L<sub>new</sub> ( x   y   z   xyz )</i>	
stopParticles	79
stop_particles	
velocities <i>v<sub>max</sub> [start pid] [count N]</i>	79
invalidate_system	80
parallel_tempering::main -rounds <i>N</i> -swap <i>swap</i> -perform <i>perform</i> [-init <i>init</i> ] [-values { <i>T<sub>i</sub></i> }] [-connect <i>master</i> ] [-port <i>port</i> ] [-load <i>jnode</i> ] [-resrate <i>N<sub>reset</sub></i> ] [-info <i>info</i> ]	80
parallel_tempering::set_shareddata <i>data</i>	80
metadynamics	81
metadynamics set off	
metadynamics set distance <i>pid<sub>1</sub> pid<sub>2</sub> d<sub>min</sub> d<sub>max</sub> b<sub>height</sub> b<sub>width</sub> f<sub>bound</sub> d<sub>bins</sub></i>	
metadynamics set relative_z <i>pid<sub>1</sub> pid<sub>2</sub> z<sub>min</sub> z<sub>max</sub> b<sub>height</sub> b<sub>width</sub> f<sub>bound</sub> z<sub>bins</sub></i>	
metadynamics print_stat current_coord	
metadynamics print_stat coord_values	
metadynamics print_stat profile	
metadynamics print_stat force	
metadynamics load_stat <i>profile_list force_list</i>	
Required features: METADYNAMICS	
analyze mindist [ <i>type_list_a type_list_b</i> ]	84
analyze distto <i>pid</i>	
analyze distto <i>x y z</i>	
analyze nbhood <i>pid r_catch</i>	85
analyze nbhood <i>x y z r_catch</i>	
analyze distribution <i>part_type_list_a part_type_list_b</i> [ <i>rmin [rmax [rbins [log_flag [int_flag]]]]</i> ]	87
analyze radial_density_map <i>xbins ybins xrange yrange</i> [ <i>axisofrotation centerofrotation beadtpeelist [thetabins]</i> ]	88
analyze modes2d	88
analyze get_lipid_orients	88
analyze lipid_orient_order	
analyze bilayer_set	89
analyze bilayer_density_profile	



analyze cell_gpb <i>Manningparameter outercellradius innercellradius</i> [ <i>accuracy [numberofinteractions]</i> ]	89
analyze get_folded_positions [-molecule] [shift <i>x y z</i> ]	89
analyze V <sub>kappa</sub> [( reset   read   set <i>V<sub>κ,1</sub> V<sub>κ,2</sub> avk</i> ) ]	89
analyze ( rdf   <rdf> ) <i>part_type_list_a part_type_list_b</i> [ <i>rmin rmax rbins</i> ]	89
analyze structurefactor <i>type order</i>	90
analyze vanhove <i>type rmin rmax rbins</i> [ <i>tmax</i> ]	90
analyze centermass <i>part_type</i>	90
analyze momentofinertiamatrix <i>typeid</i>	91
analyze find_principal_axis <i>typeid</i>	
analyze gyration_tensor [ <i>typeid</i> ]	91
analyze aggregation <i>dist_criteria s_mol_id f_mol_id</i> [ <i>min_contact [charge_criteria]</i> ]	91
analyze necklace <i>pearl_threshold back_dist space_dist first length</i>	92
analyze holes <i>typeid<sub>probe</sub> mesh_size</i>	92
analyze energy	92
analyze energy ( total   kinetic   coulomb   magnetic )	
analyze energy bonded <i>bondid</i>	
analyze energy nonbonded <i>typeid1 typeid2</i>	
analyze pressure	93
analyze pressure total	
analyze pressure ( totals   ideal   coulomb   tot_nonbonded_inter   tot_nonbonded_intra )	
analyze pressure bonded <i>bondid</i>	
analyze pressure nonbonded <i>typeid1 typeid2</i>	
analyze pressure nonbonded_intra [ <i>typeid</i> ]	
analyze pressure nonbonded_inter [ <i>typeid</i> ]	
analyze stress_tensor	94
analyze stress_tensor total	
analyze stress_tensor ( totals   ideal   coulomb   tot_nonbonded_inter   tot_nonbonded_intra )	
analyze stress_tensor bonded <i>bond_type</i>	
analyze stress_tensor nonbonded <i>typeid1 typeid2</i>	
analyze stress_tensor nonbonded_intra [ <i>typeid</i> ]	
analyze stress_tensor nonbonded_inter [ <i>typeid</i> ]	
analyze local_stress_tensor <i>periodic_x periodic_y periodic_z range_start_x</i> <i>range_start_y range_start_z range_x range_y range_z bins_x bins_y</i> <i>bins_z</i>	94
analyze set chains [ <i>chain_start n_chains chain_length</i> ]	95
analyze set topo_part_sync	
analyze set	

<code>analyze ( re   &lt;re&gt; ) [chain_start n_chains chain_length]</code>	96
<code>analyze ( rg   &lt;rg&gt; ) [chain_start n_chains chain_length]</code>	97
<code>analyze ( rh   &lt;rh&gt; ) [chain_start n_chains chain_length]</code>	98
<code>analyze ( internal_dist   &lt;internal_dist&gt; ) [chain_start n_chains chain_length]</code>	98
<code>analyze ( bond_dist   &lt;bond_dist&gt; ) [index index]</code> <code>[chain_start n_chains chain_length]</code>	98
<code>analyze ( bond_l   &lt;bond_l&gt; ) [chain_start n_chains chain_length]</code>	99
<code>analyze ( formfactor   &lt;formfactor&gt; ) qmin qmax qbins</code> <code>[chain_start n_chains chain_length]</code>	99
<code>analyze rdfchain rmin rmax rbins [chain_start n_chains chain_length]</code>	100
<code>analyze ( &lt;g1&gt;  &lt;g2&gt;  &lt;g3&gt; ) [chain_start n_chains chain_length]</code>	100
<code>analyze g123 [-init] [chain_start n_chains chain_length]</code>	
<code>analyze append</code>	100
<code>analyze remove [index]</code>	
<code>analyze replace index</code>	
<code>analyze push [size]</code>	
<code>analyze configs config</code>	
<code>analyze configs</code>	101
<code>analyze stored</code>	
<code>uwarn data nrep col [s_tau] [plot]</code>	102
<code>uwarn data nrep f [s_tau [f_args]] [plot]</code>	
<code>observable new name [parameters+]</code>	102
<code>observable id print [formatted]</code>	103
<code>observable id delete</code>	105
<code>observable new needs_profile_specs [other_parameters] [ minx minx ]</code> <code>[ maxx maxx ] [ miny miny ] [ maxy maxy ] [ minz minz ]</code> <code>[ maxx maxx ] [ xbins xbins ] [ ybins ybins ] [ zbins zbins ]</code>	107
<code>observable new needs_radial_profile_specs [other_parameters]</code> <code>[ center &lt;cx&gt; &lt;cy&gt; &lt;cz&gt; ] [ maxx maxx ] [ minz minz ]</code> <code>[ maxx maxx ] [ rbins rbins ] [ phibins phibins ] [ zbins zbins ]</code>	107
<code>correlation new obs1 id1 [obs2 id2] corr_operation</code> <code>operation dt dt tau_max tau_max [tau_lin tau_lin]</code> <code>[compress1 name [compress2 name] ]</code>	108
<code>correlation</code>	108
<code>correlation n_corr</code>	
<code>correlation id autoupdate { start   stop}</code>	109
<code>correlation id update</code>	
<code>correlation id finalize</code>	

<code>correlation <i>id</i> write_to_file <i>filename</i></code>	111
<code>correlation <i>id</i> print</code>	
<code>correlation <i>id</i> print [ average1   variance1   correlation_time ]</code>	
<code>correlation <i>id</i> print [ average_errorbars ]</code>	
<code>blockfile <i>channel</i> write variable {<i>varname1 varname2 ...</i>}</code>	111
<code>blockfile <i>channel</i> write variable all</code>	
<code>blockfile <i>channel</i> write tclvariable { <i>varname1 varname2 ...</i>}</code>	112
<code>blockfile <i>channel</i> write tclvariable all</code>	
<code>blockfile <i>channel</i> write tclvariable reallyall</code>	
<code>blockfile <i>channel</i> write particles <i>what</i> ( <i>range</i>   all )</code>	115
<code>blockfile <i>channel</i> write bonds <i>range</i></code>	
<code>blockfile <i>channel</i> write interactions</code>	
<code>blockfile <i>channel</i> write random</code>	116
<code>blockfile <i>channel</i> write bit_random</code>	
<code>blockfile <i>channel</i> write seed</code>	
<code>blockfile <i>channel</i> write bitseed</code>	
<code>blockfile <i>channel</i> write configs</code>	116
<code>blockfile <i>channel</i> write start <i>tag</i></code>	116
<code>blockfile <i>channel</i> write end</code>	
<code>blockfile <i>channel</i> write <i>tag</i> [<i>arg</i>]...</code>	
<code>blockfile <i>channel</i> read start</code>	117
<code>blockfile <i>channel</i> read toend</code>	
<code>blockfile <i>channel</i> read ( particles   interactions   bonds   variable   seed   random   bitrandom   configs )</code>	
<code>blockfile <i>channel</i> read auto</code>	
<code>checkpoint_set <i>destination</i> [<i>num_configs</i> [<i>tclvar</i> [<i>iaflag</i> [<i>varflag</i> [<i>ranflag</i>]]]]]</code>	117
<code>checkpoint_read <i>source</i></code>	118
<code>polyBlockWrite <i>path</i> ( <i>param_list</i>   all ) <i>part_list</i></code>	119
<code>polyBlockWriteAll <i>destination</i> [( <i>tclvar</i>   all ) [( <i>whatever</i>  - ) [( <i>state</i>   seed  - )]]]</code>	120
<code>writevsvf <i>channelId</i> [( <i>short</i>   verbose )] [<i>radius</i> ( <i>radii</i>   auto )] [<i>typedesc typedesc</i>] [<i>ignore_charges</i>]</code>	120
<code>writevcf <i>channelId</i> [( <i>short</i>   verbose )] [( <i>folded</i>   absolute )] [<i>pids</i> ( <i>pids</i>   all )] [<i>userdata userdata</i>]</code>	121
<code>vtfpid <i>pid</i></code>	122
<code>writespf <i>file</i> [-molecule] <i>N<sub>P</sub> MPC N<sub>CI</sub> N<sub>p</sub>S N<sub>n</sub>S</i></code>	123
<code>writepdb <i>file</i></code>	124
<code>writepdbfoldchains <i>file chain_start n_chains chain_length box_l</i></code>	
<code>writepdbfoldtopo <i>file shift</i></code>	

<code>imd connect</code> [ <i>port</i> ]	124
<code>imd positions</code> [( <code>-unfolded</code>   <code>-fold_chains</code> )]	
<code>imd listen</code> <i>seconds</i>	
<code>imd disconnect</code>	
<code>prepare_vmd_connection</code> [ <i>filename</i> [ <i>wait</i> [ <i>start</i> ]] [ <i>constraints</i> ]]	124
<code>lbfluid</code> [ <i>gpu</i> ] <sup>2</sup> [ <i>agrid</i> <i>agrid</i> ] <sup>1 or 2</sup> [ <i>dens</i> <i>density</i> ] <sup>1 or 2</sup> [ <i>visc</i> <i>viscosity</i> ] <sup>1 or 2</sup> [ <i>tau</i> <i>lb_timestep</i> ] <sup>1 or 2</sup> [ <i>bulk_visc</i> <i>bulk_viscosity</i> ] <sup>1 or 2</sup> [ <i>ext_force</i> <i>f<sub>x</sub></i> <i>f<sub>y</sub></i> <i>f<sub>z</sub></i> ] <sup>1 or 2</sup> [ <i>friction</i> <i>gamma</i> ] <sup>1 or 2</sup> [ <i>gamma_odd</i> <i>gamma_odd</i> ] <sup>1 or 2</sup> [ <i>gamma_even</i> <i>gamma_even</i> ] <sup>1 or 2</sup>	125
Required features: <sup>1</sup> LB <sup>2</sup> LB_GPU	
<code>lbfluid print_interpolated_velocity</code> <i>x y z</i>	126
<code>lbfluid print</code> [ <i>vtk</i> ] <i>property filename</i>	128
<code>lbfluid save_ascii_checkpoint</code> <i>filename</i> <code>lbfluid save_binary_checkpoint</code> <i>filename</i> <code>lbfluid load_ascii_checkpoint</code> <i>filename</i> <code>lbfluid load_-binary_checkpoint</code> <i>filename</i>	128
<code>thermostat lb</code> <sup>1 or 2</sup> <i>T</i>	128
Required features: <sup>1</sup> LB <sup>2</sup> LB_GPU	
<code>lbnode</code> <i>x y z</i> ( <code>print</code>   <code>set</code> ) <i>args</i>	128
Required features: LB	
<code>lbboundary shape</code> <i>shape_args</i> [ <i>velocity vx vy vz</i> ]	129
Required features: LB_BOUNDARIES	
<code>lbboundary force</code> [ <i>n<sub>boundary</sub></i> ]	129
Required features: LB_BOUNDARIES	
<code>lbfluid cpu</code>	130
<code>lbfluid gpu</code>	
Required features: <sup>1</sup> LB <sup>2</sup> LB_GPU	
<code>setmd mu_E</code> $\mu E_x$ $\mu E_y$ $\mu E_z$	137
Required features: LB LB_ELECTROHYDRODYNAMICS	
<code>system_com</code>	138
<code>system_com_vel</code>	138
<code>galileiTransformParticles</code>	138
<code>countBonds</code> <i>particle<sub>i</sub>list</i>	139
<code>findPropPos</code> <i>particle<sub>p</sub>property<sub>i</sub>list property</i>	139
<code>findBondPos</code> <i>particle<sub>p</sub>property<sub>i</sub>list</i>	140
<code>timeStamp</code> <i>path prefix postfix suffix</i>	141

## 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.h` (see section 3.4 on page 24). To activate `FEATURE`, add the following line to the header file:

```
#define FEATURE
```

### B.1. General features

- **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 68) 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 5.6 on page 55 for details on these algorithms.
- **DIPOLLES** This activates the dipole-moment property of particles; In addition, the various magnetostatics algorithms, such as P3M are switched on. See section 5.6 on page 55 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 26 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.
- **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 36 for possible constraint forms.

- **TUNABLE\_SLIP** Switch on tunable slip conditions for planar wall boundary conditions. See section 5.8.1 on page 65 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**
- **BOND\_CONSTRAINT** Turns on the RATTLE integrator which allows for fixed lengths bonds between particles.
- **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**
- **OVERLAPPED**
- **COLLISION\_DETECTION** Allows particles to be bound oo 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*.

In addition, there are switches that enable additional features in the integrator or thermostat:

- NEMD Enables the non-equilibrium (shear) MD support (see section 6.3 on page 73).
- NPT Enables an on-the-fly NPT integration scheme (see section 6.2.3 on page 73).
- DPD Enables the dissipative particle dynamics thermostat (see section 6.2.2 on page 71).
- TRANS\_DPD Enables the transversal dissipative particle dynamics thermostat (see section 6.2.2 on page 72).
- INTER\_DPD Enables the dissipative particle dynamics thermostat implemented as an interaction, allowing to choose different parameters between different particle types (see section 6.2.2 on page 72).
- INTER\_RF
- 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 137).
- LB\_ELECTROHYDRODYNAMICS Enables the implicit calculation of electro-hydrodynamics for charged particles and salt ions in an electric field.

## B.2. Interactions

The following switches turn on various short ranged interactions (see section 5.1 on page 42):

- 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 bondangle potentials, you currently need to choose the type by the feature (see section 5.4 on page 53). This will change in the near future to three independent angle potentials:

- `BOND_ANGLE_HARMONIC`
- `BOND_ANGLE_COSINE`
- `BOND_ANGLE_COSSQUARE`
- `BOND_ANGLEDIST`
- `BOND_ENDANGLEDIST`

### 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<sup>3</sup>Ms real space part.
- **ESK\_DEBUG** debugging of P<sup>3</sup>Ms  $k$ -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.

## 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.

**kremerGrest.tcl** This reproduces the data of Kremer and Grest [29]: Multiple systems with different number of neutral polymer chains of various lengths are simulated for very long times at melt density 0.85 while their static and some dynamic properties are measured. Shows the advanced features of ESPResSo: How to run several simulations from a single script. How to use online-analysis (The analyze command) with comparison to expectation values. How to get averages of the observables. How to set/restore checkpoints (Using Checkpoints, saving configurations) including auto-detection of previously derived parts of the simulation(s). How to create gnuplots from within the script and combine multiple plots onto duplex pages (Statistical Analysis and Creating Gnuplots). In the end the script will provide plots of all important quantities as .ps- and .pdf-files while compressing the data-files. Note however, that the simulation uses the original time scale, hence it may take quite some time to finish.

**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 26, namely the capability to fix a particle in space and to apply an external force.

**watch.tcl** Script to visualize any of your productions. Use the **-h** option when calling it to see how it works.

## 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 [32, 36]. 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

$$\dot{\vec{r}}_i = \frac{1}{m_i} \vec{p}_i \quad (\text{D.1})$$

$$\dot{\vec{p}}_i = -\frac{\partial U}{\partial \vec{r}_i} + q_i \vec{E}(\vec{r}_i) - \frac{\zeta}{m_i} \vec{p}_i + \vec{f}_i \quad (\text{D.2})$$

$$\dot{\vec{A}} = -\vec{E} \quad (\text{D.3})$$

$$\dot{\vec{E}} = c^2 \vec{\nabla} \times (\vec{\nabla} \times \vec{A}) - \frac{1}{\epsilon_0} \vec{j}, \quad (\text{D.4})$$

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:

$$\langle f_i^\alpha(t) f_j^\beta(t') \rangle = 2\zeta k_B T \delta_{ij} \delta_{\alpha\beta} \delta(t - t'), \quad (\text{D.5})$$

where  $\alpha$  and  $\beta$  denote Cartesian indices.

If we introduce the vector  $\vec{B} = \nabla \times \vec{A}$  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.

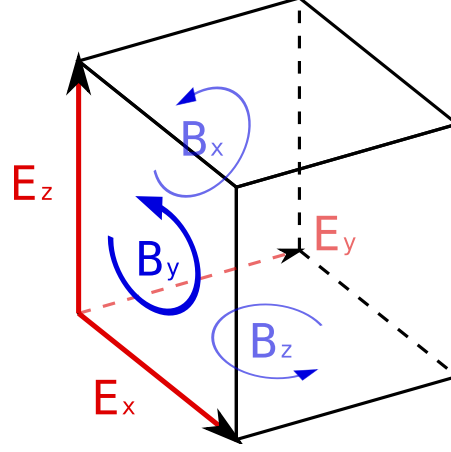


Figure D.1.: Spatial elements of a cell complex

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

$$q_{\text{plane}} = \frac{1}{N_z} \sum_i q(\vec{r}_i) \delta(z_i - z_{\text{plane}}), \quad (\text{D.6})$$

$N_z$  is the number of charges in plane  $z = z_{\text{plane}}$ . Update the  $z$ -field according to the formula

$$E_z^2 = E_z^1 + \frac{q_{\text{plane}}}{\epsilon_0 a^2}; \quad (\text{D.7})$$

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

$$q_{\text{wire}} = \frac{1}{N_y} \sum_i q(\vec{r}_i) \delta(z_i - z_{\text{plane}}) \delta(y_i - y_{\text{wire}}), \quad (\text{D.8})$$

$N_y$  now meaning the number of charges in the wire. Update  $y$ -field

$$E_y^2 = E_y^1 + \frac{q_{\text{wire}}}{\epsilon_0 a^2}; \quad (\text{D.9})$$

3. Subtract the charge  $q_{\text{wire}}$  from the each charge on the sites of  $(y_{\text{wire}}, z_{\text{plane}})$ . Update  $x$  field

$$E_x^2 = E_x^1 + \frac{q_{\text{vertex}}}{\epsilon_0 a^2} \quad (\text{D.10})$$

This scheme is repeated until the fields are completely relaxed (i. e. the energy is minimized). During repetition, the spatial dimensions are permuted to avoid a drift in one direction.

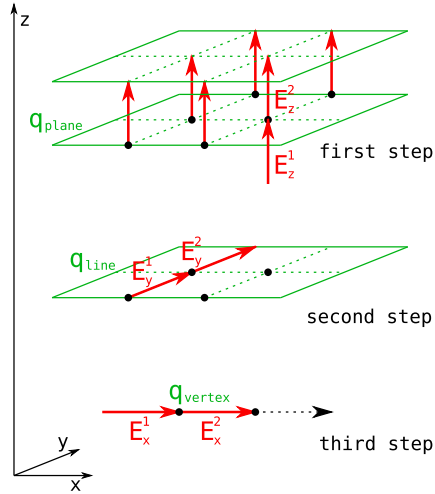


Figure D.2.: Recursive solution of Gauss' law

## D.4. Time integrator

For the time discretization we have adopted the elegant solution which was found by Rottler and Maggs [32] 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).

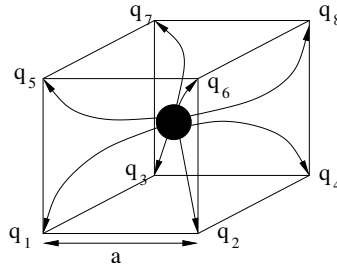


Figure D.3.: Linear interpolation scheme



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

$$\alpha_{ij} = \frac{1}{4a\epsilon_0 L^3} \sum_{\vec{k}} \frac{\cos \vec{k}(\vec{R}_i - \vec{R}_j)}{\sum_{i=1}^3 (1 - \cos \vec{k}\vec{a}_i)} \quad (\text{D.11})$$

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

$$\vec{F}_{self} = -\frac{\partial \mathcal{U}_{self}}{\partial \vec{r}} = -\sum_i \sum_j \alpha_{ij} \left[ q_i \frac{\partial q_j}{\partial \vec{r}} + q_j \frac{\partial q_i}{\partial \vec{r}} \right]. \quad (\text{D.12})$$

## 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 highly 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

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  $\beta \rightarrow 0$  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 [45]. 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

$$\tilde{E} = \frac{1}{2} \lim_{\beta \rightarrow 0} \sum_{k,l,m} \sum_{i,j=1}^N \frac{q_i q_j e^{-\beta|p_{ij}+n_{klm}|}}{|p_{ij} + n_{klm}|} =: \frac{1}{2} \lim_{\beta \rightarrow 0} \sum_{i,j=1}^N q_i q_j \phi_\beta(x_{ij}, y_{ij}, z_{ij}).$$

$\phi_\beta$  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[45]. 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 [48]. 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

$$\phi(x, y, z) = u_x u_y \sum_{p, q \neq 0} \frac{e^{2\pi f_{pq} z} + e^{2\pi f_{pq}(\lambda_z - z)}}{f_{pq} (e^{2\pi f_{pq} \lambda_z} - 1)} e^{2\pi i u_y q y} e^{2\pi i u_x p x} + 2\pi u_x u_y \left( u_z z^2 - z + \frac{\lambda_z}{6} \right).$$

where  $\lambda_{x,y,z}$  are the box dimensions,  $f_{pq} = \sqrt{(u_x p)^2 + (u_y q)^2}$ ,  $f_p = u_x p$ ,  $f_q = u_y q$ ,  $\omega_p = 2\pi u_x p$  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

$$\begin{aligned} \tilde{\phi}(x, y, z) = & 2u_x u_y \sum_{p, q > 0} \frac{\cosh(2\pi f_{pq} z)}{f_{pq} (e^{2\pi f_{pq} \lambda_z} - 1)} e^{2\pi i u_y q y} e^{2\pi i u_x p x} + \\ & 4u_x \sum_{l, p > 0} (K_0(2\pi u_x p \rho_l) + K_N(2\pi u_x p \rho_{-l})) \cos(2\pi u_x p x) - \\ & 2u_x \sum_{n \geq 1} \frac{b_{2n}}{2n(2n)!} \Re((2\pi u_y (z + iy))^{2n}) + \\ & u_x \sum_{n \geq 0} \binom{-\frac{1}{2}}{n} \frac{(\psi^{(2n)}(1+u_x x) + \psi^{(2n)}(1-u_x x))}{(2n)!} \rho^{2n} - \\ & 2 \log(4\pi). \end{aligned}$$

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

$$\tilde{\phi}(0, 0, 0) = 2u_x u_y \sum_{p, q > 0} \frac{1}{f_{pq} (e^{2\pi f_{pq} \lambda_z} - 1)} + 8u_x \sum_{l, p > 0} K_N(2\pi u_x \lambda_y p l) + 2u_x \psi^{(0)}(1) - 2 \log(4\pi).$$

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 [48]. 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

$$\phi(x, y, z) = 4u_x u_y \sum_{p,q>0} \frac{e^{-2\pi f_{pq}|z|}}{f_{pq}} \cos(\omega_p x) \cos(\omega_q y) + \\ 2u_x u_y \left( \sum_{q>0} \frac{e^{-2\pi f_q|z|}}{f_q} \cos(\omega_q y) + \sum_{p>0} \frac{e^{-2\pi f_p|z|}}{f_p} \cos(\omega_p x) \right) - \\ 2\pi u_x u_y |z|$$

and the near formula is

$$\tilde{\phi}(x, y, z) = 4u_x \sum_{l,p>0} (K_0(\omega_p \rho_l) + K_0(\omega_p \rho_{-l})) \cos(\omega_p x) - \\ 2u_x \sum_{n \geq 1} \frac{b_{2n}}{2n(2n)!} \Re((2\pi u_y(z + iy))^{2n}) + \sum_{k=1}^{N_\psi-1} \left( \frac{1}{r_k} + \frac{1}{r_{-k}} \right) - \\ u_x \sum_{n \geq 0} \binom{-\frac{1}{2}}{n} \frac{(\psi^{(2n)}(N_\psi + u_x x) + \psi^{(2n)}(N_\psi - u_x x))}{(2n)!} (u_x \rho)^{2n} - \\ 2u_x \log \left( 4\pi \frac{u_y}{u_x} \right).$$

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 [4].

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, ..., S-2, and in layers S+2, ..., 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

$$\sum_{j \in I_S, S < S_i-1} q_i q_j \frac{e^{-2\pi f_{pq}|z_i - z_j|}}{f_{pq}} \cos(\omega_p(x_i - x_j)) \cos(\omega_q(y_i - y_j)) = \\ q_i \frac{e^{-2\pi f_{pq}z_i}}{f_{pq}} \cos(\omega_p x_i) \cos(\omega_q y_i) \sum_{j \in I_S, S < S_i-1} q_j e^{2\pi f_{pq}z_j} \cos(\omega_p x_j) \cos(\omega_q y_j) + \\ q_i \frac{e^{-2\pi f_{pq}z_i}}{f_{pq}} \cos(\omega_p x_i) \sin(\omega_q y_i) \sum_{j \in I_S, S < S_i-1} q_j e^{2\pi f_{pq}z_j} \cos(\omega_p x_j) \sin(\omega_q y_j) + \\ q_i \frac{e^{-2\pi f_{pq}z_i}}{f_{pq}} \sin(\omega_p x_i) \cos(\omega_q y_i) \sum_{j \in I_S, S < S_i-1} q_j e^{2\pi f_{pq}z_j} \sin(\omega_p x_j) \cos(\omega_q y_j) + \\ q_i \frac{e^{-2\pi f_{pq}z_i}}{f_{pq}} \sin(\omega_p x_i) \sin(\omega_q y_i) \sum_{j \in I_S, S < S_i-1} q_j e^{2\pi f_{pq}z_j} \sin(\omega_p x_j) \sin(\omega_q y_j).$$

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 communication 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 resp. energy contributions for its particles.

In pseudo code, the far formula algorithm looks like:

1. for each layer  $s = 1, \dots, S$ 
  - 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.  $\Xi_s^{(\pm, s/c, s/c)} \pm = \xi_j^{(\pm, s/c, s/c)}$
2.  $\Xi_3^{(l, s/c, s/c)} = \Xi_1^{(+, s/c, s/c)}$
3. for each layer  $s = 4, \dots, S$

- 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  $s = (S-3), \dots, 1$ 
  - 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  $s = 1, \dots, S$ 
  - 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 [4, 5], Arnold et al. [6, 7].

### 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. [52].

## E.3. MMM1D

In one dimensionally periodic systems with  $z$  being the periodic coordinate, the far formula looks like

$$\begin{aligned}
\phi(\rho, z) &= 4u_z \sum_{p \neq 0} K_0(\omega\rho) \cos(\omega z) - 2u_z \log\left(\frac{\rho}{2\lambda_z}\right) - 2u_z \gamma \\
F_\rho(\rho, z) &= 8\pi u_z^2 \sum_{p \neq 0} p K_1(\omega\rho) \cos(\omega z) + \frac{2u_z}{\rho} \\
F_z(\rho, z) &= 8\pi u_z^2 \sum_{p \neq 0} p K_0(\omega\rho) \sin(\omega z),
\end{aligned}$$

the near formula is

$$\begin{aligned}
\tilde{\phi}(\rho, z) &= -u_z \sum_{n \geq 0} \left( \frac{-\frac{1}{2}}{n} \right) \frac{(\psi^{(2n)}(N_\psi + u_z z) + \psi^{(2n)}(N_\psi - u_z z))}{(2n)!} (u_z \rho)^{2n} - 2u_z \gamma + \\
&\quad \sum_{k=1}^{N_\psi-1} \left( \frac{1}{r_k} + \frac{1}{r_{-k}} \right) \\
\tilde{F}_\rho(\rho, z) &= -u_z^3 \sum_{n \geq 0} \left( \frac{-\frac{1}{2}}{n} \right) \frac{(\psi^{(2n)}(N_\psi + u_z z) + \psi^{(2n)}(N_\psi - u_z z))}{(2n)!} (u_z \rho)^{2n-1} + \\
&\quad \sum_{k=1}^{N_\psi-1} \left( \frac{\rho}{r_k^3} + \frac{\rho}{r_{-k}^3} \right) \\
\tilde{F}_z(\rho, z) &= -u_z^2 \sum_{n \geq 0} \left( \frac{-\frac{1}{2}}{n} \right) \frac{(\psi^{(2n+1)}(N_\psi + u_z z) + \psi^{(2n+1)}(N_\psi - u_z z))}{(2n)!} (u_z \rho)^{2n} + \\
&\quad \sum_{k=1}^{N_\psi-1} \left( \frac{z+k\lambda_z}{r_k^3} + \frac{z-k\lambda_z}{r_{-k}^3} \right),
\end{aligned}$$

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  $\pm n\lambda_z$ ,  $n \neq 0$  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 [45]. 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

$$\begin{aligned} \psi(x, y, z) = & 4u_x u_y \sum_{p,q>0} \frac{\cosh(2\pi f_{pq} z)}{f_{pq}(e^{2\pi f_{pq} \lambda_z} - 1)} \cos(\omega_p x) \cos(\omega_q y) + \\ & 2u_x u_y \sum_{p>0} \frac{\cosh(2\pi f_p z)}{f_p(e^{2\pi f_p \lambda_z} - 1)} \cos(\omega_p x) + \\ & 2u_x u_y \sum_{q>0} \frac{\cosh(2\pi f_q z)}{f_q(e^{2\pi f_q \lambda_z} - 1)} \cos(\omega_q y). \end{aligned}$$

The implementation is very similar to MMM2d, except that the separation between slices closely, 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, 7].

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.

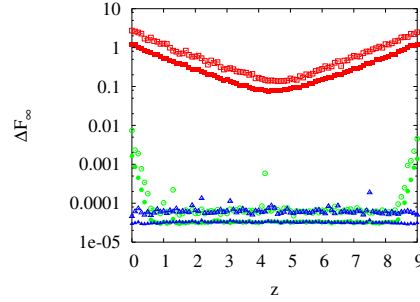


Figure E.1.: Error distribution of the ELC method.

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. 6.2.3
- [2] Hans C. Andersen. Rattle: A ”velocity” version of the shake algorithm for molecular dynamics calculations. *J. Comp. Phys.*, 51:24–34, 1983. 4.2.1, 5.3.4
- [3] A. Arnold and C. Holm. MMM1D: A method for calculating electrostatic interactions in 1D periodic geometries. *J. Chem. Phys.*, 123(12):144103, September 2005. E.3
- [4] 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, 1 November 2002. E.2, E.2
- [5] Axel Arnold and Christian Holm. A novel method for calculating electrostatic interactions in 2D periodic slab geometries. *Chem. Phys. Lett.*, 354:324–330, 2002. E.2
- [6] Axel Arnold, Jason de Joannis, and Christian Holm. Electrostatics in Periodic Slab Geometries I. *J. Chem. Phys.*, 117:2496–2502, 2002. 5.6.6, E.2, E.5
- [7] Axel Arnold, Jason de Joannis, and Christian Holm. Electrostatics in Periodic Slab Geometries II. *J. Chem. Phys.*, 117:2503–2512, 2002. E.2, E.5
- [8] 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. 5.6.7
- [9] 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>. 5.6.6
- [10] H. J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren, A. DiNola, and J. R. Haak. Molecular dynamics with coupling to a heat bath. *J. Chem. Phys.*, 81:3684–3690, 1984. 6.2.1

- [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. 5.7.2
- [12] Juan J. Cerda, V. Ballenegger, O. Lenz, and C. Holm. P3M algorithm for dipolar interactions. *J. Chem. Phys.*, 129:234104, 2008. 5.6.1, 5.7.1, 5.7.1
- [13] M. Deserno. *Counterion condensation for rigid linear polyelectrolytes*. PhD thesis, Universität Mainz, 2000. 5.6.1, 5.7.1
- [14] M. Deserno and C. Holm. How to mesh up Ewald sums. i. *J. Chem. Phys.*, 109:7678, 1998. 5.6.1, 5.7.1
- [15] M. Deserno and C. Holm. How to mesh up Ewald sums. ii. *J. Chem. Phys.*, 109:7694, 1998. 5.6.1, 5.7.1
- [16] M. Deserno, C. Holm, and H. J. Limbach. *Molecular Dynamics on Parallel Computers*, chapter How to mesh up Ewald sums. World Scientific, Singapore, 2000. 5.6.1, 5.7.1
- [17] M Doi and S F Edwards. *The theory of polymer dynamics*. Oxford Science Publications, 1986. 8.2.1
- [18] P.P. Ewald. Die berechnung optischer und elektrostatischer gitterpotentiale. *Ann. Phys.*, 64:253–287, 1921. 5.6.1, 5.7.1
- [19] Daan Frenkel and Berend Smit. *Understanding Molecular Simulation*. Academic Press, San Diego, second edition, 2002. 1.1, 9.2.6
- [20] 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. 2.3, 6.2.1
- [21] 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}. 12.6
- [22] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. IOP, 1988. 5.6.1, 5.6.1, 5.7.1
- [23] F. Höfling, Karl-Ulrich Bamberg, and Thomas Franosch. Anomalous transport resolved in space and time by fluorescence correlation spectroscopy. *Soft Matter*, 7:1358, 2011. 9.2.2
- [24] W. Humphrey, A. Dalke, and K. Schulten. VMD: Visual molecular dynamics. *J. Mol. Graphics*, 14:33–38, 1996. 10.3
- [25] C. Junghans and S. Poblete. A reference implementation of the adaptive resolution scheme in ESPResSo. *Comp. Phys. Comm.*, 181(8):1449–1454, 2010. 6.5

- [26] C. Junghans, M. Praprotnik, and K. Kremer. Transport properties controlled by a thermostat: An extended dissipative particle dynamics thermostat. *Soft Matter*, 4: 156, 2008. 6.2.2
- [27] 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>. jce:titleComputer Physics Communications Special Edition for Conference on Computational Physics Kaohsiung, Taiwan, Dec 15-19, 2009j/ce:title. 5.6.7
- [28] Jiri Kolafa and John W. Perram. Cutoff errors in the ewald summation formulae for point charge systems. *Molecular Simulation*, 9(5):351–368, 1992. 5.6.1, 5.6.1, 5.7.1
- [29] K. Kremer and G. S. Grest. Dynamics of entangled linear polymer melts: A molecular-dynamics simulation. *J. Chem. Phys.*, 92:5057, 1990. C
- [30] H. J. Limbach and C. Holm. Single-chain properties of polyelectrolytes in poor solvent. *J. Phys. Chem. B*, 107(32):8041–8055, 2003. 8.1.18
- [31] 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}. 9.2.6
- [32] A. C. Maggs and V. Rosseto. Local simulation algorithms for coulombic interactions. *Phys. Rev. Lett.*, 88:196402, 2002. 5.6.5, D, D.4
- [33] Bernward A. Mann. *The Swelling Behaviour of Polyelectrolyte Networks*. PhD thesis, Johannes Gutenberg-University, Mainz, Germany, December 2005. 6.2.3
- [34] 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. 7.7
- [35] 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. 6.2.2
- [36] Igor Pasichnyk and Burkhard Dünweg. Coulomb interactions via local dynamics: A molecular-dynamics algorithm. *J. Phys.: Condens. Matter*, 16(38):3999–4020, September 2004. 5.6.5, D
- [37] S. Poblete, M. Praprotnik, K. Kremer, and Luigi Delle Site. Coupling different levels of resolution in molecular simulations. *Jour. Chem. Phys.*, 132(11):114101, 2010. 6.5

- [38] Matej Praprotnik, Luigi Delle Site, and Kurt Kremer. Adaptive resolution molecular-dynamics simulation: Changing the degrees of freedom on the fly. *The Journal of Chemical Physics*, 123(22):224106–14, 2005. 6.5
- [39] Matej Praprotnik, Luigi Delle Site, and Kurt Kremer. Multiscale simulation of soft matter: From scale bridging to adaptive resolution. *Annual Review of Physical Chemistry*, 59(1):545–571, 2008. 6.5
- [40] 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}. 9.2.2, 9.2.6, 9.2.6
- [41] Michael Rubinstein and Ralph H. Colby. *Polymer Physics*. Oxford University Press, Oxford, UK, 2003. 8.2.1
- [42] 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}. 9.2.6
- [43] 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>. 8.1.19
- [44] 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. 5.8.1
- [45] E. R. Smith. Electrostatic energy in ionic crystals. *Proc. R. Soc. Lond. A*, 375: 475–505, 1981. E.1, E.4
- [46] T. Soddemann, B. Dünweg, and K. Kremer. A generic computer model for amphiphilic systems. *Eur. Phys. J. E*, 6:409, 2001. 5.1.4, 6.3
- [47] 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. 6.2.2
- [48] R. Strebel. *Pieces of software for the Coulombic  $n$  body problem*. Dissertation, ETH Zürich, 1999. URL <http://e-collection.ethbib.ethz.ch/show?type=diss&nr=13504>. E.1
- [49] S. Succi. *The lattice Boltzmann equation for fluid dynamics and beyond*. Oxford University Press, USA, 2001. 12.4
- [50] 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. 8.1.21

- [51] C. Tyagi, M. Sützen, 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. 5.6.7
- [52] 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. E.2.1
- [53] 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. 5.6.6
- [54] Ulli Wolff. Monte carlo errors with less errors. *Comput. Phys. Commun.*, 156:143–153, 2004. 8.4, 9.2.5

# Index

- aggregation, **92**
- analysis, 87
  - aggregation, **92**
  - bond distances internal first monomer, **99**
  - bond lengths, **100**
  - center of mass, **91**
  - chains, **98**
  - end-to-end distance of a chain, **98**
  - energies, **94**
  - finding holes, **93**
  - form factor of a chain, **100**
  - gyration tensor, **92**
  - hydrodynamic radius of a chain, **98**
  - internal distances within a chain, **99**
  - local stress tensor, **96**
  - minimal particle distance, **87**
  - moment of inertia matrix, **91**
  - particle distance, **87**
  - particle distribution, **88**
  - particles in the neighbourhood, **88**
  - pearl-necklace structures, **92**
  - pressure, **94**
  - principal axis of the moment of inertia, **91**
  - radial distribution function, **100**
  - radial distribution function  $g(r)$ , **90**
  - radius of gyration of a chain, **98**
  - stress tensor, **95**
  - structure factor  $S(q)$ , **90**
  - topologies, **97**
  - van Hove autocorrelation function  $G(r, t)$ , **91**
- Analysis in the Core, 104
- analyze** (Tcl-command), **87**
- Anisotropic interactions, 47
- blockfile** (Tcl-command), **115**
- blocks, 117
- BMHTF interaction, **45**
- bond distances internal first monomer, **99**
- bond lengths, **100**
- bond-angle interactions, **53**
- bonded interaction type id, 50
- bonded interactions, **50**
- box\_1** (global variable), **68**
- Buckingham interaction, **46**
- build directory, 21
- cell\_grid** (global variable), **68**
- cell\_size** (global variable), **68**
- cellsystem** (Tcl-command), **74**
- center of mass, **91**
- chains, **98**
- change\_volume** (Tcl-command), **79**
- checkpoint\_read** (Tcl-command), **119**
- checkpoint\_set** (Tcl-command), **119**
- configuration header, 24
- configure, 12, 20
- configure options, 21
- constraint** (Tcl-command), **36**
- copy\_particles** (Tcl-command), **35**
- correlation** (Tcl-command), **109**
- Correlations, 108
- Coulomb interactions, **55**
- counterions** (Tcl-command), **31**
- crosslink** (Tcl-command), **35**
- DAWAANR method, **64**
- Debye-Hückel potential, **57**
- diamond** (Tcl-command), **32**
- dielectric** (Tcl-command), **62**



Dielectric interfaces, 61, 62  
 dihedral interactions, 54  
 Dipolar interactions, 62  
 Directional Lennard-Jones interaction, 47  
 DLC method, 64  
 domain decomposition, 74  
 DPD, 66, 71  
 DPD interaction, 66  
 dpd\_gamma (global variable), 68  
 dpd\_r\_cut (global variable), 68  
  
 ELC method, 60  
 Electrostatic interactions, 55  
 end-to-end distance of a chain, 98  
 energies, 94  
 energy unit, 9  
  
 features, 20, 24, 181  
     ADDITIONAL\_CHECKS, 184  
     ADRESS, 182  
     ASYNCH\_BARRIER, 186  
     BMHTF\_NACL, 184  
     BOND\_ANGLE\_COSINE, 184  
     BOND\_ANGLE\_COSSQUARE, 184  
     BOND\_ANGLE\_HARMONIC, 184  
     BOND\_ANGLEDIST, 184  
     BOND\_CONSTRAINT, 182  
     BOND\_ENDANGLEDIST, 184  
     BOND\_VIRTUAL, 182  
     BUCKINGHAM, 184  
     CELL\_DEBUG, 185  
     COLLISION\_DETECTION, 182  
     COMFIXED, 182  
     COMFORCE, 182  
     COMM\_DEBUG, 185  
     CONSTRAINTS, 181  
     DIPOLES, 181  
     DPD, 183  
     DPD\_MASS\_LIN, 183  
     DPD\_MASS\_RED, 183  
     ELECTROSTATICS, 181  
     ESK\_DEBUG, 185  
     ESR\_DEBUG, 185  
     EVENT\_DEBUG, 185  
     EWALD\_DEBUG, 185  
     EXCLUSIONS, 182  
     EXTERNAL\_FORCES, 181  
     FENE\_DEBUG, 186  
     FFT\_DEBUG, 185  
     FORCE\_CORE, 186  
     FORCE\_DEBUG, 185  
     GAY\_BERNE, 183  
     GHOST\_DEBUG, 185  
     GHOST\_FORCE\_DEBUG, 185  
     GRID\_DEBUG, 185  
     HALO\_DEBUG, 185  
     HERTZIAN, 183  
     INTEG\_DEBUG, 185  
     INTER\_DPD, 183  
     INTER\_RF, 183  
     LATTICE\_DEBUG, 185  
     LB, 183  
     LB\_DEBUG, 186  
     LB\_ELECTROHYDRODYNAMICS, 183  
     LENNARD\_JONES, 183  
     LENNARD\_JONES\_GENERIC, 183  
     LJ\_ANGLE, 183  
     LJ\_DEBUG, 186  
     LJ\_WARN\_WHEN\_CLOSE, 184  
     LJCOS, 183  
     LJCOS2, 183  
     MAGGS\_DEBUG, 185  
     MASS, 182  
     MEM\_DEBUG, 185  
     METADYNAMICS, 182  
     MODES, 182  
     MOL\_CUT, 184  
     MOLFORCES, 182  
     MOLFORCES\_DEBUG, 186  
     MORSE, 184  
     MORSE\_DEBUG, 186  
     MPI\_CORE, 186  
     NEMD, 183  
     NO\_INTRA\_NB, 184  
     NPT, 183  
     OLD\_RW\_VERSION, 182  
     OLD\_DIHEDRAL, 184

ONEPART\_DEBUG, 186  
 OVERLAPPED, 182  
 P3M\_DEBUG, 185  
 PARTIAL\_PERIODIC, 181  
 PARTICLE\_DEBUG, 185  
 POLY\_DEBUG, 186  
 PTENSOR\_DEBUG, 185  
 RANDOM\_DEBUG, 185  
 ROTATION, 181  
 ROTATIONAL\_INERTIA, 181  
 SMOOTH\_STEP, 184  
 SOFT\_SPHERE, 184  
 STAT\_DEBUG, 186  
 TABULATED, 183  
 THERMO\_DEBUG, 186  
 THERMOSTAT\_IGNORE\_NON\_VIRTUAL, 182  
 TRANS\_DPD, 183  
 TUNABLE\_SLIP, 182  
 VERLET\_DEBUG, 185  
 VIRTUAL\_SITES.COM, 182  
 VIRTUAL\_SITES.DEBUG, 186  
 VIRTUAL\_SITES.NO\_VELOCITY, 182  
 VIRTUAL\_SITES.RELATIVE, 182  
 VIRTUAL\_SITES.THERMOSTAT, 182  
 FENE bond, 50  
 FFTW, 10  
 finding holes, 93  
 form factor of a chain, 100  
 gamma (global variable), 68  
 Gay-Berne interaction, 49  
 Generic Lennard-Jones interaction, 44  
 global variables, 115  
     box\_l, 68  
     cell\_grid, 68  
     cell\_size, 68  
     dpd\_gamma, 68  
     dpd\_r\_cut, 68  
     gamma, 68  
     integ\_switch, 68  
     local\_box\_l, 68  
     max\_cut\_bonded, 68  
     max\_cut\_nonbonded, 68  
     max\_cut, 68  
     max\_num\_cells, 68  
     max\_part, 68  
     max\_range, 68  
     max\_skin, 69  
     min\_global\_cut, 69  
     min\_num\_cells, 69  
     n\_layers, 69  
     n\_nodes, 69  
     n\_part\_types, 69  
     n\_part, 69  
     node\_grid, 69  
     npt\_p\_ext, 69  
     npt\_p\_inst, 69  
     nptiso\_gamma0, 69  
     nptiso\_gammav, 69  
     periodicity, 69  
     piston, 69  
     skin, 69  
     temperature, 69  
     thermo\_switch, 69  
     time\_step, 69  
     time, 69  
     timings, 69  
     transfer\_rate, 69  
     verlet\_flag, 70  
     verlet\_reuse, 70  
 gyration tensor, 92  
 harmonic bond, 51  
 Hertzian interaction, 47  
 hydrodynamic radius of a chain, 98  
 ICC\*, 61  
 iccp3m (Tcl-command), 61  
 icosaeder (Tcl-command), 34  
 IMD, 125  
 imd (Tcl-command), 125  
 Installation, 20  
 integ\_switch (global variable), 68  
 integrate (Tcl-command), 79  
 inter (Tcl-command), 42  
 Interaction DPD, 72

interactions, **42**  
  BMHTF, **45**  
  bond-angle, **53**  
  bonded, **50**  
  Buckingham, **46**  
  Coulomb, **55**  
  DAWAANR method, **64**  
  Debye-Hückel, **57**  
  dihedral, **54**  
  Dipolar, **62**  
  Directional Lennard-Jones, **47**  
  DLC method, **64**  
  DPD, **66**  
  ELC method, **60**  
  Electrostatic, **55**  
  FENE, **50**  
  Gay-Berne, **49**  
  Generic Lennard-Jones, **44**  
  harmonic, **51**  
  hertzian, **47**  
  Lennard-Jones, **43**  
  Lennard-Jones cosine, **44**  
  Maggs method, **59**  
  Magnetostatic, **62**  
  MDDS method, **64**  
  MEMD, **59**  
  MMM1D, **58**  
  MMM2D, **58**  
  Morse, **46**  
  non-bonded, **42**  
  P3M, **55**  
  rigid bond, **51**  
  smooth-step, **45**  
  soft-sphere, **47**  
  subtracted Lennard-Jones, **51**  
  tabulated, **43**  
  tabulated bond, **52**  
  Tunable-slip boundary interactions, **65**  
interactive mode, **24**  
internal distances within a chain, **99**  
**invalidate\_system** (Tcl-command), **80**  
label:DPDthermostat, **71**  
**lb** (Tcl-command), **137**  
length unit, **9**  
Lennard-Jones cosine interaction, **44**  
Lennard-Jones interaction, **43**  
local stress tensor, **96**  
**local\_box\_1** (global variable), **68**  
Maggs method, **59**  
Magnetostatic interactions, **62**  
make, **12**  
**max\_cut** (global variable), **68**  
**max\_cut\_bonded** (global variable), **68**  
**max\_cut\_nonbonded** (global variable), **68**  
**max\_num\_cells** (global variable), **68**  
**max\_part** (global variable), **68**  
**max\_range** (global variable), **68**  
**max\_skin** (global variable), **69**  
Maxwell Equation Molecular Dynamics, **59**  
MDDS method, **64**  
MEMD, **59**  
**metadynamics** (Tcl-command), **85**  
**min\_global\_cut** (global variable), **69**  
**min\_num\_cells** (global variable), **69**  
minimal particle distance, **87**  
MMM1D method, **58**  
MMM2D method, **58**  
moment of inertia matrix, **91**  
momentum exchange method, **74**  
Morse interaction, **46**  
MPI, **10**  
Multiple tau correlator, **113**  
myconfig.h, **24**  
**n\_layers** (global variable), **69**  
**n\_nodes** (global variable), **69**  
**n\_part** (global variable), **69**  
**n\_part\_types** (global variable), **69**  
NEMD, **73**  
**nemd** (Tcl-command), **73**  
**node\_grid** (global variable), **69**  
Non-bonded interactions, **42**  
**npt\_p\_ext** (global variable), **69**  
**npt\_p\_inst** (global variable), **69**

**nptiso\_gamma0** (global variable), **69**  
**nptiso\_gammav** (global variable), **69**  
  
**observable** (Tcl-command), **104**  
**Observables**, **104**  
  
**P3M** method, **55**  
**parallel\_tempering** (Tcl-command), **81**  
**part** (Tcl-command), **26**  
particle distance, **87**  
particle distribution, **88**  
particles in the neighbourhood, **88**  
pearl-necklace structures, **92**  
**periodicity** (global variable), **69**  
physical units, **9**  
**piston** (global variable), **69**  
**polymer** (Tcl-command), **30**  
**prepare\_vmd\_connection** (Tcl-command),  
**126**  
**pressure**, **94**  
principal axis of the moment of inertia,  
**91**  
  
quick reference of Tcl-commands, **171**  
  
radial distribution function, **100**  
radial distribution function  $g(r)$ , **90**  
radius of gyration of a chain, **98**  
random number generators, **116**  
random seed, **116**  
Rattle Shake algorithm, **51**  
requirements, **10**  
rigid bond, **51**  
  
**salt** (Tcl-command), **32**  
**setmd** (Tcl-command), **68**  
shear-rate method, **74**  
**skin** (global variable), **69**  
smooth-step interaction, **45**  
soft-sphere interaction, **47**  
source directory, **21**  
**stop\_particles** (Tcl-command), **80**  
**stopParticles** (Tcl-command), **80**  
stored configurations, **101**, **117**  
stress tensor, **95**  
  
structure factor  $S(q)$ , **90**  
subtracted Lennard-Jones bond, **51**  
  
tabulated bond interactions, **52**  
tabulated interaction, **43**  
Tcl global variables, **116**  
Tcl-commands  
    **analyze**, **87**  
    **blockfile**, **115**  
    **cellsystem**, **74**  
    **change\_volume**, **79**  
    **checkpoint\_read**, **119**  
    **checkpoint\_set**, **119**  
    **constraint**, **36**  
    **copy\_particles**, **35**  
    **correlation**, **109**  
    **counterions**, **31**  
    **crosslink**, **35**  
    **diamond**, **32**  
    **dielectric**, **62**  
    **iccp3m**, **61**  
    **icosaeder**, **34**  
    **imd**, **125**  
    **integrate**, **79**  
    **inter**, **42**  
    **invalidate\_system**, **80**  
    **lb**, **137**  
    **metadynamics**, **85**  
    **nemd**, **73**  
    **observable**, **104**  
    **parallel\_tempering**, **81**  
    **part**, **26**  
    **polymer**, **30**  
    **prepare\_vmd\_connection**, **126**  
    **salt**, **32**  
    **setmd**, **68**  
    **stop\_particles**, **80**  
    **stopParticles**, **80**  
    **thermostat**, **70**  
    **uwerr**, **103**  
    **velocities**, **80**  
    **writedb**, **124**  
    **writedbfoldchains**, **124**  
    **writedbfoldtopo**, **124**

- writepsf, **124**
- writevcf, **123**
- writevsf, **122**
- Tcl/Tk, 10
- temperature (global variable), **69**
- thermo\_switch (global variable), **69**
- thermostat (Tcl-command), **70**
- time (global variable), **69**
- time unit, 9
- time\_step (global variable), **69**
- timings (global variable), **69**
- topologies, **97**
- transfer\_rate (global variable), **69**
- Tunable-slip boundary interaction, **65**
- units, 9
- uwerr (Tcl-command), **103**
- van Hove autocorrelation function  $G(r, t)$ ,  
**91**
- vcf, **121**
- velocities (Tcl-command), **80**
- verlet\_flag (global variable), **70**
- verlet\_reuse (global variable), **70**
- virtual sites, **38**
- vsf, **121**
- vtf, **121**
- whitespace, 115
- writepdb (Tcl-command), **124**
- writepdbfoldchains (Tcl-command), **124**
- writepdbfoldtopo (Tcl-command), **124**
- writepsf (Tcl-command), **124**
- writevcf (Tcl-command), **123**
- writevsf (Tcl-command), **122**