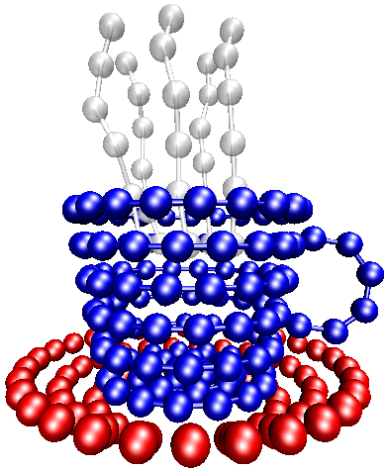


ESPResSo User's Guide

April 9, 2009

Contents

1 Introduction

Make the following lists full text.

- ESPResSo is a generic soft matter simulation packages
- for molecular dynamics simulations in soft matter research
- focussed on coarse-grained models
- employs modern algorithms (Lattice-Boltzmann, DPD, P3M, ...)
- written in C for maximal portability
- Tcl-controlled
- parallelized

1.1 Guiding principles

(from paper: 2.1 Goals and principles)
ESPResSo

- does *not* do the physics for you!
- requires you to understand what you do (can not be used as a black box)
- gives you maximal freedom (flexibility)
- is extensible
- integrates system setup, simulation and analysis, as this can't be strictly separated in soft matter simulations
- has no predefined units
- sets as few defaults as possible

1.2 Algorithms contained in ESPResSo

The following algorithms are implemented in ESPResSo:

- ensembles: NVE, NVT, NpT
- charged systems:

  - P3M for fully periodic systems
  - ELC and MMM-family of algorithms for charged systems with non-periodic boundary conditions
  - Maggs algorithm

- Hydrodynamics:
  - DPD (as a thermostat)
  - Lattice-Boltzmann

1.3 Basic program structure

(from paper: 2.2 Basic program structure)

- Control level: `Tcl`
- "Kernel" written in `C`
- This user's guide will focus on the control level

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 $\frac{kJ}{\text{mol}}$, 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 σ 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 σ 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 ϵ to the energy in $\frac{kJ}{\text{mol}}$. 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 [1] in the version 8.3 or later. Some example scripts will only work with Tcl 8.4. You do not only need the interpreter, but also the header files and libraries. Depending on the operating system, these may come in separate development packages. If you want to use a graphical user interface (GUI) for your simulation scripts, you will also need Tk.

FFTW In addition, ESPResSo needs the FFTW library [2] for Fourier transforms. ESPResSo can work with both the 2.1.x and 3.0.x series. Again, the header files are required.

MPI Finally, if you want to use ESPResSo in parallel, you need a working MPI environment (version 1.2). Currently, the following MPI implementations are supported:

- LAM/MPI is the preferred variant
- MPICH, which seems to be considerably slower than LAM/MPI in our benchmarks.
- On AIX systems, ESPResSo can also use the native POE parallel environment.
- On DEC/Compaq/HP OSF/Tru64, ESPResSo can also use the native dmpirun MPI environment.

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 decriptions:

- Different *variants* of a command are labelled (1), (2), ...
- Keywords and literals of the command that have to be typed exactly as given are written in `typewriter` font.
- If the command has variable arguments, they are set in *italicfont*. The description following the syntax definition should contain a detailed explanation of the argument and its type.
- (*alt1* | *alt2*) specifies, that one of the alternatives *alt1* or *alt2* can be used.
- [*argument*] specifies, that the arugment *argument* is optional, *i.e.* it can be omitted.
- When an optional argument or a whole command is marked by a superscript label ([1]), this denotes that the argument can only be used, when the corresponding feature (see appendix B on page 141) specified in "Required features" is activated.

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

Example

(1) `constraint wall normal` n_x n_y n_z `dist` d `type` id

(2) `constraint sphere center` c_x c_y c_z `radius` rad `direction` $direction$ `type` id

(3) `constraint rod center` c_x c_y `lambda` $lambda$ [1]

(4) `constraint ext_magn_field` f_x f_y f_z [2,3]

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

2 First steps

2.1 Quick installation

If you have the requirements (see section 1.5 on page 10) installed, in many cases, to compile ESPResSo, it is enough to execute the following sequence of two steps in the directory where you have unpacked the sources:

```
configure
make
```

Mention minimal configuration without myconfig.h

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 on page 19). Furthermore, many features of ESPResSo can be selectively turned on or off in the local configuration header of ESPResSo (see section 3.2 on page 20) 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 how to solve the problem. The configuration process can be controlled with the help of a number of options that are explained in section 3.3 on page 21.

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 the ESPResSo program for correctness.
- It can build the documentation.

The details of the usage of `make` are described in section 3.4 on page 22.

When these steps have successfully completed, ESPResSo can be started with the command (see section 3.5 on page 24)

```
Espresso
```

2.2 Running ESPResSo

[1]

ESPResSo is implemented as an extension to the Tcl script 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.

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

```
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
}
```

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

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 [7], and the charges interact via the Coulomb potential:

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

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

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

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

Now we can integrate the system:

[2] Note: there also exists a predefined tcl function *degrees_of_freedom* which does the same.

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

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

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}
        ...
}
```

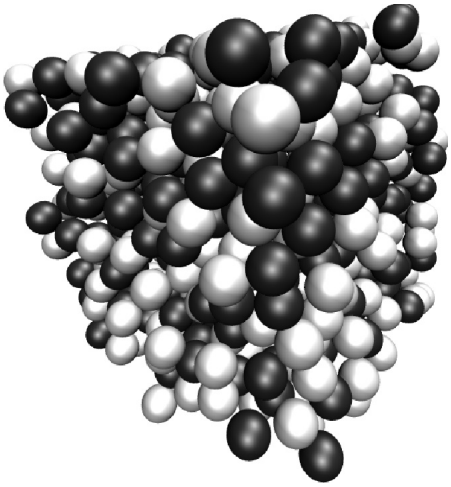


Figure 2.1: VMD Snapshot of the salt system

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` *$n_n$odes script* `[`*config*`... ]`

where *$n_n$odes* is the number of CPUs ESPResSo should be running on.

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

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

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

The code example given before is still quite simple, and the reader is encouraged to try to extend the example a little bit, e. g. by using differently sized particle, or changing the interactions. If something does not work, ESPResSo will give comprehensive error messages, which should make it easy to identify mistakes. For real simulations, the

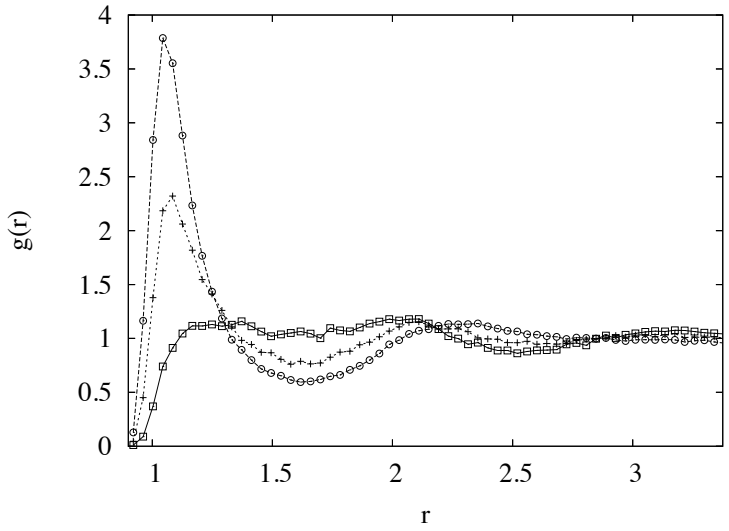


Figure 2.2: Radial distribution functions $g_{++}(r)$ between equal charges (rectangles) and $g_{+-}(r)$ for opposite charges (circles). The plus symbols denote $g(r)$ for an uncharged system.

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 Compiling and installing ESPResSo

- Compiling ESPResSo is a necessary evil
- Features can be compiled in or not
- For maximal efficiency, compile in only the features that you use
- ESPResSo can be obtained from the ESPResSo home page [1].
- If you are looking for the ESPResSo binary or the object files, read 3.1
- other than in most packages, ESPResSo will probably not be installed, or it will only be installed locally. Refer to 3.4.1 on page 23 for details.

3.1 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, 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. As the source directory is not modified during the compilation process, it is possible to compile more than one binary versions of ESPResSo from a single set of source files.

The location of the build directory is determined when `configure` is called. Depending on whether it is called from the source directory where it resides, or from some other directory, the build system will act different.

When `configure` is called from another current working directory than the source directory, this directory will become the *build directory*. All files will be generated below the build directory. This way, you can make as many builds of ESPResSo as you like, each build having different compiler flags and built-in features, and for as many platforms as you want. All further commands concerning compiling and running ESPResSo have to be called from this directory, instead of from the source directory.

When `configure` is called from the source directory where the script resides, the ESPResSo build system has limited built-in capabilities to handle different computer hardware. A new subdirectory is created in the source directory and `configure` is recursively called from this directory, making the subdirectory the build directory. The directory is called `obj-`*platform*`/`, where *platform* is an automatically determined descriptor of the CPU type where the script was started, *e.g.* `obj-Athlon_64-pc-linux`.

[1] `http://www.espresso.mpg.de`

Note that this heuristic will work in many cases, but it may not always work as intended. When you notice any problems, you can always call `configure` from another directory.

In this case it is also possible to run the commands `make` and `Espresso` directly in the source directory. Furthermore, the option `--enable-chooser` will be set in the recursive call of `configure` that activates the automatic binary chooser (see section 3.4.1 on page 23).

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
```

3.2 `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 negatively affect ESPResSo's performance. Instead, compile in only the features that are actually required. 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 141 lists and describes all available features. When no configuration header is provided by the user, a default header 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 (see 3.1 on the previous page), 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. For an example how this can be employed, see section 3.1.

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 3.3 on the facing page), or when `make` is called with the setting `myconfig=`*myconfig_header* (see section 3.4 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
```

3.3 Running `configure`

The shell script `configure` collects all the information required by the compilation process. 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 how to solve the problem. The generic syntax of calling the `configure` script is:

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

Note that in the ESPResSo build system, the files generated by the configuration and compilation process are not placed next to the source files, but into a separate *build directory* instead. Refer to section 3.1 on page 19 for details.

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
```

3.3.1 Options

`--enable-chooser` This option will enable the automatic binary chooser mechanism for ESPResSo (see section 3.4.1 on page 23). This option will be automatically enabled, when the `configure` script is called from the source directory, otherwise it will be disabled. It is not recommended to set the option manually.

`--enable-debug` This option will enable compiler flags required for debugging the ESPResSo binary and is disabled by default.

`--enable-profiling` This option will enable compiler flags required for profiling the ESPResSo binary and is disabled by default.

`--disable-processor-optimization` This option will control whether `configure` will check for several optimization flags to be used by the compiler. This option is enabled by default.

`--disable-xlc-qipa` This option is only useful when the IBM C-compiler `xlc` is used and will control whether or not the compiler flag `-qipa` is used. If you come upon problems when using the ESPResSo binary on IBM machines, try using `--disable-xlc-ipa`. The option is enabled by default.

`--with-myconfig=MYCONFIG_HEADER` This option sets the name of the local configuration header (see 3.2 on page 20). It defaults to "`myconfig.h`".

`--with-mpi=MPI/ --without-mpi` Sets the MPI implementation that should be used, or disables MPI. By default, `configure` will test automatically what MPI implementation is available. The following implementations are known:

`fake, no` This will disable MPI completely. Equivalent to `--without-mpi`.

`lam` Use the LAM/MPI environment (`http://www.lam-mpi.org/`).

`mpich` Use the MPICH environment (`http://www-unix.mcs.anl.gov/mpi/mpich/`).

`poe` Use the POE environment (IBM).

`dmpi` Use the DMPI environment (Tru64).

`generic` Use a generic MPI implementation. This will try to find an MPI compiler and an MPI runtime environment.

`--with-efence / --without-efence` Whether or not to use the "electric fence" memory debugging library (`http://freshmeat.net/projects/efence/`). Efence is not used by default.

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

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

`--with-fftw=VERSION / --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.

3.4 `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 [target...] [variable=value]`

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

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

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

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

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

`dist` Creates a `.tar.gz`-file of the ESPResSo sources. This will include all source files as they currently are in the source directory, *i.e.* it will include local changes. This is useful to give your version of ESPResSo to other people. The variable `extra` can be used to specify additional files and directories that are to be included in the archive file.
Example: `make dist extra="myconfig.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 (*e.g.* `/usr/local/bin64/`). For the actual locations where the different files are installed, refer to section 3.4.1.
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.

3.4.1 Installation directories

Other than most software, ESPResSo is not necessarily installed into the system, but can also be used directly from the build directory. The rest of this section is only interesting if you plan to install ESPResSo.

Normally, the ESPResSo-binary `Espresso-bin` is installed in the directory `$prefix/libexec/` and a the wrapper script `Espresso` in the directory `$prefix/bin/` that handles the MPI invocation.

When the `configure`-script is called from the source directory or when the option `--enable-chooser` is given, an automatic binary chooser is installed in the directory `$prefix/bin/` and the ESPResSo-binary and the MPI wrapper script are installed in an architecture-specific subdirectory `$exec-prefix/lib/espresso/obj-`*platform*`/`. When called, the binary chooser will automatically call the MPI wrapper script from the right subdirectory.

3.5 Running ESPResSo

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

`Espresso [`*tcl_script* `[`*N_processors* `[`*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.

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 ...*] [`q` *charge*] [1] [`quat` *q1 q2 q3 q4*] [2]
[`omega` *x y z*] [2] [`torque` *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]

Required features: [1] `ELECTROSTATICS` [2] `ROTATION` [3] `EXTERNAL_FORCES` [4] `EXCLUSION` [5] `MASS` [6] `DIPOLES`

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* ≥ 0.
  The *typeid* is used in the `inter` command (see section 5 on page 36) 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* ≥ 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.

Docs required (JC?).

- [`q` *charge*] Sets the charge of this praticle to q.
- [`quat` *q1 q2 q3 q4*]

uired

Docs required (JC?).

- [`omega` *x y z*]
- [`torque` *x y z*] When printing the values, unsing *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.
- [`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*`...+`] Restrictions: *pid2* must be an existing particle. Between the current particle an the exclusion partner(s), no nonbonded interactions are calculated. Note that unlike bonds, exclusions are stored with both partners. Therefore this command adds the defined exclusions to both partners.
- [`exclude delete` *pid2*`...`] Searches for the given exclusion and deletes it. Again deletes the exclusion with both partners.
- [`mass` *mass*] Sets the mass of this particle to *mass*. If not set, all particles have a mass of 1 in reduced units.
- [`dipm` *moment*] Sets the dipol moment of this particle to *moment*.
- [`dip` *dx dy dz*] Sets the orientation of the dipol axis to (dx, dy, dz).

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`

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

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

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

4.2 Creating groups of particle

4.2.1 `polymer`: Setting up polymer chains

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` $typeid_{\text{neutral}}$ [$typeid_{\text{charged}}$]]
[`bond` *bondid*] [`angle` ϕ [θ [*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 SHAKE algorithm.

Link to rattle/shake.

- [`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_{charged}] Sets the stride between the indices of two charged monomers. This defaults defaults to 1, meaning that all monomers in the chain are charged.

- [`types` $typeid_{\text{neutral}}$ $typeid_{\text{charged}}$] 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 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.

Link to bonded interactions

- [`angle` ϕ [θ [x y z]]] Allows for setting up helices or planar polymers: ϕ 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 33).

4.2.2 `counterions`: Set up counterions

Syntax

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

Required features: [1] `ELECTROSTATICS`

Description

This command will create N counterions in the simulation box.

Arguments

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

4.2.3 `salt`: Set up salt ions

Syntax

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

Required features: [1] `ELECTROSTATICS`

Description

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

Arguments

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

- [`rad` r] The salt ions are only placed in a sphere with radius r around the origin.

4.2.4 `diamond`: Setting up diamond polymer networks

Syntax

`diamond` a $bond\_length$ $monomers\_per\_chain$ [`counterions` N_{CI}] [`charges` val_{node} val_{monomer} val_{CI}] [1] [`distance` d_{charged}] [1] [`nonet`]

Required features: [1] `ELECTROSTATICS`

Description

Creates a diamond-shaped polymer network with 8 tetra-functional nodes connected by $2 * 8$ polymer chains of length MPC in a unit cell of length a. For inter-particle bonds interaction 0 is taken which must be a two-particle bond.

A picture would be helpful.

Which typeids are used for the different particles?

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 $d_{\mathrm{charged}} > 1$ the remaining chain monomers are uncharged.

Define what *nonet* does.

- [`nonet`]

4.2.5 `icosaeder`: Setting up an icosaeder

Syntax

`icosaeder` a $monomers\_per\_chain$ [`counterions` N_{CI}] [`charges` val_{monomers} val_{CI}] [1] [`distance` d_{charged}] [1]

Required features: [1] `ELECTROSTATICS`

Description

Creates a modified icosaeder to model a fullerene (or soccer ball). The edges are modeled by polymer chains connected at the corners of the icosaeder. For inter-particle bonds interaction 0 is taken which must be a two-particle bond.

A picture would be helpful

Arguments

- a Defines the size of the icosaeder.

- *monomers_per_chain* Specifies the number of chain monomers along one edge.

- [`counterions` N_{CI}] Specifies the number of counterions to be placed into the system.

- [`charges` val_{monomers} val_{CI}] Set the charges of the monomers to val_{monomers} and the charges of the counterions to val_{CI}.

- [`distance` d_{charged}] Specifies the distance between two charged monomer along the edge. If $d_{\mathrm{charged}} > 1$ the remaining monomers are uncharged.

4.2.6 `crosslink`: Cross-linking polymers

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.

4.3 `constraint`: Setting up constraints

Syntax

(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 cylinder center` c_x c_y c_z `axis` n_x n_y n_z `radius` *rad* `length` *length* `direction` *direction* `type` *id*
(4) `constraint maze nsphere` n `dim` d `sphrad` r_s `cylrad` r_c `type` *id*
(5) `constraint pore center` c_x c_y c_z `axis` n_x n_y n_z `radius` *rad* `length` *length* `type` *id*
(6) `constraint rod center` c_x c_y `lambda` *lambda* [1]
(7) `constraint plate height` h `sigma` *sigma* [1]
(8) `constraint ext_magn_field` f_x f_y f_z [2,3]
(9) `constraint plane cell` x y z `type` *id*

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

Description

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

Variants (6) and (7) 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 (8) 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 (9) is essential for the use of tunable-slip boundary interactions for microchannel flows like the Plane Poiseuille or Plane Couette Flow.

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.

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 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 (5) 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.

Variant (6) specifies an electrostatic interaction between the charged particles in the system to an infinitely long rod with a line charge of *lambda* which is alinge along the z-axis and centered at c_x and c_y.

Variant (7) specifies the electrostatic interactinos between the charged particles in the system and an inifinitely large plate in the x-y-plane at height h. The plate carries a charge density of *sigma*.

Variant (8) specifies the dipolar coupling of particles with a dipolar moment to an external field f_x f_y f_z.

Variant (9) 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 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 nth 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.

5 Setting up interactions

In ESPResSo, interactions are setup 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 particles; it however only applies to sets of particles for which it has been explicitely set. A bonded interaction between a set of particles has to be specified explicitely by the `part bond` command, while the `inter` command is used to define the interaction parameters.

5.1 Getting the currently defined interactions

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}
```

5.2 Non-bonded, short-ranged 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 5.2.13 for details.

5.2.1 Lennard-Jones interaction

Syntax

`inter` *type1 type2* `lennard-jones` ϵ σ r_{cut} c_{shift} r_{off} $[r_{\text{cap}}$ $r_{\text{min}}]$

Required features: `LENNARD_JONES`

Description

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

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

The traditional Lennard–Jones potential is the "work–horse" potential for 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_{cut} determines the radius where the potential is cut off. Typically, one will choose the c_{shift} such, that the potential is continuous at the cutoff radius.

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 $r_{\text{cut}} = 2^{\frac{1}{6}}\sigma$ and $c_{\text{shift}} = \frac{1}{4}$. The WCA potential is purely repulsive, and is often used to mimick hard sphere repulsion.

The total force on a particle can be capped by using the command `inter ljforcecap`, see section 5.2.13, or on an individual level using the r_{cap} variable. This is set to 0 by default (no capping).

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

5.2.2 Generic Lennard-Jones interaction

Syntax

`inter` *type1 type2* `lj-gen` ϵ σ r_{cut} c_{shift} r_{off} e_1 e_2 b_1 b_2

Required features: `LENNARD_JONES_GENERIC`

Description

This command defines a generalized version of the Lennard-Jones interaction (see section 5.2.1) between particles of the types *type1* and *type2*. The potential is defined by

$$V_{\text{LJ}}(r) = \begin{cases} 4\epsilon(b_1(\frac{\sigma}{r-r_{\text{off}}})^{e_1} - b_2(\frac{\sigma}{r-r_{\text{off}}})^{e_2} + c_{\text{shift}}) & \text{, if } r_{\text{min}} + r_{\text{off}} < r < r_{\text{cut}} + r_{\text{off}} \\ 0 & \text{, otherwise} \end{cases} . \quad (5.2)$$

The total force on a particle can be capped by using the command `inter ljforcecap`, see section 5.2.13, or on an individual level using the r_{cap} variable. This is set to 0 by default (no capping).

5.2.3 Lennard-Jones cosine interaction

Syntax

(1) `inter` *type1* *type2* `lj-cos` ϵ σ r_{cut} r_{off}
(2) `inter` *type1* *type2* `lj-cos2` ϵ σ r_{off} ω
Required features: (1) `LJCOS` (2) `LJCOS2`

Description
specifies a Lennard-Jones interaction with cosine tail [16] between particles of the types *type1* and *type2*. The first variant behaves as follows: Until the minimum of the Lennard-Jones potential at $r_{\text{min}} = r_{\text{off}} + 2^{\frac{1}{6}}\sigma$, it behaves identical to the unshifted Lennard-Jones potential ($c_{\text{shift}} = 0$). Between r_{min} and r_{cut}, a cosine is used to smoothly connect the potential to 0, *i.e.*

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

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

In the second variant, the cutoff radius is $r_{\text{cut}} = r_{\text{min}} + \omega$, and the potential between r_{min} and r_{cut} is given by

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

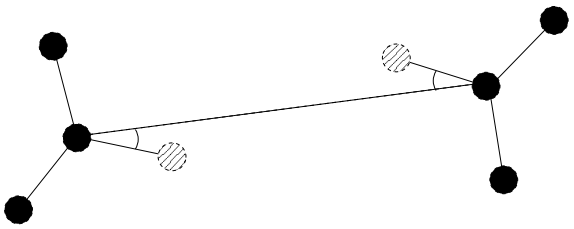
Only the second variant allows capping the force using `inter ljforcecap`, see section 5.2.13.

5.2.4 Directional Lennard-Jones interaction

Syntax

`inter` *type1* *type2* `lj-angle` ϵ σ r_{cut} $b1_a$ $b1_b$ $b2_a$ $b2_b$
Required features: `LJ_ANGLE`

Description



Specifies a 12-10 Lennard-Jones interaction with angular dependence between particles of the types *type1* and *type2*. These two particles need two bonded partners oriented

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

$$U(r_{ik}, \theta_{jik}, \theta_{ikn}) = \epsilon \left[5 \left(\frac{\sigma}{r} \right)^{12} - 6 \left(\frac{\sigma}{r} \right)^{10} \right] \cos^2 \theta_{jik} \cos^2 \theta_{ikn}, \tag{5.5}$$

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 σ. 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).

The contribution to the pressure is not yet implemented.

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.

5.2.5 Smooth step interaction

Syntax

`inter` *type1* *type2* `smooth-step` σ_1 n ϵ k_0 σ_2 r_{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

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

for $r < r_{cut}$, and $V(r) = 0$ elsewhere. With n around 10, the first term creates a short range repulsion similar to the Lennard-Jones potential, while the second term provides a much softer repulsion. This potential therefore introduces two length scales, the range of the first term, σ_1, and the range of the second one, σ_2, where in general $\sigma_1 < \sigma_2$.

5.2.6 BMHTF potential

Syntax

`inter` *type1* *type2* `bmhtf-nacl` A B C D σ r_{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:

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

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

For NaCl, the parameters should be chosen as follows:

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

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

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

5.2.7 Morse interaction

Syntax

`inter` *type1* *type2* `morse` ϵ α r_{min} r_{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 5.2.13.

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

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

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

5.2.8 Buckingham interaction

Syntax

`inter` *type1* *type2* `buckingham` A B C D r_{cut} r_{discont} ϵ_{shift}

Required features: `BUCKINGHAM`

Description

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

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

for $r_{\text{discont}} < r < r_{\text{cut}}$. Below r_{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 5.2.13.

5.2.9 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:

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

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

5.2.10 Gay-Berne interaction

Syntax

`inter` *type1* *type2* `gay-berne` ϵ_0 σ_0 r_{cutoff} *k1* *k2* μ ν

Required features: `ROTATION`

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 σ_0 and the well-depth ϵ_0.

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

$$V(\mathbf{r}_{ij}, \hat{\mathbf{u}}_i, \hat{\mathbf{u}}_j) = 4\epsilon(\hat{\mathbf{r}}_{ij}, \hat{\mathbf{u}}_i, \hat{\mathbf{u}}_j)\left(\tilde{r}_{ij}^{-12} - \tilde{r}_{ij}^{-6}\right), \tag{5.11}$$

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

$$\tilde{r} = \frac{r - \sigma(\hat{\mathbf{r}}, \hat{\mathbf{u}}_i, \hat{\mathbf{u}}_j) + \sigma_0}{\sigma_0}, \tag{5.12}$$

$$\sigma(\hat{\mathbf{r}}, \hat{\mathbf{u}}_i, \hat{\mathbf{u}}_j) = \sigma_0 \left\{1 - \frac{1}{2}\chi\left[\frac{(\hat{\mathbf{r}}\cdot\hat{\mathbf{u}}_i + \hat{\mathbf{r}}\cdot\hat{\mathbf{u}}_j)^2}{1 + \chi\hat{\mathbf{u}}_i\cdot\hat{\mathbf{u}}_j} + \frac{(\hat{\mathbf{r}}\cdot\hat{\mathbf{u}}_i - \hat{\mathbf{r}}\cdot\hat{\mathbf{u}}_j)^2}{1 - \chi\hat{\mathbf{u}}_i\cdot\hat{\mathbf{u}}_j}\right]\right\}^{-\frac{1}{2}} \tag{5.13}$$

and

$$\epsilon(\hat{\mathbf{r}}, \hat{\mathbf{u}}_i, \hat{\mathbf{u}}_j) = \epsilon_0\left(1 - \chi^2(\hat{\mathbf{u}}_i\cdot\hat{\mathbf{u}}_j)\right)^{-\frac{\nu}{2}}\left[1 - \frac{\chi'}{2}\left(\frac{(\hat{\mathbf{r}}\cdot\hat{\mathbf{u}}_i + \hat{\mathbf{r}}\cdot\hat{\mathbf{u}}_j)^2}{1 + \chi'\,\hat{\mathbf{u}}_i\cdot\hat{\mathbf{u}}_j} + \frac{(\hat{\mathbf{r}}\cdot\hat{\mathbf{u}}_i - \hat{\mathbf{r}}\cdot\hat{\mathbf{u}}_j)^2}{1 - \chi'\,\hat{\mathbf{u}}_i\cdot\hat{\mathbf{u}}_j}\right)\right]^{\mu}. \tag{5.14}$$

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

5.2.11 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 5.2.13.

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_{points} and the minimal and maximal tabulated separation distances $r_{\min}$ and $r_{\max}$. 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 $r_{\min}$ and $r_{\max}$ with a fixed distance of $(r_{\max} - r_{\min})/(N_{\text{points}} - 1)$; the distance values r in the file are ignored and only included for human readability.

5.2.12 Tunable-slip boundary interaction

Syntax

`inter` *type1* *type2* `tunable_slip` T γ_L r_{cut} δt 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 δ_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, γ_L the friction coefficient and r_{cut} is the cut-off radius of this layer. δt 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_{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_{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 [15] before using this interaction.

5.2.13 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 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 $F_{\text{max}} = 0$. 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.

5.3 Bonded interactions

Syntax

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

Description

Bonded interactions are identified by their *bonded interaction type identificator bondid*, which is a non-negative integer. The `inter` *bondid* command is used to specify the type and parameters of a bonded interaction, which applies to all particles connected explicitely by this bond using the `part` command (see section 4.1 on page 25). 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 sampe 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 $\Delta r_{\max}$ $[r_0]$

Description

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

$$V(r) = -\frac{1}{2} K {\Delta r_{\max}}^2 \ln \left[1 - \left(\frac{r - r_0}{\Delta r_{\max}} \right)^2 \right]. \tag{5.15}$$

5.3.2 Harmonic bond

Syntax

`inter` *bondid* `harmonic` K R

Description

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

$$V(r) = \frac{1}{2} K \left(r - R \right)^2 \tag{5.16}$$

5.3.3 Subtracted Lennard-Jones bond

Syntax

`inter` *bondid* `subt_lj` *reserved* R

Description

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

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

5.3.4 Rigid bonds

Syntax

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

Description

Docs

5.3.5 Bond-angle interactions

Syntax

`inter` *bondid* `angle` K $[\phi_0]$

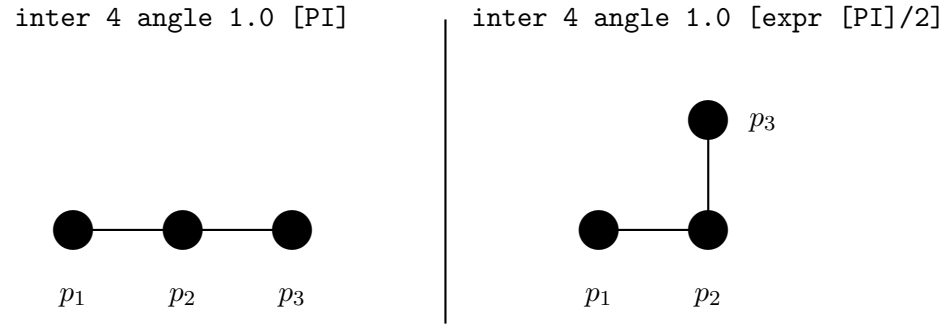
Required features: `BOND_ANGLE_HARMONIC` `BOND_ANGLE_COSINE` `or BOND_ANGLE_COSSQUARE`

Description

This creates a bond type with identificator *bondid* with an angle dependent potential. This potential is defined between three particles. The particle for which the bond is created, is the central particle, and the angle ϕ between the vectors from this particle to the two others determines the interaction. K is the bending constant, and the optional parameter phi_0 is the equilibirum bond angle in radian ranging from 0 to π. 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,

$$V(\phi) = \frac{K}{2}\left(\phi - \phi_0\right)^2 . \tag{5.17}$$

  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):

$$V(\alpha) = K\left[1 - \cos(\phi - \phi 0)\right] \tag{5.18}$$

  Around ϕ_0, this potenial is close to a harmonic one (both are $1/2(\phi - \phi_0)^2$ in leading order), but it is periodic and smooth for all angles ϕ.

- Cosine square bond angle potential (requires feature BOND_ANGLE_COSSQUARE):

$$V(\alpha) = \frac{K}{2}\left[\cos(\phi) - \cos(\phi_0)\right]^2 \tag{5.19}$$

  This form is used for example in the GROMOS96 force field. The potential is $1/8(\phi - \phi_0)^4$ around ϕ_0, and therefore much flatter than the two potentials before.

5.3.6 Dihedral interactions

Syntax

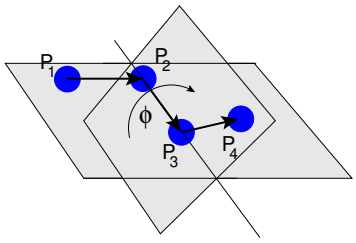
`|inter` *bondid* `dihedral` n K p

Description

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

$$V(\phi) = K\left[1 - \cos(n\phi - p)\right], \tag{5.20}$$

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. ϕ is the dihedral angle between the particles defined by the particle quadrupel p_1, p_2, p_3 and p_4, *i.e.* the angle between the planes defined by the particle triples p_1, p_2 and p_3 and p_2, p_3 and p_4:



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

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

$$V(\phi) = K\left[1 + p\,\cos(n\phi)\right], \tag{5.21}$$

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

5.3.7 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 identificator *bondid* with a two-body bond length ((1)), three-body angle ((2)) or four-body dihedral ((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 section5.2.11).

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 5.3.5). It is assumed that the potential is tabulated for all angles between 0 and π, 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 5.3.5) acts

perpendicular to the connecting vector between the particle and the center particle p_2 in the plane defined by the three particles. The force on the center particle p_2 balances the other two forces.

- Variant (3) tabulates a torsional dihedral angle potential (see section 5.3.6). It is assumed that the potential is tabulated for all angles between 0 and 2π. *This potential is not tested yet! Use on own risk, and please report your findings and eventually necessary fixes.*

5.3.8 Virtual bonds

Syntax

| `inter` *bondid* `virtual_bond`

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

5.4 Coulomb interaction

Electrostatic interactions are very computation time-intensive. 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.

Syntax

| (1) `inter coulomb 0.0`
| (2) `inter coulomb` [l_B *method*] [*parameters*]
| (3) `inter coulomb`

Description
This command defines how ESPResSo deals with electrostatic interactions.

Variant (1) completely disables Coulomb interactions hence deactivating the electrostatic subsystem, while variant (2) sets up one of the methods described below to treat electrostatic interactions. l_B denotes the Bjerrum length, which measures the strength of the electrostatic interaction. For a pair of particles at distance r with charge q each, the interaction is given by

$$U^C(r) = l_B k_B T \frac{q^2}{r}. \tag{5.22}$$

Using the electrostatic interaction also requires to assign 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
```

Variant (3) 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}
```

5.4.1 P3M

Syntax

(1) `inter coulomb p3m` r_{cut} *mesh cao alpha*
(2) `inter magnetic p3m` r_{cut} *mesh cao alpha*

Description

The variant (1) activates the P3M method to handle the electrostatic interactions (charge-charge), whereas the variant (2) activates the P3M method to handle the magnetostatic interactions (magnetic dipole-dipole) (aka dipolar-P3M).

For electrostatics the interaction is:

$$U^{C-P3M} = \ell_B k_B T \frac{q_1 q_2}{r} \tag{5.23}$$

Here $\ell_B = e_o^2/(4\pi\epsilon k_B T)$ is the Bjerrum length.

For magnetostatics the interaction is:

$$U^{D-P3M} = \ell_{Bm} k_B T \left(\frac{(\vec{\mu}_i \cdot \vec{\mu}_j)}{r^3} - \frac{3(\vec{\mu}_i \cdot \vec{r}_{ij})(\vec{\mu}_j \cdot \vec{r}_{ij})}{r^5} \right) \tag{5.24}$$

Here $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$, and ℓ_{Bm} is a non dimensional parameter similar to the Bjerrum length in electrostatics which can help to tune the effect of the medium on the magnetic interaction between two magnetic dipoles.

Make sure that you know the relevance of the P3M parameters before using P3M! If you are not sure, read the following references [6, 8, 10, 3, 4, 5, 2, 1].

Tuning P3M

Syntax

`inter ( coulomb | magnetic ) p3m ( tune | tunev2 ) accuracy` *accuracy*
`[r_cut` r_{cut}`] [mesh` *mesh*`] [cao` *cao*`] [alpha` α`]`

Description

Make sure you know how to tune p3m parameters before using the automatic tuning feature.

For the charge-charge interaction the function utilizes the analytic expression of the error estimate for the P3M method in the book of Hockney and Eastwood [8, eqn 8.23] in order to obtain the rms error in the force for a system of N randomly distributed particles in a cubic box. For the real space error the estimate of Kolafa and Perram[10] is used. For the magnetic case, magnetic dipole-dipole interaction, the expressions of the error estimate for the dipolar-P3M are those written in Cerda et al. [1]. At this

moment, the P3M methods only work for cubic boxes. The magnetostatics must be activated enabling the flag "MAGNETOSTATICS" in the file "myconfig.h".

The two tuning methods follow different methods for determining the optimal parameter. While the `tune` version simply 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 achievable from the `tune` version, and normally the obtained accuracy is much closer to the desired value.

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

Note that any previous settings of r_{cut}, *cao* and *mesh* will be remembered. So if you want to retune your electrostatics, *e.g.* after a major system change, you should use

`inter ( coulomb | magnetic )` l_B `p3m tune accuracy` *acc* `r_cut 0 mesh 0 cao 0`

Some additional p3m parameters have preset value

`epsilon = metallic`

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

`n_interpol = 32768`

Number of interpolation points for the charge assignment function. When this is set to 0, interpolation is turned off.

`mesh_off = 0.5 0.5 0.5`

Offset of the first mesh point from the lower left corner of the simulation box in units of the mesh constant. As soon as p3m is turned on the additional parameters can be changed with:

`inter coulomb` *parameter*$_n$*ame value*`+`

If only magnetic dipoles are present in the sytem (no charges present), it is advisable to turn off the flag "ELECTROSTATICS" in the file "myconfig.h" to improve the performance of the program. Mixed systems containing magnetic dipoles and charges can be simulated without problem enabling both flags: "ELECTROSTATICS" and "MAGNETOSTATICS". For each kind of particle one should use the prefered algorithm. For magnetic dipole-dipole interactions the only algorithm implemented at this moment is the dipolar-P3M. In principle there is no problem to use a non P3M method to deal with charges (see folowing sections) while one uses dipolar-P3M to deal with the magnetic dipoles. In case you use P3M for both charges and magnetic dipoles, make sure that you tune each P3M method (coulomb/magnetic) to the desired accuracy.

he cutoff
Why is
a normal
ed
?

5.4.2 Debye-Hückel potential

Syntax

`inter coulomb dh` κ r_{cut}

Description
Defines the electrostatic potential by

$$U^{C-DH} = \ell_B k_B T \frac{q_1 q_2 exp(-\kappa r)}{r} \tag{5.25}$$

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

5.4.3 MMM2D

Syntax

`inter coulomb mmm2d` *maximal_pairwise_error* [*fixed_far_cutoff*]
[`dielectric` ϵ_t ϵ_m ϵ_b] [`dielectric-contrasts` Δ_t Δ_b]

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 **??** on page ??.

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.

5.4.4 MMM1D

Syntax

(1) `inter coulomb mmm1d` *switch_radius* [*bessel_cutoff*] *maximal_pairwise_error*
(2) `inter coulomb mmm1d tune` *maximal_pairwise_error*

Description
MMM1D coulomb method for systems with periodicity 0 0 1. Needs the nsquared cell system (see section 6.4 on page 63). 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 ?? on page ??.

5.4.5 Maggs' method

Syntax

| `inter coulomb maggs` *f_mass mesh field_friction* [`yukawa` *kappa* r_{cut}]

Description

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

$$U = \ell_B k_B T \frac{q_1 q_2}{r} \tag{5.26}$$

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

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.
- *field_friction* is the value of the friction coefficient for the transversal field degrees of freedom (reserved for future development).

Unphysical self–energies that arise as a result of the lattice interpolation of charges, are corrected by a subtraction scheme based either on the exact lattice Green's function or the combination of the direct subtraction scheme plus the Yukawa subtraction scheme (second method).

For the case of Yukawa screened simulation (second method) one has to enter screening parameter *kappa* and the cut-off of the Yukawa potential r_{cut}.

5.4.6 ELC

Syntax

| `inter coulomb elc` *maximal_pairwise_error gap_size* [*far_cutoff*]

Description

This is a special procedure that converts a 3d method, *i.e.* P3M at the moment, to a 2d method, in computational order N. This is definitely faster than MMM2D for larger numbers of particles (¿400 at reasonable accuracy requirements). The maximal pairwise error is the LUB error of the force between any two charges without prefactors (see

the papers). The 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 far cutoff finally 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 ?? on page ??.

Make sure that you read the papers on ELC before using it !!!

references

5.4.7 DLC

Syntax

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

Description

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

At present, the program assumes that the gap wihtout 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 DLC-method, see: A.Brodka, Chem. Phys. Let. 400, 62-67, (2004).

5.4.8 MDDS

Syntax

| `inter magnetic` $l_B m$ `mdds n_cut` *value_n_cut* [*far_cutoff*]

Description

The command enables the magnetic dipolar direct sum. This method is not intendeed to do simulations, rahter to check the results you get from Espresso. It is the exact calculation all the other magnetic methods with PBC try to mimic but doing the calculation faster. The value-n-cut is the number of replicas that should be taken into account in each direction If Periodic Boundaries are not in 3 dimensions, the mehtod takes only replicas in those directions which are periodic. Thus, using partially periodic boundaries, once can easily calculate the values for 2D+h and 1D+h systems.

5.4.9 DAWAANR

Syntax

| `inter magnetic` $l_B m$ `dawaanr` [*far_cutoff*]

Description

The command enables the "dipolar all-with-all and no replicas" method. It is a very special method intended to calculations when you have a single magnetic polymer chain inside and infinite volume where no replicas of the system exist along any of the directions in the space. You should setup formally a box length large enough for the initial conformation to be in, but after that, the box has no use. No minimum image convention is used. You compute the interactions from the positions where the particles has diffused to at each moment. You can use for particles not linked, but the method only makes sense in the case you want to study the evolution of a single chain in the space.

Notice that P3M and any other methods that assumes PBC is of no practical use here. A rough way to mimic it with P3M would be assigning a very long box, but it would be a waste of time and resources.

5.5 Other interaction types

5.5.1 Fixing the center of mass

Syntax

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

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).

5.5.2 Pulling particles apart

Syntax

| `inter` *typeid1* *typeid2* `comforce` *flag* *dir* *force* *fratio*

Description

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

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:

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

Missing: lattice_switch, dpd_tgamma, n_rigidbonds

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

document what happens to the particles when `box_l` is changed!

`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_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_num_cells` (int) Minimal number of cells for the link cell algorithm. Reasonable values range in $1e-6N^2$ to $1e-7N^2$. 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 larger values for high particle numbers.

`n_layers` (int, *read-only*) Number of layers in cell structure LAYERED (see section 6.4 on page 63).

`n_nodes` (int, *read-only*) Number of nodes.

`n_part` (int, *read-only*) Total number of particles.

`n_part_types` (int, *read-only*) Number of particle types that were used so far in the `inter` command (see chaptertcl:inter).

`node_grid` (int[3]) 3D node grid for real space domain decomposition (optional, if unset an optimal set is chosen automatically).

`nptiso_gamma0` (double, *read-only*)

Docs missing.

`nptiso_gammav` (double, *read-only*)

Docs missing.

`npt_p_ext` (double, *read-only*) Pressure for NPT simulations.

`npt_p_inst` (double) Pressure calculated during an NPT_isotropic integration.

`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.

6.2.1 Langevin thermostat

Syntax

| `thermostat langevin` *temperature gamma*

Description

Reference

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.

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

6.2.2 Dissipative Particle Dynamics (DPD) thermostat

Syntax

| `thermostat dpd` *temperature gamma r_cut* [`WF` *wf tgamma tr_cut* `TWF` *twf*]

Description

Standard DPD thermostat

ESPResSo implements the DPD thermostat exactly as it is described in [17]. We use the standard *Velocity-Verlet* integration scheme, e.g. the DPD only influences the calculation of the forces. No special measurements have been taken into account to selfconsistently determine the velocities and the dissipative forces as it is for example described in [13]. DPD adds a velocity dependent dissipative force and a random force to the usual conservative pairforces (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 σ and ζ is given by the dissipation fluctuation theorem:

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

The parameters *gamma r_cut* define the strength of the friction ζ 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*) :

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

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 propably want to try this out yourself to get a feeling for how fast temperature jumps during a simulation are. The dpd thermostat does not act on the system center of mass motion. Therefore, before using dpd, you have to stop the center of mass motion of your system, which you can achieve by using the command `galileiTransformParticles` (see section **??** on page ??) . This may be repeated once in a while for long runs due to round off errors (check this with the command `system_com_vel` from section **??** on page ??).

Two restrictions apply for the dpd implementation of ESPResSo:

- As soon as at least one of the two interacting particles is fixed (see 4 on how to fix a particle in space) the dissipative and the stochastic force part is set to zero for both particles (you should only change this hardcoded behaviour if you are sure not to violate the dissipation fluctuation theorem).
- DPD does not take into account any internal rotational degrees of freedom of the particles if ROTATION is switched on. Up to the current version DPD only acts on the translatorial degrees of freedom.

Transverse DPD thermostat

This is an extension of the above thermostat[9], which dampes the degrees of freedom perpendicular on the axis between two particles. To switch it on, the feature TRANS_-DPD is required.

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: That this thermostat does NOT conserve angular momentum!!!!

Interaction DPD thermostat

The DPD thermostat can also be set up as a normal interaction to make friction between different particle types different.

Syntax

| `thermostat inter_dpd` *temperature*

Description
and

Syntax

| `inter` *type1 type2* `inter_dpd` *gamma r_cut* [`WF` *wf tgamma tr_cut* `TWF` *twf*]
| Required features: `INTER_DPD`

Description
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 cases implemented. 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*

Description
This theormstat is based on the Anderson thermostat and will thermalize the box geometry. It will only do isotropic changes of the box.

Docs, reference

6.2.4 Turning off all thermostats

Syntax

| `thermostat off`

Description
Turns off all thermostats and sets all thermostat variables to zero.

6.2.5 Getting the parameters

Syntax

| `thermostat`

Description
Returns the thermostat parameters.

Document return format.

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`

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 [16].

Variant (1) chooses the momentum exchange method. In this method, in each step the *n_exchange* largest positive x-components of the velocity in the middle slab are selected and exchanged with the *n_exchange* largest negative x-components of the velocity in the top slab.

Variant (2) chooses the shear-rate method. In this method, the targetted x-component of the mean velocity in the top and middle slabs are given by

$$target\_velocity = \pm shearrate \frac{L_z}{4} \tag{6.1}$$

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

$$\eta = \frac{F}{\dot{\gamma} L_x L_y} \tag{6.2}$$

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 12.1 on page 131.

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 cellsystem is the default system and suits most applications with short ranged interactions. The particles are divided up spatially into small

compartments, the cells, such that the cell size is larger than the maximal interaction range. In this case interactions only occur between particles in adjacent cells. Since the interaction range should be much smaller than the total system size, leaving out all interactions between non-adjacent cells can mean a tremendous speed-up. Moreover, since for constant interaction range, the number of particles in a cell depends only on the density. The number of interactions is therefore of the order N instead of order N^2 if one has to calculate all pair interactions.

6.4.2 N-squared

Syntax

```
|cellsystem nsquare
```

Description

This selects the very primitive nsquared cellsystem, which calculates the interactions for all particle pairs. Therefore it loops over all particles, giving an unfavorable computation time scaling of N^2. However, algorithms like MMM1D or the plain Coulomb interaction in the cell model require the calculation of all pair interactions.

In a multiple processor environment, the nsquared cellsystem uses a simple particle balancing scheme to have a nearly equal number of particles per CPU, *i.e.* n nodes have m particles, and $p-n$ nodes have $m+1$ particles, such that $n*m+(p-n)*(m+1) = N$, the total number of particles. Therefore the computational load should be balanced fairly equal among the nodes, with one exception: This code always uses one CPU for the interaction between two different nodes. For an odd number of nodes, this is fine, because the total number of interactions to calculate is a multiple of the number of nodes, but for an even number of nodes, for each of the $p-1$ communication rounds, one processor is idle.

E.g. for 2 processors, there are 3 interactions: 0-0, 1-1, 0-1. Naturally, 0-0 and 1-1 are treated by processor 0 and 1, respectively. But the 0-1 interaction is treated by node 1 alone, so the workload for this node is twice as high. For 3 processors, the interactions are 0-0, 1-1, 2-2, 0-1, 1-2, 0-2. Of these interactions, node 0 treats 0-0 and 0-2, node 1 treats 1-1 and 0-1, and node 2 treats 2-2 and 1-2.

Therefore it is highly recommended that you use nsquared only with an odd number of nodes, if with multiple processors at all.

6.4.3 Layered cell system

Syntax

```
|cellsystem layered n_layers
```

Description

This selects the layered cell system, which is specifically designed for the needs of the MMM2D algorithm. Basically it consists of a nsquared algorithm in x and y, but a domain decomposition along z, i. e. the system is cut into equally sized layers along the

z axis. The current implementation allows for the cpus to align only along the z axis, therefore the processor grid has to have the form 1x1xN. However, each processor may be responsible for several layers, which is determined by *n_layers*, i. e. the system is split into N*n_layers layers along the z axis. Since in x and y direction there are no processor boundaries, the implementation is basically just a stripped down version of the domain decomposition cellsystem.

7 Running the simulation

7.1 `integrate`: Running the simulation

Syntax

(1) `integrate` *steps*
(2) `integrate set` *method* [*parameter*]...

Description

Docs missing!

7.2 `change_volume`: Changing the box volume

Which integrators do exist?

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`

Documentation not up to date!

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 ???, while `read_checkpoint - integrate` has two at the beginning of the 2nd integrate (because loading a new system from disk typically requires re-initializing the system), and since ??? 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 (this is being done automatically if using tcl_checkpoint_set and tcl_checkpoint_read beginning with v1.1 of ESPResSo), because in that case both scenarios will call ??? twice at the beginning of the second integration phase thus having their random number sequences in total sync.

???

7.6 Parallel tempering

Syntax

`|parallel_tempering::main -rounds` N `-swap` *swap* `-perform` *perform*
`[-init` *init*`]` `[-values` $\{T_i\}$`]` `[-connect` *master*`]` `[-port` *port*`]`
`[-load` j_{node}`]` `[-resrate` N_{reset}`]` `[-info` *info*`]`

Description

This command can be used to run a parallel tempering simulation. Since the simulation routines and the calculation of the swap probabilities are provided by the user, the method is not limited to sampling in the temperature space. However, we assume in the following that the sampled values are temperatures, and call them accordingly. It is possible to use multiple processors via TCP/IP networking, but the number of processors can be smaller than the number of temperatures.

Arguments

- *swap* specifies the name of the routine calculating the swap probability for a system. The routine has to accept three parameters: the *id* of the system to evaluate,

and two temperatures T_1 and T_2. The routine should return a list containing the energy of the system at temperatures T_1 and T_2, respectively.

- *perform* specifies the name of the routine performing the simulation between two swap tries. The routine has to accept two parameters: the *id* of the system to propagate and the temperature T at which to run it. Return values are ignored.
- *init* specifies the name of a routine initializing a system. This routine can for example create the particles, perform some intial equilibration or open output files. The routine has to accept two parameters: the *id* of the system to initialize and its initial temperature T. Return values are ignored.
- R specifies the number of swap trial rounds; in each round, neighboring temperatures are tried for swapping alternatingly, i.e. with four temperatures, The first swap trial round tries to swap $1 \leftrightarrow 2$ and $3 \leftrightarrow 4$, the second round $2 \leftrightarrow 3$, and so on.
- *master* the name of the host on which the parallel_tempering master node is running.
- *port* the TCP/IP port on which the parallel_tempering master should listen. This defaults to 12000.
- j_{node} specifies how many systems to run per ESPResSo-instance. If this is more than 1, it is the user's responsibility to manage the storage of configurations, see below for examples. This defaults to 1.
- R_{reset} specifies after how many swap trial rounds to reset the counters for the acceptance rate statistics. This defaults to 10.
- *info* specifies which output the parallel tempering code should produce:

  `none` parallel tempering will be totally quiet, except for fatal errors

  `comm` information on client activities, such as connecting, is printed to stderr

  `all` print lots of information on swap energies and probabilities to stdout. This is useful for debugging and quickly checking the acceptance rates.

  This defaults to `all`.

Introduction

The basic idea of parallel tempering is to run N simulations with configurations C_i in parallel at different temperatures $T_1 < T_2 < \ldots < T_N$, and exchange configurations between neighboring temperatures. This is done according to the Boltzmann rule, *i.e.* the swap probability for two configurations A and B at two different parameters T_1 and T_2 is given by

$$\min\left(1, \exp - \left[\beta(T_2)U_{\mathrm{A}}(T_2) + \beta(T_1)U_{\mathrm{B}}(T_1) - \beta(T_1)U_{\mathrm{A}}(T_1) - \beta(T_2)U_{\mathrm{B}}(T_2)\right]\right), \qquad (7.1)$$

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 indepedent of T,

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

$$\min(1, \exp - [(1/T_2 - 1/T_1)(U_\mathrm{A} - U_\mathrm{B})/k_B]. \tag{7.2}$$

However, T can also be chosen to be any other parameter, for example the Bjerrum length, *i.e.* the the strength of the electrostatic interaction. In this case, $\beta(T) = \beta$ is a constant, but the energy $U_C(T)$ of a configuration C depends on T, and one needs the full expression (7.1). 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.

Th ESPResSo implementation of parallel tempering repeatedly propagates all configurations C_i and tries to swap neighboring temperatures. After the first propagation, the routine attempts to swap temperatures T_1 and T_2, T_3 and T_4, and so on. After the second propagation, swaps are attempted between temperatures T_2 and T_3, T_4 and T_5, and so on. For the propagation, parallel tempering relies on a user routine; typically, one will simply propagate the configuration by a few 100 MD time steps.

Details on usage and an example

The parallel tempering code has to be loaded explicitely by `source "scripts/parallel_tempering.tcl"` from the Espresso directory. To make use of the parallel tempering tool, one needs to implement three methods: the propagation, the energy calculation and an initialization routine for a configuration. A typical initialization routine will look roughly like this:

```
proc init {id temp} {
  # create output files for temperature temp
  set f [open "out-$temp.dat" w]; close $f
  init_particle_positions
  thermostat langevin $temp 1.0
  equilibration_integration
  global config
  set config($id) "{[part]} [setmd time]"
}
```

The last two lines are only necessary if each instance of ESPResSo handles more than one configuration, *e.g.* if you have 300 temperatures, but only 10 ESPResSo processes (*i.e.*`-load 30`). In this case, all user provided routines need to save and restore the configurations. Saving the time is not necessary because the simulation tine across swaps is not meaningful anyways; it is however convenient for investigating the (temperature-)history of individual configurations.

A typical propagation routine accordingly looks like this

```
proc perform {id temp} {
  global config
```

```
  particle delete
  foreach p [lindex $config($id) 0] { eval part $p }
  setmd time [lindex $config($id) 1]
  thermostat langevin $temp 1.0
  set f [open "out-$temp.dat" a];
  integrate 1000
  puts $f "[setmd time] [analyze energy]"
  close $f
  set config($id) "{[part]} [setmd time]"
}
```

Again, the saving and storing of the current particle properties in the config array are only necessary if there is more than one configuration per process. In practice, one will rescale the velocities at the beginning of perform to match the current temperature, otherwise the thermostat needs a short time to equilibrate. The energies necessary to determine the swap probablility are calculated like this:

```
proc swap {id temp1 temp2} {
  global config
  particle delete
  foreach p $config($id) { eval part $p }
  set epot [expr [analyze energy total] - [analyze energy kinetic]]
  return "[expr $epot/$temp1] [expr $epot/$temp2]"
}
```

Note that only the potential energy is taken into account. The temperature enters only indirectly through the inverse temperature prefactor, see Eqn. (7.1).

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`.

8 Analysis

Intro: `analyze` can measure observables, but also define topologies and store configurations

Missing: `radial_density_map`, `modes2d`, `get_lipid_orients`, `get_folded_positions`, `bilayer_set`, `bilayer_density_profile`, `lipid_orient_order`, `cell_gpb`, `Vkappa`

8.1 Measuring observables

The `analyze`-command 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 $log_f lag \geq 1$). If an integrated distribution is required, use $int_f lag = 1$. 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 9.1 on page 88):

{ *parameters* }
{
  { *r dist*(*r*) }
  ⋮
}

8.1.4 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 9.1 on page 88):

{ *parameters* }
{
  { *r rdf*(*r*) }
  ⋮
}

8.1.5 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, $\frac{2\pi}{L} <= q <= \frac{2\pi}{L} order$. Do not chose parameter *order* too large, becase the number of calculations grows as $order^3$.

Output format
The output corresponds to the blockfile format (see section 9.1 on page 88):

{ *q_value* $S(q)$*_value* }
⋮

8.1.6 Van-Hove autocorrelation function $G(r,t)$

Syntax

`|analyze vanhove` *type rmin rmax rbins*

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 84) 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 acoording to the specification given by *rmin*, *rmax* and *rbins*. 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 9.1 on page 88):

{ `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.

8.1.7 Center of mass

Syntax

`|analyze centermass` $part_type$

Description

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

8.1.8 Moment of intertia 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.

8.1.9 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.

8.1.10 Identifying pearl-necklace structures

Syntax

`analyze necklace` *pearl_treshold back_dist space_dist first length*

Description

Algorithm for identifying pearl necklace structures for polyelectrolytes in poor solvent [12]. The first three parameters are tuning parameters for the algorithm: *pearl_treshold* 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.

8.1.11 Finding holes

Syntax

| `analyze holes` $typeid_{\text{probe}}$ $mesh\_size$

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 [14]. 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 fictious 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 continous 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. .

I think there is still a bug in there (Hanjo)

8.1.12 Energies

Syntax

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

Describe t different e componen returned b different commands

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* } ...

8.1.13 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.

Document arguments nb_inter, nb_intra, tot_nb_inter and tot_nb_intra

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

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

Description of how electrostatic contribution to Pressure is calculated

where f is the number of degrees of freedom of each particle, V is the volume of the system, $E_{kinetic}$ is the kinetic energy, F_{ij} the force between particles i and j, and r_{ij} is the distance between them. The kinetic energy divided by the degrees of freedom is

$$\frac{2E_{kinetic}}{f} = \frac{1}{3}\sum_i m_i v_i^2 \tag{8.2}$$

when the ROTATION option is turned off and

$$\frac{2E_{kinetic}}{f} = \frac{1}{6}\sum_i (m_i v_i^2 + I_i w_i^2)) \tag{8.3}$$

when the ROTATION option is compiled in. I_i is the moment of inertia of the particle and w_i is the angular velocity.

Care should be taken when using constraints of any kind, since these are not accounted for in the pressure calculations.

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.

8.1.14 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 8.1.13 on the preceding page. It is called a stress tensor but the sign convention follows that of a pressure tensor.

The stress tensor is calculated by

$$p^{(kl)} = \frac{\sum_i m_i v_i^{(k)} v_i^{(l)}}{V} + \frac{\sum_{j>i} F_{ij}^{(k)} r_{ij}^{(l)}}{V} \tag{8.4}$$

where the notation is the same as for `analyze pressure` in 8.1.13 on the previous page 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. This means that when the ROTATION option is compiled in the instantaneous pressure calculated with `analyze pressure` will be different from the pressure implied by the stress tensor. However, the time averages should be in agreement.

If the P3M and MMM1D electostatic methods are used, these interactions are not included in the stress tensor. The DH and RF methods, in contrast, are included.

Care should be taken when using constraints of any kind, since these are not accounted for in the stress tensor calculations.

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.

8.1.15 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 in divided into *bins_x* bins in the x direction, *bins_y* bins in the y direction and *bins_z* bins in the z direction such that the total number of bins is *bins_x***bins_y***bins_z*. For each of these bins a stress tensor is calculated using the Irving Kirkwood method. That is, a given interaction contributes towards the stress tensor in a bin proportional to the fraction of the line connecting the two particles that is within the bin.

If the P3M and MMM1D electostatic methods are used, these interactions are not included in the local stress tensor. The DH and RF methods, in contrast, are 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 Topologies

Topologies intro

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 chains`

Update documentation for set_topology

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 $chain\_start + n\_chains * chain\_length$). Variant (2) 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 84).

Output format
{ *re error_of_re re2 error_of_re2* }

Radius of gyration

Syntax

`analyze ( rg | <rg> )` [*chain_start n_chains chain_length*]

Reference?

Description
Returns the radius of gyration averaged over all chains. If `<rg>` is used, the radius of gyration is averaged over all stored configurations (see section 8.3 on page 84).

Output format
{ *rg* *error_of_rg* *rg2* *error_of_rg2* }

Hydrodynamic radius

Syntax

`analyze ( rh | <rh> )` [*chain_start* *n_chains* *chain_length*]

Reference?

Description
Returns the hydrodynamic radius averaged over all chains. If `<rh>` is used, the hydodynamic radius is averaged over all stored configurations (see section 8.3 on page 84).

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. If `<internal_dist>` is used, the values are averaged over all stored configurations (see section 8.3 on page 84).

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, ...).

Bond distances

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 84).

Output format
{ $bdf(0)$ $bdf(1)$ ... $bdf(chain\_length - 1 - index)$ }

Bond lengths

Syntax

| `analyze ( bond_l | <bond_l> )` [*chain_start n_chains chain_length*]

Description
Analyses the bond lengths of the chains in the system. Returns its average, the standard deviation, the maximum and the minimum. If you want to look only at specific chains, use the optional arguments, *i.e.* $chain\_start = 2 * MPC$ and $n\_chains = 1$ to only include the third chain's monomers. If `<bond_l>` is used, the value will be averaged over all stored configurations (see section 8.3 on page 84).

Output format
{ *mean stddev max min* }

Form factor

Syntax

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

Description

Check this!

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

$$S(q) = \frac{1}{chain\_length} \sum_{i,j=1}^{chain\_length} \frac{\sin(qr_{ij})}{qr_{ij}} \tag{8.5}$$

of a single chain, averaged over all chains for $qbin + 1$ logarithmically spaced q-vectors $qmin, \ldots, qmax$ where $qmin > 0$ and $qmax > qmin$. If `<formfactor>` is used, the form factor will be averaged over all stored configurations (see section 8.3 on page 84).

Output format
{
  { q $S(q)$ }
  ⋮
}

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

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 $rdf1(r)$ $rdf2(r)$ $rdf3(r)$ }
  ⋮
}

g123

Title?

Syntax

| (1) `analyze ( <g1>| <g2>| <g3> )` [$chain_start$ n_chains $chain_length$]
| (2) `analyze g123 [-init]` [$chain_start$ n_chains $chain_length$]

What's the difference between g2 and g3???

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 on the next page).

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 9.1 on page 88).

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 ...* }
  ⋮
}

Make this an appendix?

8.4 Statistical analysis and plotting

8.4.1 Plotting

Syntax

`plotObs` *file* { *x1*`:`*y1 x2*`:`*y2 ...*} [`titles` { *title1 title2 ...*}]
[`labels` { *xlabel* [*ylabel*] }] [`scale` *gnuplot − scale*]
[`cmd` *gnuplot − command*] [`out` *filebase*]

Description

Uses GNUPLOT to create plots of the data in *file* and writes it to the file *filebase*`.ps` (default: *file*`.ps`). The data in *file* should be stored column-wise. *x1*, *x2* ... and *y1*, *y2* ... denote the columns used for the data of the x- and y-axis, respectively.

Arguments

- [`titles` { *title1 title2 ...*}] can be used to specify the titles of the different plots
- [`labels` { *xlabel* [*ylabel*] }] will define the labels of the axis. If *ylabel* is omitted, the filename *file* is used as label for the y-axis.
- [`scale` *gnuplot − scale*] will define the scaling of the axis (*e.g.* `scale logscale xy`) (default: `nologscale xy`)
- [`cmd` *gnuplot − command*] allows to pass any other commands to gnuplot. For example, use `plotObs ...cmd "set key left"` to adjust the titles on the left side.
- [`out` *filebase*] can be used to change the output file. By default, the plot will be written to *file*`.ps`.

8.4.2 Joining plots

Syntax

`plotJoin` { *source1 source2 ...*} *final*

Description

Joins the plot files *source1*, *source2*, ... into a single file *final*, while placing any two files on one page. Note that the resulting files may be huge and therefore hard to print!

8.4.3 Computing averages and errors

Syntax

(1) `calcObAv` *file index* [*start*]
(2) `calcObErr` *file index* [*start*]
(3) `calcObsAv` *file* { *i1 i2 ...*} [*start*]
(4) `nameObsAv` *file* { *name1 name2 ...*} [*start*]
(5) `findObsAv` *val what*

Description

These commands will compute mean values or errors of the data in file *file*. The data in *file* should be stored column-wise. If *start* is specified, the first *start* lines will be ignored.

Variant (1) returns the mean value of the column with index *index* in *file*, variant (2) returns the error of its mean value. Variant (3) computes mean values and errors of the observables with index $i1, i2, \ldots$ in *file*. It expects the first line of *file* to contain the names of the columns, which it will also return.

In variant (4), the names used in the first line of *file* can be used to specify which column is to be used. The mean value and its error are computed for each of the columns.

????

Variant (5) extracts the values whose names are given in the tcl-list *val* at their respective positions in *what*, where *what* has the list-format as returned by variant (3), returning just these values as tiny tcl-list.

Output format (variant (3)*)*

```
{
  #samples
  { name1 name2 ... }
  { mean1 mean2 ... }
  { error1 error2 ... }
}
```

Output format (variant (4)*)*

```
{
  #samples
  mean1 mean2 ...
  error1 error2 ...
}
```

8.5 `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 accordings to Wolff [18].

Arguments

How exactly does the Tcl-list look like?

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

$$data = \begin{pmatrix} a_1^{1,1} & a_2^{1,1} & a_3^{1,1} & \cdots \\ a_1^{2,1} & a_2^{2,1} & a_3^{2,1} & \cdots \\ \vdots & \vdots & \vdots & \vdots \\ a_1^{N_1,1} & a_2^{N_1,1} & a_3^{N_1,1} & \cdots \\ a_1^{1,2} & a_2^{1,2} & a_3^{1,2} & \cdots \\ \vdots & \vdots & \vdots & \vdots \\ a_1^{N_R,R} & a_2^{N_R,R} & a_3^{N_R,R} & \cdots \end{pmatrix}$$

- *nrep* is a vector whose elements specify the length of the individual replica.

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

- f is a user defined Tcl function returning a double with first argument a vector which has as many entries as data has columns. If f is given instead of the column, the corresponding derived quantity is analyzed.
- *f_args* are further arguments to f.
- *s_tau* is the estimate $S = \tau/\tau_{\mathrm{int}}$ as explained in section (3.3) of [18]. 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 τ_{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 χ^2 fit of the replica estimates.

The function returns an error message if the windowing failed or if the error in one of the replica is to large.

9 Input / Output

9.1 `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 rsp. double values, which then follow in a whitespace separated list. Such blocks can be read in conveniently using `block_read_data` and written using `block_write_data`.

Sampe C-code doesn't work, as ESPResSo-library has been removed!

9.1.1 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 57) 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.

9.1.2 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.

9.1.3 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`

How is a Tcl-range specified?

Description

Variant (1) writes particle information in a standardized format to *channel*. *what* can be any list of parameters that can be specified in `part` $part_i d$ `print`, except for `bonds`. Note that `id` and `pos` will automatically be added if missing. *range* is a Tcl list of ranges which particles to write. 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*.

9.1.4 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 respecitve random number generators (see section **??** on page ??) 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.

9.1.5 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 84) to *channel*.

9.1.6 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`.

9.1.7 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.

Needs to be rewritten!

Variants (3) and (4) read whole blocks. Variant (3) reads the beginning of one block, checks wether 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.

9.2 Checkpointing

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

9.2.1 Creating a checkpoint

Syntax

| `checkpoint_set` *destination* [num_c*onfigs* [*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.

9.2.2 Reading a checkpoint

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.

9.2.3 Writing a checkpoint 2

Syntax

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

Title!

Clean up. Describe arguments in argument env.

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. *param$_l$ist* 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`. *part$_l$ist* 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.

9.2.4 Writing a checkpoint 3

Syntax

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

Title!

Clean up. Describe arguments in argument env.

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. With this one can set real checkpoints which should reproduce the script-state as precisely as possible.

9.3 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 by VMD to describe the topology of a PDB file. You need the PDB and PSF files for example for IMD.

9.3.1 `writepsf`: Writing the topology

Syntax

`writepsf` *file* [`-molecule`] N_P MPC N_CI N_pS N_nS

Description

Writes the current topology to the file *file* (here, *file* is not a channel, since additional information cannot be written anyway). N_P, MPC and so on are parameters describing a system consisting of equally long charged polymers, counterions and salt. This information is used to set the residue name and can be used to color the atoms in VMD. If you specify `-molecule`, the residue name is taken from the molecule identity of the particle. Of course different kinds of topologies can also be handled by modified versions of `writepsf`.

9.3.2 `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).

9.4 Writing VTF files

There are two commands in ESPResSo that support writing files in the VMD formats VTF, VSF and VCF.[1] The commands can be used to write the structure (`writevsf`) and coordinates (`writevcf`) of the system to a single trajectory file (usually with the extension `.vtf`), or to separate files (extensions `.vsf` and `.vtf`).

9.4.1 `writevsf`: Writing the topology

Syntax

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

Description

Writes a structure block describing the system's structure to the channel given by *channelId*. *channelId* must be an identifier for an open channel such as the return value of an invocation of `open`. The atom ids used in the file are not necessarily identical to ESPResSo's particle ids. To get the atom id used in the vtf file from an ESPResSo particle id, use the command `vtfpid` described below. This makes it easy to write additional structure lines to the file, e.g. to specify the `resname` of particle compounds, like chains. The output of this command can be used for a standalone VSF file, or at the beginning of a trajectory VTF file that contains a trajectory of a whole simulation.

Arguments

- [(`short` | `verbose`)] Specify, whether the output is in a human-readable, but somewhat longer format (`verbose`), or in a more compact form (`short`). The default is `verbose`.

- [`radius` (*radii* | `auto`)] Specify the VDW radii of the atoms. *radii* is either `auto`, or a Tcl-list describing the radii of the different particle types. When the keyword `auto` is used and a Lennard-Jones interaction between two particles of the given type is defined, the radius is set to be $\frac{\sigma_{LJ}}{2}$ plus the LJ shift. Otherwise, the radius 0.5 is substituted. The default is `auto`.
  Example: `writevsf $file radius {0 2.0 1 auto 2 1.0}`

- [`typedesc` *typedesc*] *typedesc* is a Tcl-list giving additional VTF atom-keywords to specify additional VMD characteristics of the atoms of the given type. If no description is given for a certain particle type, it defaults to `name` *`name`* `type` *`type`*, where *name* is an atom name and *type* is the type id.
  Example: `writevsf $file typedesc {0 "name colloid" 1 "name pe"}`

[1] A description of the format and a plugin to read the format in VMD is found in the subdirectory `vmdplugin/` of the ESPResSo source directory.

9.4.2 `writevcf`: Writing the coordinates

Syntax

```
writevcf channelId [( short | verbose )] [( folded | absolute )]
      [pids ( pids | all )]
```

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}`

9.4.3 `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.

9.5 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.

9.5.1 `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 **??** on 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.

9.5.2 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.

9.5.3 Automatically setting up a VMD connection

Syntax

| `prepare_vmd_connection` [*filename* [*wait* [*start*]]]

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 PSF/PDB-files to *filename*.psf and *filename*.pdb (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. When *wait* is provided, ESPResSo will wait for *wait* seconds before it continues, giving VMD some time to start up and connect.

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 `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"
```

9.6 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.

I do not understand this. How does the error look?

For error conditions such as the examples given above, a Tcl error message of the form

tcl_error `background 0` *error_a error_b* `1` *error_c*

is returned. Following possibly a normal Tcl error message, after the background keyword all severe errors are listed node by node, preceeded by the node number. A special error is `<consent>`, which means that one of the slave nodes found exactly the same errors as the master node. This happens mainly during the initialization of the integrate, *e.g.* if the time step is not set. In this case the error message will be

```
background_errors 0 {time_step not set} 1 <consent>
```

In each case, the current action was not fulfilled, and possibly other parts of the internal data also had to be changed to allow ESPResSo to continue, so you should really know what you do if you try and catch these errors.

10 Auxilliary commands

Missing commands: Probably all from `scripts/auxiliary.tcl`**?** `galileiTransformParticles` `system_com_vel`

10.1 Finding particles and bonds

10.1.1 `countBonds`

Syntax

| `countBonds` *particle_list*

Description

Returns a Tcl-list of the complete topology described by *particle_list*, which must have the same format as the output of the command `part` (see section 4.1 on page 25).

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.

10.1.2 `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]`.

10.1.3 `findBondPos`

Syntax

| `findBondPos` *particle$_p$roperty$_l$ist*

Description
Returns the index of the bonds within *particle$_p$roperty$_l$ist*, which is expected to have the same format as `[part` *particle_number*`]`; hence its output is the same as `[findPropPos` *particle$_p$roperty$_l$ist* `bonds]`. If the particle does not have any bonds, `-1` is returned.

10.1.4 `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.

10.2 Additional Tcl math-functions

The following procedures are found in scripts/ABHmath.tcl.

- CONSTANTS
  - `PI`
    returns π with 16 digits precision.
  - `KBOLTZ`
    Returns Boltzmann constant in Joule/Kelvin
  - `ECHARGE`
    Returns elementary charge in Coulomb
  - `NAVOGADRO`
    Returns Avogadro number
  - `SPEEDOFLIGHT`
    Returns speed of light in meter/second
  - `EPSILON0`
    Returns dielectric constant of vaccum in Coulomb$\hat{2}$/(Joule meter)
  - `ATOMICMASS`
    Returns the atomic mass unit u in kilogramms

- MATHEMATICAL FUNCTIONS
  - `sqr <arg>`

    returns the square of *arg*.
  - `min <arg1> <arg2>`

    returns the minimum of *arg1* and *arg2*.
  - `max <arg1> <arg2>`

    returns the maximum of *arg1* and *arg2*.
  - `sign <arg>`

    returns the signum-function of *arg*, namely +1 for *arg* > 0, -1 for < 0, and =0 otherwise.
- RANDOM FUNCTIONS
  - `gauss_random`

    returns random numbers which have a Gaussian distribution
  - `dist_random <dist> [max]`

    returns random numbers in the interval $[0, 1]$ which have a distribution according to the distribution function p(x) *dist* which has to be given as a tcl list containing equally spaced values of p(x). If p(x) contains values larger than 1 (default value of max) the maximum or any number larger than that has to be given *max*. This routine basically takes the function p(x) and places it into a rectangular area ([0,1],[0,max]). Then it uses to random numbers to specify a point in this area and checks wether it resides in the area under p(x). Attention: Since this is written in tcl it is probably not the fastest way to do this!
  - `vec_random [len]`

    returns a random vector of length *len* (uniform distribution on a sphere) This is done by chosing 3 uniformly distributed random numbers $[-1, 1]$ If the length of the resulting vector is <= 1.0 the vector is taken and normalized to the desired length, otherwise the procedure is repeated until succes. On average the procedure needs 5.739 random numbers per vector. (This is probably not the most efficient way, but it works!) Ask your favorit mathematician for a proof!
  - `phivec_random <v> <phi> [len]`

    return a random vector at angle *phi* with *v* and length *len*
- PARTICLE OPERATIONS

  Operations involving particle positions. The parameters *pi* can either denote the particle identity (then the particle position is extracted with the The part command command) or the particle position directly When the optional *box* parameter for minimum image conventions is omited the functions use the the `setmd box_l` command.

- `bond_vec <p1> <p2>`

  Calculate bond vector pointing from particles *p2* to *p1* return = (*p1*.pos - *p2*.pos)
- `bond_vec_min <p1> <p2> [box]`

  Calculate bond vector pointing from particles *p2* to *p1* return = MinimumImage(*p1*.pos - *p2*.pos)
- `bond_length <p1> <p2>`

  Calculate bond length between particles *p1* and *p2*
- `bond_length_min <p1> <p2> [box]`

  Calculate minimum image bond length between particles *p1* and *p2*
- `bond_angle <p1> <p2> <p3> [type]`

  Calculate bond angle between particles *p1*, *p2* and *p3*. If *type* is "r" the return value is in radiant. If it is "d" the return value is in degree. The default for *type* is "r".
- `bond_dihedral <p1> <p2> <p3> <p4> [type]`

  Calculate bond dihedral between particles *p1*, *p2*, *p3* and *p4* If *type* is "r" the return value is in radiant. If it is "d" the return value is in degree The default for *type* is "r".
- `part_at_dist <p> <dist>`

  return position of a new particle at distance *dist* from *p* with random orientation
- `part_at_angle <p1> <p2> <phi> [len]`

  return position of a new particle at distance *len* (default=1.0) from *p2* which builds a bond angle *phi* for (*p1*, *p2*, p-new)
- `part_at_dihedral <p1> <p2> <p3> <theta> [phi] [len]`

  return position of a new particle at distance *len* (default=1.0) from *p3* which builds a bond angle *phi* (default=random) for (*p2*, *p3*, p-new) and a dihedral angle *theta* for (*p1*, *p2*, *p3*, p-new)

- INTERACTION RELATED

  Help functions related to interactions implemented in ESPResSo.

  - `calc_lj_shift <lj_sigma> <lj_cutoff>`

    returns the value needed to shift the Lennard Jones potential to zero at the cutoff.

- VECTOR OPERATIONS

  A vector *v* is a tcl list of numbers with an arbitrary length Some functions are provided only for three dimensional vectors. corresponding functions contain 3d at the end of the name.

  - `veclen <v>`

return the length of a vector

- `veclensqr <v>`
  return the length of a vector squared
- `vecadd <a> <b>`
  add vector *a* to vector *b*: return = (*a*+*b*)
- `vecsub <a> <b>`
  subtract vector *b* from vector *a*: return = (*a*-*b*)
- `vecscale <s> <v>`
  scale vector *v* with factor *s*: return = (*s***v*)
- `vecdot_product <a> <b>`
  calculate dot product of vectors *a* and *b*: return = (*a*.*b*)
- `veccross_product3d <a> <b>`
  calculate the cross product of vectors *a* and *b*: return = (*a* x *b*)
- `vecnorm <v> [len]`
  normalize a vector to length *len* (default 1.0)
- `unitvec <p1> <p2>`
  return unit vector pointing from position *p1* to position *p2*
- `orthovec3d <v> [len]`
  return orthogonal vector to *v* with length *len* (default 1.0) This vector does not have a random orientation in the plane perpendicular to *v*
- `create_dihedral_vec <v1> <v2> <theta> [phi] [len]`
  create last vector of a dihedral (*v1*, *v2*, res) with dihedral angle *theta* and bond angle (*v2*, res) *phi* and length *len* (default 1.0). If *phi* is ommited or set to rnd then *phi* is assigned a random value between 0 and 2 Pi.

- TCL LIST OPERATIONS
  - `average <list>`
    Returns the avarage of the provided *list*
  - `list_add_value <list> <val>`
    Add *val* to each element of *list*
  - `flatten <list>`
    flattens a nested *list*
  - `list_contains <list> <val>`
    Checks wether *list* contains *val*. returns the number of occurences of *val* in *list*.

- REGRESSION
  - `LinRegression <l>`
    l is a list x1 y1 x2 y2 ... of points. LinRegression returns the least-square linear fit a*x+b and the standard errors da and db.

– `LinRegressionWithSigma <l>`

l is a list x1 y1 s1 x2 y2 s2 ... of points with standard deviations. LinRegression returns the least-square linear fit a*x+b plus the standard errors da and db, cov(a,b) and chi.

10.2.1 `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

10.2.2 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 ¡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)!

- 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!

10.3 Checking for features of ESPResSo

In an ESPResSo-Tcl-script, you can get information whether or not one or some of the features are compiled into the current program with help of the following Tcl-commands:

- `code_info`

  provides information on the version, compilation status and the debug status of the used code. It is highly recommended to store this information with your simulation data in order to maintain the reproducibility of your results. Exemplaric output:

  ```
  ESPRESSO: v1.5.Beta (Neelix), Last Change: 23.01.2004
  { Compilation status { PARTIAL_PERIODIC } { ELECTROSTATICS }
  { EXTERNAL_FORCES } { CONSTRAINTS } { TABULATED }
  { LENNARD_JONES } { BOND_ANGLE_COSINE } }
  { Debug status { MPI_CORE FORCE_CORE } }
  ```

- `has_feature <feature> ...`

  tests, if *feature* is compiled into the ESPResSo kernel. A list of possible features and their names can be found here.

- `require_feature <feature> ...`

  tests, if *feature* is feature is compiled into the ESPResSo kernel, will exit the script if it isn't and return the error code 42. A list of possible features and their names can be found here.

11 External package: mbtools

mbtools[1] is a set of tcl packages for setting up, analyzing and running simulations of lipid membrane systems.

mbtools comes with a basic set of tabulated forces and potentials for lipid interactions and some example scripts to help explain the syntax of the commands. If you make use of mbtools or of these potentials please acknowledge us with a citation to:

* Cooke, I. R., Kremer, K. and Deserno, M. (2005): Tunable, generic model for fluid bilayer membranes, Phys. Rev. E. 72 - 011506

11.1 Introduction

mbtools is located in the folder `Espresso/packages/mbtools`.

One of the main features of mbtools is the ability to easily create initial lipid configurations with interesting geometries. These include flat membranes, cylinders, spheres, toroids, and randomly distributed gases. Each of these shapes is referred to as a geometry and any number of geometries can be combined in a single simulation. Once the geometry has been chosen the user specifies the molecules which should be placed in this geometry. For example one could choose sphere as a geometry and then define two different lipids and/or a protein to be placed on the sphere. Within reason (e.g. size restrictions) it should be possible to use any mixture of known molecule types on any geometry. The molecule types available at present include proteins, lipids of any length, and spherical colloids.

mbtools includes several miscellaneous utility procedures for performing tasks such as warmup, setting tabulated interactions, designating molecules to be trapped and a variety of topology related sorting or data analysis functions.

The analysis part of the mbtools package is designed to wrap together all the analysis for a simulation into a single simple interface. At the beginning of the simulation the user specifies which analyses should be performed by appending its name and arguments to a variable, `analysis_flags`. After the analysis is setup one can then simply call `do_analysis` to perform all the specified proceedures. Analysis will store a data value each time `do_analysis` is called. Then when a call to `print_averages` is made the average of all stored values is printed to a file and the store of values is reset to nil.

[1] This documentation was written by Ira R. Cooke and published on his website. It has been transcripted by Tristan Bereau.

11.2 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://www.espresso.mpg.de`.

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 (You might have to use `Espresso_bin` when running on multiple processors). Obviously you will need to have the `$ESPRESSO_SOURCE` and `$PLATFORM` environment variables set for it to work. After this executable the relative paths to two tcl scripts are given. The first of these `main.tcl` is given as an argument to espresso and is therefore interpreted first by the espresso tcl interpreter. The second tcl script `simplebilayer.tcl` is in turn passed as an argument to `main.tcl`.

Why separate the tcl commands into two files ?

This is really a matter of preference but if we keep all of the key commands and complex coding in a single file `main.tcl` and delegate simple parameter setting to a separate file it tends to be much easier to manage large numbers of jobs with regularly changing requirements. Regardless of your personal preferences, the important point to note is that all of the important commands are contained in `main.tcl` and you should probably start there to get an understanding for how mbtools works.

Running the simplebilayer example should produce a directory called `simplebilayer` which contains the output from your simulation. To view the results cd to the simplebilayer directory and look at the contents. The directory contains many files including:

- The configurations generated during warmup : `warm.*.gz`
- pdb files corresponding to warmup configurations : `warm.vmd*.gz`
- The configurations generated during the main run : `simplebilayer.*.gz`
- pdb files corresponding to main run configs : `simplebilayer.vmd*.gz`
- The most recently generated checkpoint file : `checkpoint.latest.gz`
- A directory containing the second most recent checkpoint file : `checkpoint_bak`
- A file containing the topology of the system : `simplebilayer.top`
- The original parameter file with which you ran the simulation : `simplebilayer.tcl`

- A original parameter file with which you ran the simulation : `simplebilayer.tcl`
- Files containing analysis output for example : `time_vs_boxl_tmp`
- Force and energy tables : `*.tab`
- VMD script for visualising the warmup : `warm_animation.script`
- VMD script for visualising the main trajectory : `vmd_animation.script`

To visualise your results using the vmd scripts you need to make sure that you have vmd installed properly and that you have the special vmd procedures used by the espresso team (i.e. support for the loadseries command). Then you can visualise by typing:

```
vmd -e vmd_animation.script
```

11.3 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.

11.3.1 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 11.5)
- **moltypes* A list of molecule types (see documentation in 11.5)
- **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

11.4 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.

11.4.1 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.

11.4.2 Available analysis routines

```
boxl  :  -verbose  : output ||  time_vs_boxl
```

Simply obtains the box dimensions from ESPResSo.

```
clusters : -alipid.arg -verbose : output || time_vs_clust,
           sizehisto.[format %05d $time]
```

- alipid [1.29] Value for the area per lipid to be used in a making a rough calculation of the area of clusters

Calls the espresso command `analyze aggregation` which groups molecules in the system into aggregates. Output to `time_vs_clust` is: maximum cluster size, minimum cluster size, average size of clusters including those of size 2 or greater, standard deviation of clusters including those of size 2 or greater, number of clusters of size 2 or greater, total average cluster size, total cluster size standard deviation, total number of clusters, length of the interface between clusters, standard deviation of the interface length, number of clusters for which length was calculate.

Additionally, at each call of `print_averages` the complete size histogram is printed to a file with the formatted name `sizehisto.[format %05d $time]`.

```
density_profile  :  -nbins.arg -hrange.arg -beadtypes.arg
                   -colloidmoltypes.arg -r.arg -nogrid
                   -verbose  : output ||  av_zprof
```

- *nbins* [100] Number of slices into which the height range is divided for the purpose of calculating densities

- *hrange* [6] The maximum vertical distance from the bilayer midplane for which to calculate densities. Note that the complete vertical range is therefore 2*varhrange
- *beadtypes* [0] A tcl list of the bead types for which to calculate a density profile
- *colloidmoltypes* [] A tcl list of molecule types identifying the molecules which are colloids in the system. The default value is a null list
- *r* [0] A tcl list of sphere radii corresponding to the radii for each colloid type in the system. If this is non-zero the density profile will be calculated in spherical shells about the colloids in the system identified via colloidmoltypes or if colloidmoltypes is not set then the system center of mass is assumed for the colloid/vesicle center
- *nogrid* If this is set a grid mesh will not be used to refine the density profile calculation by taking into account vertical differences between mesh points

Calculates the number density of each of the beadtypes given in beadtypes as a function of the vertical distance from the bilayer midplane. Lipids are also sorted according to their orientation and assigned to upper or lower leaflets accordingly. Thus for a system with 3 beadtypes we would obtain 6 columns of output corresponding to 0 (lower) 1 (lower) 2 (lower) 2 (upper) 1 (upper) 0 (upper) where the number refers to the bead type and upper or lower refers to the bilayer leaflet.

```
energy : -verbose : output || time_vs_energy
```

Obtains the internal energies of the system from the `analyze energy` command of ESPResSo.

```
flipflop : -verbose : output || time_vs_flip
```

Makes a call to the `analyze get_lipid_orients` command of ESPResSo and compares this with a reference set of lipid orients obtained at the start of the simulation with `setup_analysis`. Based on this comparison the number of lipids which have flipped from their original positions is calculated

```
fluctuations : -verbose : output || powav.dat
```

Routine for calculating the power spectrum of height and thickness fluctuations for a flat bilayer sheet. Uses the `modes_2d` routine in ESPResSo to calculate the height and thickness functions and perform the fft. See the documentation in the file `fluctuations.tcl` for detail on what is calculated and how to obtain a stiffness value from the resulting output. Note that this routine causes a crash if it detects a large hole in the bilayer.

```
localheights : -range.arg -nbins.arg -rcatch.arg -verbose :
               output || av_localh
```

- *range* [1.0] Range of local height deviations over which to bin

- *nbins* [100] Number of slices to divide up the height range into for the purposes of creating a profile
- *rcatch* [1.9] The distance about a single lipid to use a starting value for finding the 6 closest neighbours

For each lipid we calculate its 6 nearest neighbours and then calculate the height difference between the central lipid and these neighbours. Taking these 6 values for each lipid we then create a histogram of number densities as a function of the height difference.

```
localorients : -range.arg -nbins.arg -verbose : output || av_localo
```

- range [1.0] Range of orientation deviations to consider
- nbins [100] Number of bins to use for histogram

Calculates the projection of the lipid orientation vector onto the *xy* plane for each lipid and then bins the absolute values of these vectors.

```
orient_order : -verbose : output || time_vs_oop
```

Calculates the orientational order parameter S for each lipid through a call to the espresso command `analyze lipid_orient_order`.

```
stress_tensor : -verbose : output || time_vs_stress_tensor
```

Calculates all 9 elements of the pressure tensor for the system through a call to the espresso command `analyze stress_tensor`

```
pressure : -verbose : output || time_vs_pressure
```

Calculates the isotropic pressure through a call to `analyze pressure`. Results are printed as a list of the various contributions in the following order: *p_inst*, *total*, *ideal*, *FENE*, *harmonic*, *nonbonded*. Where *p_inst* is the instantaneous pressure obtained directly from the barostat.

```
stray : -verbose : output || time_vs_stray
```

Calculates the number of stray lipids based on a call to `analyze get_lipid_orients`.

11.4.3 Adding a new routine

To add a new analysis routine you should create a new file called `myanalysis.tcl` which will contain all of your code. At the top of this file you should declare a namespace for your analysis code and include all of the internal variables inside that namespace as follows;

```
namespace eval ::mbtools::analysis::myanalysis {
    variable av_myresult
    variable av_myresult_i
    variable f_tvsresult
    variable verbose

    namespace export setup_myanalysis
    namespace export analyze_myanalysis
    namespace export printav_myanalysis
    namespace export resetav_myanalisis
}
```

Import your new file into the analysis package by adding a line like the following to the `analysis.tcl` file.

```
source [file join [file dirname [info script]] myanalysis.tcl]
```

You then need to implement the following essential functions within your new namespace.

- `::mbtools::analysis::myanalysis::setup_myanalysis { args }`

  Typically you would use this function to initialise variables and open files.

  Called by `::mbtools::analysis::setup_analysis`. Arguments are allowed.

- `::mbtools::analysis::myanalysis::printav_myanalysis { void }`

  This function should print results to a file.

  Called by `::mbtools::analysis::print_averages`. Arguments are not allowed.

- `::mbtools::analysis::myanalysis::analyze_myanalysis { void }`

  This function performs the actual analysis and should update the storage and averaging variables. Called by `::mbtools::analysis::do_analysis`. Arguments are not allowed.

- `::mbtools::analysis::myanalysis::resetav_myanalysis { void }`

  This function should update averages and reset variables accordingly depending on your requirements.

  Called by `::mbtools::analysis::reset_averages`. Arguments are not allowed.

If any of these functions is not implemented the program will probably crash.

11.5 System generation

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

11.5.1 Basic commands

```
::mbtools::system_generation::setup_system  :  [system_specs]
          [iboxl] [moltypes]
```

- `system_specs` This is a list of structures called system specifications. Each such system specification in turn should be a list consisting of a geometry and a list detailing the number of each molecule type i.e.

  ```
  set system_spec { geometry n_molslist }
  ```

  The *geometry* should be specified as a list with two elements. The first element should be a string "geometry" identifying this list as a geometry. The second element is a string containing the name of a geometry type *mygeometry* followed by arguments to be passed to the routine `create_mygeometry`.

  The *n_molslist* should be specified as a list with two elements. The first element should be a string "n_molslist" identifying this list as an n_molslist. The second element is a list each element of which specifies a molecule type and the number of such molecules.

- *boxl* A list containing the lengths of each of the box side lengths.

- *moltypes* A list, each element of which specifies a molecule type and type information. The exact format and requirements of this list are detailed for each molecule separately (see below for a list of molecule types and their requirements) however regardless of mol type the first two elements of the list must be a *moltypeid* and a string specifying the moltype respectively.

Sets up the system including generating topologies and placing molecules into specified geometries. Each geometry and list of molecules to be placed into that geometry are grouped into a system spec.

Example:

The following code sets out the molecule types to be used in the simulation by setting a list called *moltypes*. In this case two different lipid types are setup and assigned to moltypeids 0 and 1 respectively. Moltype 0 will consist of three beads per lipid, the first of which is of atomtype 0 and the second and third of which are of atomtype 1. Bonds in the lipid will be of type 0 and 1 respectively.(see the `::mbtools::system_generation::place_lipid_linear` function for further details).

```
set moltypes [list { 0 lipid { 0 1 1 } { 0 1 } }
                   { 1 lipid { 0 2 2 2 } { 0 2 } } ]
```

We then construct system specs for a flat bilayer and a spherical bilayer and group these into a *system_specs* list.

First the spherical *system_specs*

```
set geometry { geometry  "sphere -shuffle -c { 0.0 0.0 15.0 } " }
set n_molslist { n_molslist {  { 0 1000 } } }
lappend spherespec $geometry
lappend spherespec $n_molslist
```

The flat system_spec

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

Now group together the *system$_s$pecs* into a master list

```
lappend system_specs $spherespec
lappend system_specs $bilayerspec
```

Make the call to `setup_system`

```
::mbtools::system_generation::setup_system $system_specs
         [setmd box_l] $moltypes

::mbtools::system_generation::get_trappedmols :
```

returns the internal list variable *trappedmols* which keeps track of all molecules that have been trapped by their center of mass. This function should be called after setup and would then typically be passed to the function `::mbtools::utils:trap_mols`.

```
::mbtools::system_generation::get_userfixedparts :
```

returns the internal list variable *userfixedparts* which keeps track of all particles that have been fixed in position during the setup. This is useful for later releasing particles after warmup routines have been completed.

```
::mbtools::system_generation::get_middlebead :
```

returns the internal variable *middlebead*.

11.5.2 Available geometries

```
flat  :  -fixz -bondl.arg -crystal -half
```

- *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.

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 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.

`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 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 -shuffle -bondl.arg
```

- *exclude.arg* [] an exclusion 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 no molecules will be placed such that their centers of mass are within the zone.

```
readfile : -ignore.arg -f.arg -t.arg
```

- *ignore.arg* [] particle properties to be ignored during the file read.
- *f.arg* [] The file containing the configuration to be used for setup. Must be an espresso blockfile with box length, particle and bonding information.
- *t.arg* [] The topology file corresponding to the file to be read.
- *tol.arg* [0.000001] Tolerance for comparison of box dimensions.

Use particle positions contained in a file to initialise the locations of particles for a particular geometry. The box dimensions in the file and those set by the user are compared and an error is returned if they are not the same to within a tolerance value of *tol*. Even though we read from a file we also generate a topology from the n_m*olslist* and this topology is compared with the topology that is read in to check if the number of particles are the same.

```
singlemol : -c.arg -o.arg -trapflag.arg -ctrap.arg
            -trapspring.arg -bondl.arg
```

- *c.arg* [0.0 0.0 0.0] The molecule center. Exactly what this means depends on the molecule type.
- *o.arg* [0.0 0.0 1.0] The orientation vector for the molecule. This is also molecule type dependent
- *trapflag.arg* [0 0 0] Set this optional argument to cause a molecule to be trapped by its center of mass. You should give three integers corresponding to each of the three coordinate axes. If a value of 1 is given then motion in that axis is trapped.

- *ctrap.arg* [""] Set this optional argument to the central point of the trap. This works much like an optical trap in that molecules will be attracted to this point via a simple harmonic spring force
- *trapspring.arg* [20] The spring constant for the trap potential (harmonic spring).

Simply place a single molecule at the desired position with the desired orientation.

11.5.3 Adding a new geometry

To create a routine for setting up a system with a new type of geometry *mygeom*. Start by creating a new file `mygeom.tcl` inside the `system_generation` directory. The new file should declare a new namespace *mygeom* as a sub namespace of `::mbtools::system_generation` and export the proceedure `create_mygeom`. Thus your `mygeom.tcl` file should begin with the lines

```
namespace eval ::mbtools::system_generation::mygeom {
  namespace export create_mygeom
}
```

Import your new file into the `system_generation` package by adding a line like the following to the `system_generation.tcl` file

```
source [file join [file dirname [info script]] mygeom.tcl]
```

You then need to implement the `create_mygeom` proceedure within your new namespace as follows

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

11.5.4 Available molecule types

```
lipid  : typeinfo :  { moltypeid "lipid" particletypelist
         bondtypelist }
```

- *particletypelist* A list of the particle types for each atom in the lipid. The particles are placed in the order in which they appear in this list.
- *bondtypelist* A list of two *bondtypeid*s. The first id is used for bonds between consecutive beads in the lipid. The second *bondtypeid* defines the pseudo bending potential which is a two body bond acting across beads separated by exactly one bead.

Places atoms in a line to create a lipid molecule.

```
hollowsphere : typeinfo : { moltypeid "hollowsphere"
               sphereparticlelist bondtype natomsfill }
```

- *sphereparticlelist* A list of the particle types for each atom in the hollowsphere. The atoms that make up the outer shell must be listed first followed by the atoms that make up the inner filling.
- *bondtype* The typeid for bonds linking atoms in the outer shell.
- *natomsfill* Number of filler atoms. The atom types for these will be obtained from the last *natomsfill* in the *sphereparticlelist*.

Creates a sphere of beads arranged such that they have an approximate spacing of *bondl* and such that they optimally cover the sphere. The optimal covering is obtained using the `icover` routines which are copyright R. H. Hardin, N. J. A. Sloane and W. D. Smith, 1994, 2000. Thus the routine will only work if you have installed icover and if you can successfully run it from the command line in the directory that you started your espresso job. These routines are serious overkill so if anybody can think of a nice simple algorithm for generating a covering of the sphere let us know.

```
protein : typeinfo : { moltypeid "protein" particletypelist
          bondtypelist }
```

- *particletypelist* A list of the particle types for each atom in the protein.
- *bondtypelist* A list of bondtypeids.

Create a protein molecule.

```
spanlipid : typeinfo : { moltypeid "protein" particletypelist
            bondtypelist }
```

- *particletypelist* A list of the particle types for each atom in the lipid. Since this is a spanning lipid the first and last elements of this list would typically be head beads.
- *bondtypelist* A list of two *bondtypeid*s with the same meaning as explained above for standard lipids.

Create a lipid which spans across the bilayer.

11.5.5 Adding a new molecule type

To add a new molecule type you need to define a proceedure which determines how the atoms that make up the molecule should be placed. This proc will live directly in the `::mbtools::system_generation` namespace. Examples can be found in `place.tcl`.

In order to register your new molecule type to allow placement in any geometry you need to add a call to it in the function `::mbtools::system_generation::placemol`. Make sure that all arguments to your `place_mymolecule` routine are included in this function call.

11.6 Utils

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

11.6.1 Setup commands

```
::mbtools::utils::setup_outputdir  :  [outputdir] -paramsfile.arg
          -tabdir.arg -tabnames.arg -startf.arg -ntabs.arg
```

- *outputdir* Complete path of the directory to be setup. At least the parent of the directory must exist
- *paramfile* [] Name of a file to be copied to the output directory
- *tabdir* [] Full path name of the directory where forcetables are kept
- *tabnames* [] Complete list of forcetables to be used in the simulation. These will be copied to the output directory

This routine is designed to setup a directory for simulation output. It copies forcetables and the parameter file to the directory after creating it if necessary.

```
::mbtools::utils::read_startfile : [file]
```

- *file* Complete path of the file to be read. Should be an espresso blockfile.

Read in particle configuration from an existing file or simulation snapshot

```
::mbtools::utils::read_checkpoint : [dir]
```

- *dir* Directory containing the checkpoint file which must be called `checkpoint.latest.gz`.

Read in a checkpoint and check for success. Warn if the checkpoint does not exist.

```
::mbtools::utils::read_topology : [file]
```

- *file* Complete path of the file that contains the topology information.

Read in the topology from a file and then execute the `analyze set "topo_part_sync"` command of ESPResSo.

```
::mbtools::utils::set_topology : [topo]
```

- *topo* A valid topology.

Set the given topology and then execute the `analyze set "topo_part_sync"` command of ESPResSo.

`::mbtools::utils::set_bonded_interactions : [bonded_parms]`

- *bonded$_p$arms* A list of bonded interactions. Each element of this list should contain all the appropriate arguments in their correct order for a particular call to the espresso `inter` command. See the espresso `inter` command for a list of possible bonded interactions and correct syntax.

Set all the bonded interactions.

`::mbtools::utils::set_nb_interactions : [nb_parms]`

- *nb_parms* A list of interactions. Each element of this list should contain all the appropriate arguments in their correct order for a particular call to the espresso `inter` command. See the espresso `inter` command for a list of possible non-bonded interactions and correct syntax.

Set all the bonded interactions.

`::mbtools::utils::init_random : [n_procs]`

- *n_procs* The number of processors used in this job.

Initialize the random number generators on each processor based on the current time with a fixed increment to the time seed used for each proc.

```
::mbtools::utils::initialize_vmd : [flag] [outputdir]
          [ident] -extracommands.arg
```

- *flag* Depending on the value of this parameter initialize vmd to one of its possible states:
  - interactive : VMD is started and a connection to espresso established for immediate viewing of the current espresso process. With some luck this might even work sometimes! If VMD doesn't get a proper connection to espresso then it will crash.
  - offline : Just constructs the appropriate `psf` and `vmd_animation.script` files and writes them to the output directory so that `pdb` files generated with writepdb can be viewed with `vmd -e vmd_animation.script`.
  - default : Any value other than those above for flag will just result in vmd not being initialized.
- *outputdir* The directory where vmd output will be written.
- *ident* A basename to be be given to vmd files.
- *extracommands* [] A list of strings each of which will be written to the end of the `vmd_animationscript`. Use this to give additional commands to vmd.

Prepare for vmd output.

11.6.2 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.

11.6.3 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.

11.6.4 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::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.

11.6.5 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* An exclusion zone. This will be a tcl list. The first element of the list must be a string with the name of the exclusion zone type and subsequent elements will be further information about the exclusion zone. Available zones are:
  - *sphere* : center zone

Determines whether the point at *pos* is outside the exclusion zone. 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.

11.7 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.

11.7.1 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.

11.7.2 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"
```

12 Under the hood

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

12.1 Internal particle organization

Since basically all major parts of the main MD integration have to access the particle data, efficient access to the particle data is crucial for a fast MD code. Therefore the particle data needs some more elaborate organisation, which will be presented here. A particle itself is represented by a structure (Particle) consisting of several substructures (e. g. ParticlePosition, ParticleForce or ParticleProperties), which in turn represent basic physical properties such as position, force or charge. The particles are organised in one or more particle lists on each node, called Cell cells. The cells are arranged by several possible systems, the cellsystems as described above. A cell system defines a way the particles are stored in ESPResSo, i. e. how they are distributed onto the processor nodes and how they are organised on each of them. Moreover a cell system also defines procedures to efficiently calculate the force, energy and pressure for the short ranged interactions, since these can be heavily optimised depending on the cell system. For example, the domain decomposition cellsystem allows an order N interactions evaluation.

Technically, a cell is organised as a dynamically growing array, not as a list. This ensures that the data of all particles in a cell is stored contiguously in the memory. The particle data is accessed transparently through a set of methods common to all cell systems, which allocate the cells, add new particles, retrieve particle information and are responsible for communicating the particle data between the nodes. Therefore most portions of the code can access the particle data safely without direct knowledge of the currently used cell system. Only the force, energy and pressure loops are implemented separately for each cell model as explained above.

The domain decomposition or link cell algorithm is implemented in ESPResSo such that the cells equal the ESPResSo cells, i. e. each cell is a separate particle list. For an example let us assume that the simulation box has size $20 \times 20 \times 20$ and that we assign 2 processors to the simulation. Then each processor is responsible for the particles inside a $10 \times 20 \times 20$ box. If the maximal interaction range is 1.2, the minimal possible cell size is 1.25 for 8 cells along the first coordinate, allowing for a small skin of 0.05. If one chooses only 6 boxes in the first coordinate, the skin depth increases to 0.467. In this example we assume that the number of cells in the first coordinate was chosen to be 6 and that the cells are cubic. ESPResSo would then organise the cells on each node in a $6 \times 12 \times 12$ cell grid embedded at the centre of a $8 \times 14 \times 14$ grid. The additional

cells around the cells containing the particles represent the ghost shell in which the information of the ghost particles from the neighbouring nodes is stored. Therefore the particle information stored on each node resides in 1568 particle lists of which 864 cells contain particles assigned to the node, the rest contain information of particles from other nodes.a

Classically, the link cell algorithm is implemented differently. Instead of having separate particle lists for each cell, there is only one particle list per node, and a the cells actually only contain pointers into this particle list. This has the advantage that when particles are moved from one cell to another on the same processor, only the pointers have to be updated, which is much less data (4 rsp. 8 bytes) than the full particle structure (around 192 bytes, depending on the features compiled in). The data storage scheme of ESPResSo however requires to always move the full particle data. Nevertheless, from our experience, the second approach is 2-3 times faster than the classical one.

To understand this, one has to know a little bit about the architecture of modern computers. Most modern processors have a clock frequency above 1GHz and are able to execute nearly one instruction per clock tick. In contrast to this, the memory runs at a clock speed around 200MHz. Modern double data rate (DDR) RAM transfers up to 3.2GB/s at this clock speed (at each edge of the clock signal 8 bytes are transferred). But in addition to the data transfer speed, DDR RAM has some latency for fetching the data, which can be up to 50ns in the worst case. Memory is organised internally in pages or rows of typically 8KB size. The full 2×200 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.

13 Getting involved

- What to do when you want to become involved
- How to submit a bug report
- Reference to developer's guide

A ESPResSo quick reference

`part` *pid* [`pos` *x y z*] [`type` *typeid*] [`v` *vx vy vz*] [`f` *fx fy fz*] [`bond` *bondid pid2 ...*] [`q` *charge*] [1] [`quat` *q1 q2 q3 q4*] [2] [`omega` *x y z*] [2] [`torque` *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] Required features: [1] `ELECTROSTATICS` [2] `ROTATION` [3] `EXTERNAL_FORCES` [4] `EXCLUSION` [5] `MASS` [6] `DIPOLES`	25
`part` *pid* `print` [(`id` | `pos` | `type` | `folded_position` | `type` | `q` | `v` | `f` | `fix` | `ext_force` | `bond` | `connections` [*range*])]... `part`	26
`part` *pid* `delete` `part deleteall`	27
`part auto_exclusions` [*range*] `part delete_exclusions`	27
`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` $typeid_{\text{neutral}}$ [$typeid_{\text{charged}}$]] [`bond` *bondid*] [`angle` ϕ [θ [*x y z*]]] [`constraints`] [2] Required features: [1] `ELECTROSTATICS` [2] `CONSTRAINTS`	28
`counterions` *N* [`start` *pid*] [`mode` (`SAW` | `RW`) [*shield* [$try_{\max}$]]] [`charge` *val*] [1] [`type` *typeid*] Required features: [1] `ELECTROSTATICS`	30
`salt` N_+ N_- [`start` *pid*] [`mode` (`SAW` | `RW`) [*shield* [$try_{\max}$]]] [`charges` val_+ [val_-]] [1] [`types` $typeid_+$ [$typeid_-$]] [`rad` *r*] Required features: [1] `ELECTROSTATICS`	30
`diamond` *a bond_length monomers_per_chain* [`counterions` N_{CI}] [`charges` val_{node} val_{monomer} val_{CI}] [1] [`distance` d_{charged}] [1] [`nonet`] Required features: [1] `ELECTROSTATICS`	31
`icosaeder` *a monomers_per_chain* [`counterions` N_{CI}] [`charges` val_{monomers} val_{CI}] [1] [`distance` d_{charged}] [1] Required features: [1] `ELECTROSTATICS`	31
`crosslink` *num_polymer monomers_per_chain* [`start` *pid*] [`catch` r_{catch}] [`distLink` *link_dist*] [`distChain` *chain_dist*] [`FENE` *bondid*] [`trials` $try_{\max}$]	32

`constraint wall normal` n_x n_y n_z `dist` d `type` id	33
`constraint sphere center` c_x c_y c_z `radius` rad `direction` $direction$ `type` id	
`constraint cylinder center` c_x c_y c_z `axis` n_x n_y n_z `radius` rad `length` $length$ `direction` $direction$ `type` id	
`constraint maze nsphere` n `dim` d `sphrad` r_s `cylrad` r_c `type` id	
`constraint pore center` c_x c_y c_z `axis` n_x n_y n_z `radius` rad `length` $length$ `type` id	
`constraint rod center` c_x c_y `lambda` $lambda$ [1]	
`constraint plate height` h `sigma` $sigma$ [1]	
`constraint ext_magn_field` f_x f_y f_z [2,3]	
`constraint plane cell` x y z `type` id	
Required features: `CONSTRAINTS` [1]`ELECTROSTATICS` [2]`ROTATION` [3]`DIPOLES`	
`constraint delete` [num]	34
`constraint force` n	34
`constraint` [num]	35
`inter`	36
`inter` $type1$ $type2$ `lennard-jones` ϵ σ r_{cut} c_{shift} r_{off} [r_{cap} r_{min}]	36
Required features: `LENNARD_JONES`	
`inter` $type1$ $type2$ `lj-gen` ϵ σ r_{cut} c_{shift} r_{off} e_1 e_2 b_1 b_2	37
Required features: `LENNARD_JONES_GENERIC`	
`inter` $type1$ $type2$ `lj-cos` ϵ σ r_{cut} r_{off}	37
`inter` $type1$ $type2$ `lj-cos2` ϵ σ r_{off} ω	
Required features: `LJCOS` `LJCOS2`	
`inter` $type1$ $type2$ `lj-angle` ϵ σ r_{cut} $b1_a$ $b1_b$ $b2_a$ $b2_b$	38
Required features: `LJ_ANGLE`	
`inter` $type1$ $type2$ `smooth-step` σ_1 n ϵ k_0 σ_2 r_{cut}	38
Required features: `SMOOTH_STEP`	
`inter` $type1$ $type2$ `bmhtf-nacl` A B C D σ r_{cut}	39
Required features: `BMHTF_NACL`	
`inter` $type1$ $type2$ `morse` ϵ α r_{min} r_{cut}	40
Required features: `MORSE`	
`inter` $type1$ $type2$ `buckingham` A B C D r_{cut} r_{discont} ϵ_{shift}	40
Required features: `BUCKINGHAM`	
`inter` $type1$ $type2$ `soft-sphere` a n r_{cut} r_{offset}	41
Required features: `SOFT_SPHERE`	
`inter` $type1$ $type2$ `gay-berne` ϵ_0 σ_0 r_{cutoff} $k1$ $k2$ μ ν	41
Required features: `ROTATION`	
`inter` $type1$ $type2$ `tabulated` $filename$	41
Required features: `TABULATED`	

`inter` *type1 type2* `tunable_slip` T γ_L r_{cut} δt v_x v_y v_z	42
Required features: `TUNABLE_SLIP`	
`inter ljforcecap` ($F_{\max}$ | `individual`)	43
`inter morseforcecap` ($F_{\max}$ | `individual`)	
`inter buckforcecap` ($F_{\max}$ | `individual`)	
`inter tabforcecap` ($F_{\max}$ | `individual`)	
Required features: `LENNARD_JONES MORSE BUCKINGHAM TABULATED`	
`inter` *bondid* `fene` K $\Delta r_{\max}$ [r_0]	44
`inter` *bondid* `harmonic` K R	44
`inter` *bondid* `subt_lj` *reserved* R	45
`inter` *bondid* `rigid_bond` *constrained*$_b$*ond*$_d$*istance positional*$_t$*olerance velocity*$_t$*olerance*	45
`inter` *bondid* `angle` K [ϕ_0]	46
Required features: `BOND_ANGLE_HARMONIC BOND_ANGLE_COSINE or BOND_ANGLE_COSSQUARE`	
`inter` *bondid* `dihedral` n K p	46
`inter` *bondid* `tabulated bond` *filename*	46
`inter` *bondid* `tabulated angle` *filename*	
`inter` *bondid* `tabulated dihedral` *filename*	
`inter` *bondid* `virtual_bond`	47
`inter coulomb 0.0`	48
`inter coulomb` [l_B *method*] [*parameters*]	
`inter coulomb`	
`inter coulomb p3m` r_{cut} *mesh cao alpha*	49
`inter magnetic p3m` r_{cut} *mesh cao alpha*	
`inter` (`coulomb` | `magnetic`) `p3m` (`tune` | `tunev2`) `accuracy` *accuracy* [`r_cut` r_{cut}] [`mesh` *mesh*] [`cao` *cao*] [`alpha` α]	49
`inter coulomb dh` κ r_{cut}	50
`inter coulomb mmm2d` *maximal_pairwise_error* [*fixed_far_cutoff*] [`dielectric` ϵ_t ϵ_m ϵ_b] [`dielectric-contrasts` Δ_t Δ_b]	50
`inter coulomb mmm1d` *switch_radius* [*bessel_cutoff*] *maximal_pairwise_error*	52
`inter coulomb mmm1d tune` *maximal_pairwise_error*	
`inter coulomb maggs` *f_mass mesh field_friction* [`yukawa` *kappa* r_{cut}]	52
`inter coulomb elc` *maximal_pairwise_error gap_size* [*far_cutoff*]	52
`inter magnetic mdlc` *accuracy gap_size* [*far_cutoff*]	53
`inter magnetic` $l_B m$ `mdds n_cut` *value_n_cut* [*far_cutoff*]	53
`inter magnetic` $l_B m$ `dawaanr` [*far_cutoff*]	54
`inter` *typeid1 typeid1* `comfixed` *flag*	54
`inter` *typeid1 typeid2* `comforce` *flag dir force fratio*	55
`setmd` *variable*	55
`setmd` *variable* [*value*]`+`	

`analyze momentofinertiamatrix` *typeid* `analyze find_principal_axis` *typeid*	74
`analyze aggregation` *dist_criteria s_mol_id f_mol_id* [*min_contact* [*charge_criteria*]]	74
`analyze necklace` *pearl_treshold back_dist space_dist first length*	75
`analyze holes` $typeid_{\mathrm{probe}}$ *mesh_size*	75

`analyze energy`
`analyze energy ( total | kinetic | coulomb | magnetic )`
`analyze energy bonded` *bondid*
`analyze energy nonbonded` *typeid1 typeid2*
	75

`analyze pressure`
`analyze pressure total`
`analyze pressure ( totals | ideal | coulomb |`
`tot_nonbonded_inter | tot_nonbonded_intra )`
`analyze pressure bonded` *bondid*
`analyze pressure nonbonded` *typeid1 typeid2*
`analyze pressure nonbonded_intra` [*typeid*]
`analyze pressure nonbonded_inter` [*typeid*]
	76

`analyze stress_tensor`
`analyze stress_tensor total`
`analyze stress_tensor ( totals | ideal | coulomb |`
`tot_nonbonded_inter | tot_nonbonded_intra )`
`analyze stress_tensor bonded` $bond_type$
`analyze stress_tensor nonbonded` *typeid1 typeid2*
`analyze stress_tensor nonbonded_intra` [*typeid*]
`analyze stress_tensor nonbonded_inter` [*typeid*]
	76

`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*	77
`analyze set chains` [*chain_start n_chains chain_length*] `analyze set chains`	78
`analyze ( re | <re> )` [*chain_start n_chains chain_length*]	79
`analyze ( rg | <rg> )` [*chain_start n_chains chain_length*]	80
`analyze ( rh | <rh> )` [*chain_start n_chains chain_length*]	80
`analyze ( internal_dist | <internal_dist> )` [*chain_start n_chains chain_length*]	80
`analyze ( bond_dist | <bond_dist> )` [`index` *index*] [*chain_start n_chains chain_length*]	81
`analyze ( bond_l | <bond_l> )` [*chain_start n_chains chain_length*]	81
`analyze ( formfactor | <formfactor> )` *qmin qmax qbins* [*chain_start n_chains chain_length*]	81
`analyze rdfchain` *rmin rmax rbins* [$chain_start$ n_chains $chain_length$]	82

`analyze ( <g1>| <g2>| <g3> ) [`*chain$_s$tart n$_c$hains chain$_l$ength*`]` `analyze g123 [-init] [`*chain$_s$tart n$_c$hains chain$_l$ength*`]`	82

`analyze append`
`analyze remove [`*index*`]`
`analyze replace` *index*
`analyze push [`*size*`]`
`analyze configs` *config*
	83

`analyze configs` `analyze stored`	83
`plotObs` *file* { *x1*`:`*y1 x2*`:`*y2 ...*} [`titles {` *title1 title2 ...*}] [`labels {` *xlabel* [*ylabel*] }] [`scale` *gnuplot − scale*] [`cmd` *gnuplot − command*] [`out` *filebase*]	84
`plotJoin {` *source1 source2 ...*} *final*	84
`calcObAv` *file index* [*start*] `calcObErr` *file index* [*start*] `calcObsAv` *file* { *i1 i2 ...*} [*start*] `nameObsAv` *file* { *name1 name2 ...*} [*start*] `findObsAv` *val what*	85
`uwerr` *data nrep col* [*s_tau*] [`plot`] `uwerr` *data nrep f* [*s_tau* [*f_args*]] [`plot`]	85
`blockfile` *channel* `write variable {`*varname1 varname2 ...*} `blockfile` *channel* `write variable all`	86

`blockfile` *channel* `write tclvariable {` *varname1 varname2 ...*}
`blockfile` *channel* `write tclvariable all`
`blockfile` *channel* `write tclvariable reallyall`
	86

`blockfile` *channel* `write particles` *what* (*range* | `all`) `blockfile` *channel* `write bonds` *range* `blockfile` *channel* `write interactions`	88

`blockfile` *channel* `write random`
`blockfile` *channel* `write bit_random`
`blockfile` *channel* `write seed`
`blockfile` *channel* `write bitseed`
	88

`blockfile` *channel* `write configs`	89
`blockfile` *channel* `write start` *tag* `blockfile` *channel* `write end` `blockfile` *channel* `write` *tag* [*arg*]...	89

`blockfile` *channel* `read start`
`blockfile` *channel* `read toend`
`blockfile` *channel* `read ( particles | interactions | bonds | variable |`
`seed | random | bitrandom | configs )`
`blockfile` *channel* `read auto`
	90

`checkpoint_set` *destination* [*num$_c$onfigs* [*tclvar* [*iaflag* [*varflag* [*ranflag*]]]]]	90

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 3.2 on page 20). Too activate `FEATURE`, add the following line to the header file:

```
#define FEATURE
```

This list is not complete! To get a full list of available features, look into `myconfig-sample.h`.

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 57) 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 5.4 on page 49 for details on these algorithms.
- `MAGNETOSTATICS` In analogy to ELECTROSTATICS, this switches on the various magnetostatics algorithms, such as P3M. See section 5.4 on page 49 for details on these algorithms. Requires `DIPOLES` and `ROTATION`.
- `DIPOLES` This activates the dipole-moment property of particles; you probably also want to activate MAGNETOSTATICS to get any interactions for the dipoles. Requires `ROTATION`.
- `ROTATION` Switch on rotational degrees of freedom for the particles, as well as the corresponding quaternion integrator. See section **??** on page ?? 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.

Docs for rotation missing

- `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 33 for possible constraint forms.

- `TUNABLE_SLIP` Switch on tunable slip conditions for planar wall boundary conditions. See section 5.2.12 on page 43 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` **Docs missing**

- `COMFIXED` **Docs missing**

- `MOLFORCES` **Docs missing**

- `BOND_CONSTRAINT` Turns on the RATTLE integrator which allows for fixed lengths bonds between particles. **How to use it?**

- `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-equilbrium (shear) MD support (see section **??** on page ??). **Docs missing**

- `NPT` Enables an on–the–fly NPT integration scheme (see section **??** on page ??). **Docs missing**

- `DPD` Enables the dissipative particle dynamics thermostat (see section **??** on page ??).

- `TRANS_DPD` Enables the transversal dissipative particle dynamics thermostat (see section 6.2.2 on page 60).

- `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 61).

- `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 **??** on page ??). **Docs missing**

B.2 Interactions

The following switches turn on various short ranged interactions (see section 5.2 on page 36):

- `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.
- `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 5.3.5 on page 46). This will change in the near future to three independent angle potentials:

- `BOND_ANGLE_HARMONIC`
- `BOND_ANGLE_COSINE`
- `BOND_ANGLE_COSSQUARE`

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^3Ms real space part.
- `ESK_DEBUG` debugging of P^3Ms 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.
- 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 [11]: 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 comparision 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 15000τ.

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 25, 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 Conversion of Deserno files

The following procedures are found in scripts/convertDeserno.tcl.

- `convertDeserno2MD <source_file> <destination_file>`

  converts the particle configuration stored in *source_file* from Deserno-format into blockfile-format, importing everything to ESPResSo and writing it to *destination_file*. The full particle information, bonds, interactions, and parameters will be converted and saved. If *destination_file* is "-1", the data is only loaded into ESPResSo and nothing is written to disk. If *destination_file* has the suffix `.gz`, the output-file will be compressed. The script uses some assumptions, e. g. on the *particle_type_number*s of The part command for polymers, counter-ions, or on sigma, shift, offset for Lennard-Jones-potentials (The inter command; current defaults are 2.0, 0, 0, respectively); these are all set by

  `initConversion`

  (which is automatically called by convertDeserno2MD) so have a look at the source-code of `convertDeserno.tcl` in the `scripts`-directory for a complete list of assumptions. However, if for some reasons different values need to be set, it is possible to bypass the initialization routine and/or override the default values, e. g. by explicitly executing initConversion, afterwards overwriting all variables which need to be re-set, and manually invoking the main conversion script

  `convertDeserno2MDmain <source_file> <destination_file>`

  to complete the process.

- `convertMD2Deserno <source_file> <destination_file>`

  converts the particle configuration stored in the ESPResSo-blockfile *source_file* into a Deserno-compatible *destination_file*. If *source_file* is "-1", the data is entirely taken from ESPResSo without loading anything from disk. If *source_file* has the suffix `.gz`, it is assumed to be compressed; otherwise it will be treated as containing plain text. Since Deserno stores much more than ESPResSo does due to a centralized vs. a local storage policy, it depends on correct values for the following properties, which therefore should be contained in *source_file*:

  1. the *particle_type_number* used for polymers, counter-ions, and salt-molecules (defaults are: `set type_P 0`, `set type_CI 1`, and `set type_S 2`
  2. the *bond_type_number* used for FENE-interactions (default is: `set type_FENE 0`)

As for convertDeserno2MD, the defaults are set upon initialization by

```
initConversion
```

(which is automatically called by convertMD2Deserno as well), but may be overwritten the same way as explained for tcl_convertDeserno2MD. However, parameters stored in *source_file* cannot (and will not) be overwritten, because they were part of the system originally saved and should not be altered initially. Note, that some entries in a Deserno-file cannot be determined at all, these are by default set to

```
set prefix AA0000
set postfix 0
set seed -1
set startTime -1
set endTime -1
set integrationSteps -1
set saveResults -1
set saveConfig -1
set subbox_1D -1
set ip -1
set step -1
```

but of course may be overwritten as well after calling initConversion and before continuing with

```
convertMD2DesernoMain <source_file> <destination_file>
```

the actual conversion process. The Deserno-format assumes knowledge of the topology, hence a respective analysis is conducted to identify the type and structure of the polymer network. The script allows for randomly stored polymer solutions and melts, no matter how they're messed up; however, crosslinked networks need to be aligned to be recognized correctly, i.e. they must be set up consecutively, such that the first chain with $MPC monomers corresponds to the first $MPC particles in [part], the 2nd one to the $MPC following particles, etc. etc.

- It is now possible to save the whole state of ESPResSo, including all parameters and interactions. These scripts make use of that advantage by storing everything they find in the Deserno-file - but vice versa they also expect you to provide a blockfile containing all possible informations.

These conversion scripts have been tested with both polymer melts and end-to-end-crosslinked networks in systems with or without counterions. It should work with additional salt-molecules or neutral networks as well, although that hasn't been tested yet - if you've some of these systems in a Deserno-formated file, please submit them for extensive analysis.

E Maggs algorithm

F Bibliography

[1] Juan J. Cerda, V. Ballenegger, O. Lenz, and C.Holm. P3M algorithm for dipolar interactions. *J. Chem. Phys.*, 129:234104, 2008.

[2] M. Deserno. *Counterion condensation for rigid linear polyelectrolytes*. PhD thesis, Universität Mainz, 2000.

[3] M. Deserno and C. Holm. How to mesh up Ewald sums. i. *J. Chem. Phys.*, 109: 7678, 1998.

[4] M. Deserno and C. Holm. How to mesh up Ewald sums. ii. *J. Chem. Phys.*, 109: 7694, 1998.

[5] M. Deserno, C. Holm, and H. J. Limbach. *Molecular Dynamics on Parallel Computers*, chapter How to mesh up Ewald sums. World Scientific, Singapore, 2000.

[6] P.P. Ewald. Die berechnung optischer und elektrostatischer gitterpotentiale. *Ann. Phys.*, 64:253–287, 1921.

[7] 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.

[8] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. IOP, 1988.

[9] C. Junghans, M. Praprotnik, and K. Kremer. *Transport properties controlled by a thermostat: An extended dissipative particle dynamics thermostat*, volume 4. 2008.

[10] Jiri Kolafa and John W. Perram. Cutoff errors in the ewald summation formulae for point charge systems. *Molecular Simulation*, 9(5):351–368, 1992.

[11] K. Kremer and G. S. Grest. Dynamics of entangled linear polymer melts: A molecular-dynamics simulation. *J. Chem. Phys.*, 92:5057, 1990.

[12] H. J. Limbach and C. Holm. Single-chain properties of polyelectrolytes in poor solvent. *J. Phys. Chem. B*, 107(32):8041–8055, 2003.

[13] 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.

[14] 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`.

[15] 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.

[16] T. Soddemann, B. Dünweg, and K. Kremer. A generic computer model for amphiphilic systems. *Eur. Phys. J. E*, 6:409, 2001.

[17] 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.

[18] Ulli Wolff. Monte carlo errors with less errors. *Comput. Phys. Commun.*, 156: 143–153, 2004.

Index