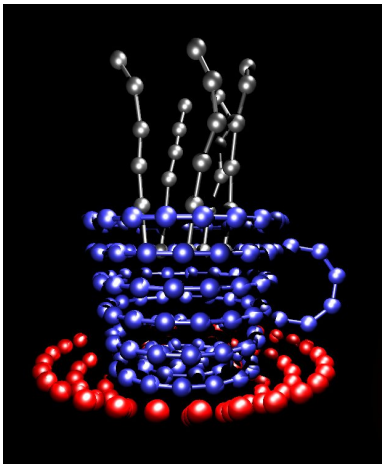

ESPResSo User's Guide

June 25, 2007

Contents

1. Introduction

(new)

- 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

(new)

- Reduced units
- comparison to "real units"
- three examples on different length scales
  - some atomistic model?
  - coarse-grained model (*e.g.* lipid bilayer)
  - billards?

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- and shell-commands can be found. The following conventions are used in these decriptions:

- 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 *italic font*. The description following the syntax definition should contain a detailed explanation of the argument and its type.
- `[`*argument*`]` specifies, that *argument* is optional, *i.e.* it can be omitted.
- `<`*alt1*`|`*alt2*`>` specifies, that one of the alternatives *alt1* or *alt2* can be used.

Example

`writevsf` *file* `[<short|verbose>] [<`*radii*`|auto>] [typedesc` *typedesc*`]`

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

2. First steps

2.1. Quick installation

If you have the requirements (see section 1.5 on the preceding page) 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
```

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 17). 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 18) 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 19.

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 21.

When these steps have successfully completed, ESPResSo can be started with the command (see section 3.5 on page 22)

```
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 [1], 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^{-4}, 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.

Now we can integrate the system:

```
set integ_steps 200
for {set i 0} { $i < 20 } { incr i} {
  set temp [expr [analyze energy kinetic]/(1.5*$n_part)]
  puts "t=[setmd time] E=[analyze energy total], T=$temp"
  integrate $integ_steps
}
```

This code block is the primary simulation loop and runs $20\times$`integ_steps` MD steps. Every `integ_steps` time steps, the potential, electrostatic and kinetic energies are printed out (the latter one as temperature). However, the simulation will crash: ESPResSo complains about particle coordinates being out of range. The reason for this is simple: Due to the initial random setup, the overlap energy is around a

million kT, which we first have to remove from the system. In ESPResSo, this is can be accelerated by capping the forces, i. e. modifying the Lennard–Jones force such that it is constant below a certain distance. Before the integration loop, we therefore insert this equilibration loop:

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

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

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

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

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

The `blockfile read auto` commands will set the Tcl variables `box_l` and `density` to the values specified in the file when encountering the `tclvariable` block, and the particle positions and types of all 216 particles are restored when the `particles` block is read.

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

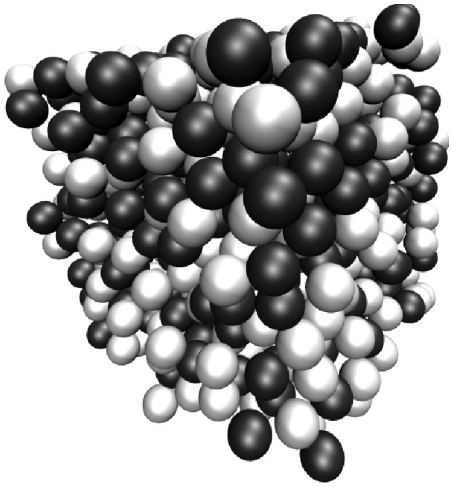

Figure 2.1.: VMD Snapshot of the salt system

can be obtained using the `analyze` command:

```
set rdf [analyze rdf 0 1 0.9 [expr $box_l/2] 100]
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:

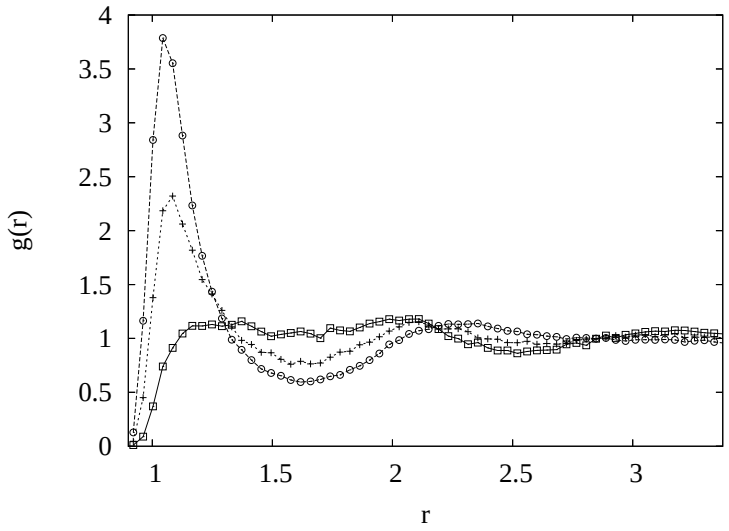

Figure 2.2.: Radial distribution functions $g_{++}(r)$ between equal charges (rectangles) and $g_{+-}(r)$ for opposite charges (circles). The plus symbols denote $g(r)$ for an uncharged system.

```
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]
  foreach value [lindex $rdf 1] {
     lappend rlist   [lindex $value 0]
     lappend rdflist [lindex $value 1] }
  set avg_rdf [vecadd $avg_rdf $rdflist]
  incr cnt
}
set avg_rdf [vecscale [expr 1/$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 that `argv` is a predefined variable: it contains all the command line parameters. Therefore this script should be called like

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

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

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

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

The code example given before is still quite simple, and the reader is encouraged to try to extend the example a little bit, e. g. by using differently sized particle, or changing the interactions. If something does not work, ESPResSo will give comprehensive error messages, which should make it easy to identify mistakes. For real simulations, the simulation scripts can extend over thousands of lines of code and contain automated adaption of parameters or online analysis, up to automatic generation of data plots. Parameters can be changed arbitrarily during the simulation process, as needed for e. g. simulated annealing. The possibility to perform non–standard simulations without the 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 22 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 22).

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. The configuration header `myconfig.h`

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 90 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 preceding 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 exploited, 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 next page), or when `make` is called with the setting `myconfig=`*myconfig_header* (see section 3.4 on page 21).

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 compilation process are not placed next to the source files, but into a separate *build directory* instead. Refer to section 3.1 on page 17 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
```

`--enable-chooser` This option will enable the automatic binary chooser mechanism for ESPResSo (see section 3.4.1 on page 22). 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-config=KNOWN_CONFIG` For some known systems, where `configure` does not find the required libraries and compiler options, predefined settings can be used. The following configuration names are known: `dino` and `blade`. The default for this option is: `none`.

`--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.

`--enable-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. This option is enabled by default.

`--with-myconfig=MYCONFIG_HEADER` This option sets the name of the local configuration header (see 3.2 on page 18). 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.

`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 furing 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 on the following page.
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` *particle_number* [`pos` *x y z*] [`type` *particle_type_number*]
[`q` *charge*][1] [`v` *x_value y_value z_value*] [`f` *x_value y_value z_value*]
[`quat` *q1 q2 q3 q4*] [`omega` *x_value y_value z_value*]
[`torque` *x_value y_value z_value*] [[`un`]`fix` *x y z*]
[`ext_force` *x_value y_value z_value*] [`bond` *bond_type_number*]
[`exclude` *exclusion_partner*...]

Required features

`PART`

[1] `ELECTROSTATICS`

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

- *particle_number*
- [`pos` *x y z*] Sets the position of this particle to (x, y, z).
- [`type` *particle_type_number*] Restrictions: *particle_type_type_number* ≥ 0.
  The *particle_type_number* is used in the inter command to define the parameters of the non bonded interactions between different kinds of particles.
- [`q` *charge*]
- [`v` *x_value y_value z_value*]
- [`f` *x_value y_value z_value*]
- [`quat` *q1 q2 q3 q4*] [`omega` *x_value y_value z_value*] [`torque` *x_value y_value z_value*]
  Require the feature ROTATION.
- [`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 independetly. For

example *fix 0 0 1* will fix motion only in z. Note that *fix* without arguments is equivalent to *fix 1 1 1* (Needs compiled flag EXTERNAL_FORCES in config.h).

- [`ext_force` *x_value y_value z_value*] An additional external force is applied to the particle (Needs compiled flag EXTERNAL_FORCES in config.h).
- [`unfix`] Release any external influence from the particle (Needs compiled flag EXTERNAL_FORCES in config.h).
- [`bond` *bond_type_number partner*+] Restrictions: *bond_type_number* ≥ 0; *partner* must be an existing particle. The *bond_type_number* is used for the inter command to define bonded interactions.
- `bond delete` Will delete all bonds attached to this particle.
- [`exclude` *exclusion_partner*+] Restrictions: *exclusion_partner* must be an existing particle. Between the current particle an the exclusion partner(s), no non-bonded interactions are calculated (Needs compiled flag EXTERNAL_FORCES in config.h). Note that unlike bonds, exclusions are stored with both partners. Therefore this command adds the defined exclusions to both partners.
- [`exclude delete` *exclusion_partner*+] Searches for the given exclusion and deletes it. Again deletes the exclusion with both partners.

4.1.2. Getting particle properties

Syntax

(1) `part` *particle_number* `print`
`( id | pos | type | folded_position | type | q | v | f | fix | ext_force | bond | connections` [*range*] `)+`
(2) `part`

Required features

Description
Variant (1) will return a list of the specified properties of particle *particle_number*, 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` *particle_number* `delete`
(2) `part deleteall`

Required features

Description

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

4.1.4. Exclusions

Syntax

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

Required features

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 neigbor 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

(1) `polymer` *num_polymers monomers_per_chain bond_length*
[`start` *part_id*] [`pos` *x y z*] [`mode` (`RW` | `SAW` | `PSAW`) [*shield* [*max_try*]]]
[`charge` *val_charged_monomer*] [`distance` *dist_charged_monomer*]
[`types` *type_neutral_monomer* [*type_charged_monomer*]] [`bond` *type_bond*]
[`angle` *phi* [*theta* [*x y z*]]]

(2) `polymer`

Required features

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 distance between two adjacent monomers.
- [`start` *part_id*] 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* [*max_try*]]] 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 *max_try* times.

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

- [`charge` *val_charged_monomer*] Sets the valency of the charged monomers. If the valency of the charged polymers *val_charged_monomer* is smaller than 10^{-10}, the charge is assumed to be zero, and the types are set to *type_charged_monomer* = *type_neutral_monomer*. This defaults to 0.0.
- [`distance` *dist_charged_monomer*] 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` *type_neutral_monomer* *type_charged_monomer*] Sets the type numbers of the neutral and charged monomer types to be used with the `part` command. If *type_neutral_monomer* is defined, *type_charged_monomer* defaults to 1. If the option is omitted, both monomer types default to 0.
- [`bond` *type_bond*] Sets the type number of the bonded interaction to be set between the monomers. This defaults to 0.
- [`angle` *phi* [*theta* [*x* *y* *z*]]] Allows for setting up helices or planar polymers: *phi* sets the angle ϕ and *theta* sets the angle θ between adjacent bonds. x, y and z set the position of the second monomer of the first chain.

4.2.2. `counterions`: Set up counterions

Syntax

```
counterions N_CI [shield [max_try ]] [start part_id] [mode ( SAW | RW )]
      [charge val_CI] [type type_CI]
```

Required features

Description

Create *N_CI* counterions in the simulation box.

Docs required.

4.2.3. `salt`: Set up salt ions

Syntax

```
salt N_pS N_nS [start part_id] [mode ( SAW | RW ) [shield [max_try]]]
     [charges val_pS [val_nS]] [types type_pS [type_nS]]
```

Required features

Description
Create *N_pS* positively and *N_nS* negatively charged salt ions of charge *val_pS* and *val_nS* within the simulation box.

4.2.4. `diamond`: Setting up diamond polymer networks

Syntax

```
diamond a bond_length MPC [counterions N_CI] [charges val_nodes val_cM val_CI]
     [distance cM_dist] [nonet]
```

Required features

Docs missing.

Description
Creates a diamond-shaped polymer network with 8 tetra-functional nodes connected by $2 * 8$ polymer chains of length *MPC*.

4.2.5. `icosahedron`: Setting up an icosahedron

Syntax

```
icosahedron a MPC [counterions N_CI] [charges val_cM val_CI]
     [distance cM_dist]
```

Required features

Docs missing.

Description
Creates a modified icosahedron to model a fulleren (or soccer ball).

4.2.6. `crosslink`: Cross-linking polymers

Syntax

```
crosslink N_P MPC [start part_id] [catch r_catch] [distLink link_dist]
     [distChain chain_dist] [FENE type_FENE] [trials max_try]
```

Required features

Docs missing.

Description
Attempts to end-crosslink the current configuration of *N_P* equally long polymers with *MPC* monomers each, returning how many ends are successfully connected.

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 10` *id*

Required features

Description

Does it really only work for LJ-potentials?

A constraint is a surface which interacts with the desired particles via a Lennard-Jones potential.

$$4\epsilon\left(\left(\frac{\sigma}{r}\right)^{12}-\left(\frac{\sigma}{r}\right)^{6}+shift\right)$$

with r being the distance of the center of the particle to the surface. The corresponding force acts in direction of the normal vector of the surface. The constraints are identified like a particle for the lennard-jones force calculation.

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.

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 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 not the whole length but a 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.

4.3.1. Deleting a constraint

Syntax

`constraint delete` *num*

Required features

Description

This will delete the constraint with the number *num*.

4.3.2. Getting the force on a constraint

Syntax

`constraint force` n

Required features

Description
Returns the force acting on the nth constraint.

4.3.3. Getting the currently defined constraints

Syntax

`constraint`

Required features

Description
Prints out all constraint information.

5. `inter`: Setting up interactions

5.1. Non-bonded, short-ranged interactions

5.1.1. Lennard-Jones interaction

Syntax

| `inter` *type1* *type2* `lennard_jones` ϵ σ *cutoff* *shift* *offset*

Required features
`LENNARD_JONES`

Description
This command will define a Lennard-Jones interaction between particles of the types *type1* and *type2*. The potential is defined by

$$V_{\mathrm{LJ}}(r) = \begin{cases} 4\varepsilon((\frac{\sigma}{r-offset})^{12} - (\frac{\sigma}{r-offset})^{6} + shift) & \text{, if } r < cutoff \\ shift & \text{, otherwise} \end{cases} \tag{5.1}$$

5.1.2. Soft-sphere interaction

Syntax

| `inter` *type1* *type2* `soft-sphere` *a* *n* *cut* *offset*

Required features

Description

Docs missing.

5.1.3. Lennard-Jones cosine interaction

Syntax

| (1) `inter` *type1* *type2* `lj-cos` *epsilon* *sigma* *cutoff* *offset*
| (2) `inter` *type1* *type2* `lj-cos2` *epsilon* *sigma* *offset* ω

Required features

Description
The Lennard-Jones+Cosine potential (Soddemann et. al. Eur. Phys. J. E. 6, 409-419 (2001))

Create bibtex reference

Variant (1): for $r < r_{min} = offset * 2^{\frac{1}{6}} * \sigma$:

Equations

$$4\varepsilon((\frac{\sigma}{r-offset})^{12} - (\frac{\sigma}{r-offset})^{6})$$

for $cutoff > r > r_{min} = offset * 2^{\frac{1}{6}} * \sigma$:

$$\frac{1}{2}\varepsilon(cos(\alpha(r - offset)^2 + \beta) - 1)$$

where α and β are given by:

$\alpha = \frac{\pi}{(cutoff-offset)^2-(r_{min}-offset)^2}$

$\beta = \pi * (1 - \frac{(r_{min}-offset)^2}{(cutoff-offset)^2-(r_{min}-offset)^2})$

Variant (2): for $r < r_{change} = offset * 2^{\frac{1}{6}} * \sigma$:

$$4\varepsilon((\frac{\sigma}{r-offset})^{12} - (\frac{\sigma}{r-offset})^6)$$

for $cutoff = offset * 2^{\frac{1}{6}} * \sigma + \omega > r > r_{change}$:

$$\varepsilon * cos^2(\frac{\pi*(r-r_{change})}{2*\omega})$$

5.1.4. Morse interaction

Syntax

| `inter` *type1* *type2* `morse` *epsilon* *alpha* *rmin* *cut*

Required features

Docs

Description

5.1.5. Buckingham interaction

Syntax

| `inter` *type1* *type2* `buckingham` *A* *B* *C* *D* *cut* *discontinuity* *shift*

Required features

Docs

Description

5.1.6. Tabulated interaction

Syntax

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

Required features

`TABULATED`

Description

An arbitrary tabulated non-bonded pair potential.

To use this potential you must provide a file which contains the tabulated forces and energies as a function of the separation distance.

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 parameter you should supply in the file is the number of data points in the table. This should be an integer. Take care when choosing an appropriate value for the number of points remembering that a copy of each lookup table is kept on each node and must be referenced very frequently.

The second parameter you should supply is the minimum tabulated separation distance. The third parameter should be the maximum tabulated separation distance This will act as the effective cutoff value for the potential. Between minval and maxval the force and energy are assumed to be tabulated at fixed intervals such that the size of this interval is given by:

$\frac{maxval-minval}{n-1}$

Where n is the number of data points in the table

The remaining data in the file should consist of n data triples *distance force energy*. Note that distance is only included for human readability of the file. Its values do not matter but it must be present to satisfy the file read format. In the future a more structured file format will be required for the tabulated input file. The values of force and energy should be given as follows:

force: $-\frac{U'(r)}{r}$

energy: $U(r)$

5.1.7. Gay-Berne interaction

Syntax

`|inter` *type1 type2* `gay-berne` *epsilon sigma cutoff k1 k2 mu nu*

Required features

Description

The Gay-Berne potential for prolate and oblate particles. The Gay-Berne potential is an anisotropic version of the classic Lennard-Jones potential, with orientational dependence in the range and well-depth functions σ and ϵ:

Typeset formulas

$$U(\mathbf{r}_{ij},\hat{\mathbf{u}}_i,\hat{\mathbf{u}}_j) = 4\epsilon(\hat{\mathbf{r}}_{ij},\hat{\mathbf{u}}_i,\hat{\mathbf{u}}_j)\left[\left(\frac{\sigma_0}{\mathbf{r}_{ij}-\sigma(\hat{\mathbf{r}}_{ij},\hat{\mathbf{u}}_i,\hat{\mathbf{u}}_j)+\sigma_0}\right)^{12}-\left(\frac{\sigma_0}{\mathbf{r}_{ij}-\sigma(\hat{\mathbf{r}}_{ij},\hat{\mathbf{u}}_i,\hat{\mathbf{u}}_j)+\sigma_0}\right)^{6}\right]$$

where

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

and

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

re unit vectors $\hat{\mathbf{u}}_i$ and $\hat{\mathbf{u}}_j$ give the orientation of the two particles and vector $\mathbf{r}_{ij} = r_{ij}\hat{\mathbf{r}}_{ij}$ is the intermolecular vector.

The parameters $\chi = \frac{k_1^2-1}{k_1^2+1}$ and $\chi' = \frac{k_2^{1/\mu}-1}{k_2^{1/\mu}+1}$ are responsible for the degree of anisotropy of the molecular properties. k_1 is the molecular elongation, and k_2 is the ratio of the potential well depths for the side-by-side and end-to-end configurations. Exponents μ and ν are adjastable parameters of the potential. There are several Gay-Berne pareme-terizations exist; the original one being $k_1 = 3$, $k_1 = 5$, $\mu = 2$ and $\nu = 1$.

5.1.8. Capping the force during warmup

Syntax

| `inter ljforcecap` *maxforce*

Required features

For which potentials does it work?

Description

This command will cap the force to *maxforce*, *i.e.* for particle distances which would lead to larger forces than *maxforce* the Lennard-Jones potential is replaced by $r * maxforce$. Particles placed exactly on top of each other will be subject to a force of magnitude *maxforce* applied in $\pm x$ direction. To return to the uncapped potential you have to set *maxforce* to 0. Note that `ljforcecap` applies to all given Lennard-Jones interactions regardless of the particle types.

5.2. Bonded interactions

Bonded interactions possess an *bonded interaction type id*. On the one hand, this id is used when particles and bonds between particles are specified in the command `part` (see section 4.1 on page 23). On the other hand, the id is used when the interaction is specified.

5.2.1. FENE bond

Syntax

| `inter` *bond_type_number* `fene` *K_fene R_fene*

Required features

Description

$$U^{FENE} = -\frac{1}{2}K_{FENE}R_{FENE}^2 \ln\left(1 - \left(\frac{r}{R_{FENE}}\right)^2\right)$$

5.2.2. Harmonic bond

Syntax

| `inter` *bond_type_number* `harmonic` *K_harmonic R_harmonic*

Required features

Description

$$U^{Harmonic} = \frac{1}{2} K_{harmonic} (r - R_{harmonic})^2$$

5.2.3. Subtracted Lennard-Jones bond

Syntax

| `inter` *bond_type_number* `subt_lj` *K_subt_lj* *R_subt_lj*

Required features

Description
This "bonded" interaction subtracts the Lennard-Jones force/energy of every bonded pair from the total force/energy. The first parameter, *K_subt_lj* is a dummy and is not used. The second parameter, *R_subt_lj* is used as a check. If the any bond length in the system exceeds this value, the program crashes. When not needed, this crashing can be disabled by commenting out a few lines in `subt_lj.h`. When using this "bonded" interaction, it is worthwhile to consider capping the Lennard-Jones potential appropriately so that round-off errors can be avoided.

5.2.4. Bond-angle interactions

Syntax

| `inter` *bond_type_number* `angle` *bend* [*phi0*]

Required features

Description
bend is the bending constant in units of $k_B T$. The optional parameter *phi0* $= \phi_o$ is the equilibirum bond angle in rad ranging from 0 to π. If this paramter is not given the default value is $\phi_o = \pi$ which corresponds to a stretched configuration.

- Harmonic bond angle potential: (flag: BOND_ANGLE_HARMONIC) This potential is also used for example in YASP and good old polyMD.

$$U^{bend}_{harmonic} = \frac{bend}{2} (\phi - \phi_0)^2$$

- Cosine bond angle potential: (flag: BOND_ANGLE_COSINE) The ESPResSo original!

$$U^{bend}_{cosine} = \frac{bend}{2} (1 + \cos(\phi - \phi 0))$$

- Cosine square bond angle potential: (flag: BOND_ANGLE_COSSQUARE) A form which is used for example in the GROMOS96 force field.

$$U^{bend}_{cossquare} = \frac{bend}{2}(\cos(\phi) - \cos(\phi_0))^2$$

5.2.5. Dihedral interactions

Syntax

| `inter` *bond_type_number* `dihedral` *mult bend phase*

Required features

Description

$$U^{dihedral} = bend\,(1 + phase\ cos(mult\,\phi))$$

Here ϕ is the dihedral angle defined by the particle quadrupel p1, p2, p3 and p4. *mult* is the multiplicity of the potential (number of minimas) and can take integer values from 1 to 6. *phase* is a phase parameter which takes the values ± 1 and *bend* is the bending constant of the potential. Together with appropriate Lennard-Jones interaction this potential can mimic a large number of atomic torsion potentials. The dihedral angle is the angle between the planes defined by the particle triples p1, p2 and p3 and p2, p3 and p4 as illustrated in the figure to the right. Dihedral bonds have to be stored at particle p2!

5.2.6. Tabulated bond interactions

Syntax

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

Required features

Description

Tabulated bonded potentials can be any potential for bond length potentials, bond angle potentials and dihedral angle potentials. The tabulated forces and energies have to be provided in a seperate file *filename*. The format of this file is identical to the one used for the non-bonded tabulated potentials (see the section about them above). The parameter *type* defines the type of the potential:

- *type* = bond (two body interaction)

  Tabulated bond length potential. The force acts in the direction of the connecting vector between the particles. The cutoff is given by the maximal tabulated

distance. For distances smaller than the tabulated range it uses a linear extrapolation based on the first two tabulated force values. The C implementations are calc_tab_bond_force and tab_bond_energy in tab.h.

- *type* = angle (three body interaction)

  Tabulated bond angle potential (see also the normal implemented bond angle potentials). The force on p_left and p_right acts perpendicular to the connecting vector between the particle and p_mid and in the plane defined by the three particles. The force on the middle particle balances the other two forces. The forces are scaled with the invers length of the connecting vectors. It is assumed that the potential is tabulated for all angles between 0 and π. The C implementations are calc_tab_angle_force and tab_angle_energy in tab.h.

- *type* = dihedral (three body interaction)

  Tabulated torsional dihedral angle potenetial (see also the normal implemented dihedral potentials). 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. The C implementations are calc_tab_dihedral_force and tab_dihedral_energy in tab.h.

5.3. Coulomb interaction

Syntax

(1) `inter coulomb 0.0`
(2) `inter coulomb`

Required features

Description

Variant (1) completely disables Coulomb interactions hence deactivating the electrostatic subsystem. Variant (2) returns the current parameters of the coulomb interaction as a tcl-list, *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}
```

which has the correct format to be used as input to `inter` as well.

5.3.1. P3M

Syntax

`inter coulomb p3m` *r_cut mesh cao alpha*

Required features

Description

Reference

Activates the P3M method to handle the Coulomb interaction

$$U^{C-P3M} = \ell_B T \frac{q_1 q_2}{r} \tag{5.2}$$

Make sure that you know the relevance of the P3M parameters before using P3M!

Tuning P3M

Syntax

```
inter coulomb p3m ( tune | tunev2 ) accuracy accuracy
      [r_cut r_cut] [mesh mesh] [cao cao] [alpha alpha]
```

Required features

Insert docs from `p3m.h`

Description

Make sure you know how to tune p3m parameters before using the automatic tuning feature. Details are described in the documentation of P3M_tune_parameters rsp P3M_adaptive_tune_parameters.

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.

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` *bjerrum* `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_name value*`+`

5.3.2. Debye-Hückel potential

Syntax

```
inter coulomb dh kappa r_cut
```

Required features

Description

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

For

$$\kappa = 0$$

this corresponds to the plain coulomb potential.

5.3.3. MMM2D

Syntax

`inter coulomb mmm2d` *maximal_pairwise_error* [*fixed_far_cutoff*]

Required features

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 ??.

5.3.4. MMM1D

Syntax

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

Required features

Description

MMM1D coulomb method for systems with periodicity 0 0 1. Needs the nsquared cell system (see section 6.4 on page 45). 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 ??.

5.3.5. Maggs' method

Syntax

| `inter coulomb maggs` *f_mass mesh field_friction* [`yukawa` *kappa r_cut*]

Required features

Description

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

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

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

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 value of the friction coefficient for the transversal field degrees of freedom (reserved for future developments)

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*.

5.3.6. ELC

Syntax

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

Required features

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. Other interaction types

5.4.1. Fixing the center of mass

Syntax

| `inter` *particle_type_number1* *particle_type_number1* `comfixed` *comfixed_flag*

Required features

Description

This interaction type applies a constraint on particles of type *particle_type_number1* 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 *particle_type_number1* 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 *particle_type_number1*, 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. The *comfixed_flag* can be set to 1 (which turns on the interaction) or 0 (to turn off the interaction).

5.4.2. Pulling particles apart

Syntax

| `inter` *particle_type_number1* *particle_type_number2*
| `comforce` *comforce_flag* *comforce_dir* *comforce_force* *comforce_fratio*

Required features

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 1 (t1) lets mark one molecule as of type 2 (t2). 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 (*comforce_dir* = 0) or perpendicular to the major axis of the bundle (*comforce_dir* = 1). *comforce_force* is used to set the magnitude of the force. *comforce_fratio* is used to set the ratio of the force applied on particles of t1 vs t2. 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 *comforce_force* is applied on t2 particles, and a force of magnitude (*comforce_force* * *comforce_fratio*) is applied on t1 particles.

5.5. Getting the currently defined interactions

Syntax

```
| inter
```

Required features

Description

Returns a list of all bonded and non-bonded interactions as a Tcl-list, in the same formats as above, *e.g.*

```
{0 0 lennard-jones 1.0 2.0 1.1225 0.0 0.0} {0 FENE 7.0 2.0} {1 angle
 1.0}
```

6. Setting up the system

6.1. `setmd`: Setting global variables.

Syntax

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

Required features

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 following global variables can be set:

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

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

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

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

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

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

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

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

`max_cut` (double, *read-only*) Maximal cutoff of real space interactions.

`max_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_seen_particle` (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. Reason-

Explain '+' in intro.

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!

???

???

able 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 the facing page).

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

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

`n_particle_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).

Docs missing.

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

Docs missing.

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

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

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

`piston` (double, *read-only*) Mass off the box when using NPT_isotropic integrator.

`periodicity` (bool[3]) Specifies periodicity for the three directions. If the feature PARTIAL_PERIODIC is set, this variable can be set to (1,1,1) or (0,0,0) at the moment. If not it is readonly and gives the default setting (1,1,1).

Correct?

`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 timing samples to take into account if set.

`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

Syntax

(1) `thermostat off`
(2) `theormstat` *method* [*parameter*]+

Required features

Description
Change thermostat settings.

Include docs from `thermostat.h`!

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

Syntax

(1) `nemd` *method parameter*
(2) `nemd off`
(3) `nemd profile`
(4) `nemd viscosity`

Required features

Description
Use NEMD (Non Equilibrium Molecular Dynamics) to simulate a system under shear.

Include docs from `nemd.h`!

Put `nemd profile|viscosity` into `analyze`?

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 11.1 on page 86.

6.4.1. Domain decomposition

Syntax

`cellsystem domain_decomposition` [`-no_verlet_list`]

Required features

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
```

Required features

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
```

Required features

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*]+

Required features

Docs missing!

Description

Which integrators do exist?

7.2. `change_volume`: Changing the box volume

Syntax

(1) `change_volume` *V_new*
(2) `change_volume` *L_new* (`x` | `y` | `z` | `xyz`)

Required features

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`

Required features

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` *part_id*] [`count` *N_T*]

Required features

Description

Sets the velocities of the particles with particle ID (see The part command) between *part_id* and *part_id*+*N_T* (defaults to '0' & '[setmd npart]-*part_id*') to a random vector with length in [-vmax,vmax], and returns the absolute value of the total velocity assigned.

7.5. `invalidate_system`

Syntax

| `invalidate_system`

Required features

Description

Documentation not up to date!

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. The C implementation is invalidate_system.

???

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

8. Analysis

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` *part_id*
(3) `analyze distto` *posx* *posy* *posz*

Required features

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 *part_id* (Variant (2)) or to the coordinates (*posx*, *posy*, *posz*) (Variant (3)).

8.1.2. Particles in the neighbourhood

Syntax

(1) `analyze nbhood` *part_id* *r_catch*
(2) `analyze nbhood` *posx* *posy* *posz* *r_catch*

Required features

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 *part_id* in variant (1) or around the spatial coordinate (*posx*, *posy*, *posz*) in variant (2).

8.1.3. Particle distribution

Syntax

`analyze distribution` *part_type_list_a* *part_type_list_b*
[*r_min* [*r_max* [*r_bins* [*log_flag* [*int_flag*]]]]]

Required features

Description
Returns its parameters and the distance distribution of particles with types specified in *part_type_list_a* (group a) around particles with types specified in *part_type_list_b* (group b) with distances between *r_min* and *r_max*, binned into *r_bins* bins. The bins are either equidistant (*log_flag*=0) or logarithmically equidistant (*log_flag* $\geq$ 1). If an integrated distribution is required, use *int_flag*=1. The distance is defined as the minimal distance between a particle of group a to any of group b.

Output format
The output corresponds to the blockfile format (see section 9.1 on page 66):

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

8.1.4. Radial distribution function

Syntax

`|analyze ( rdf | <rdf> )` *part_type_list_a part_type_list_b* [*r_min r_max r_bins*]

Required features

Description
Returns its parameters and the radial distribution function (rdf) of particles with types specified in *part_type_list_a* (group a) around particles with types specified in *part_type_list_b* (group b). The range is given by *r_min* and *r_max* and is divided into *r_bins* equidistant bins.

Output format
The output corresponds to the blockfile format (see section 9.1 on page 66):

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

8.1.5. Structure factor

Syntax

`|analyze structurefactor` *type order*

Required features

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 66):

{ *q˙value S(q)˙value* }
⋮

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

Syntax

| `analyze vanhove` *type rmin rmax rbins*

Required features

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 61) 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 66):

{ `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*

Required features

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

8.1.8. Moment of intertia matrix

Syntax

| (1) `analyze momentofinertiamatrix` *part_type*
| (2) `analyze find_principal_axis` *part_type*

Required features

Description
Variant (1) returns the moment of inertia matrix for particles of given type *part_type*. 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*]]

Required features

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*

Required features

Description
Algorithm for identifying pearl necklace structures for polyelectrolytes in poor solvent [3]. 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` *prob_part_type_number mesh_size*

Required features

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 [4]. 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 )`
(3) `analyze energy bonded` *bond_type*
(4) `analyze energy nonbonded` *part_type1* *part_type2*

Required features

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 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` *bond_type*
(5) `analyze pressure nonbonded` *part_type1* *part_type2*
(6) `analyze pressure nonbonded_intra` [*part_type*]
(7) `analyze pressure nonbonded_inter` [*part_type*]

Required features

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; in

Document arguments nb_inter, nb_intra, tot_nb_inter and tot_nb_intra

that case only a two-value tcl-list is returned, consisting of the requested contribution and its square.

Note that the ideal pressure is derived using the temperature, hence for systems with $temp = 0$ it will be zero; in those cases it is preferable to use `analyze p_IK1` (see 8.1.14) which looks at the velocities of the particles.

The command is implemented in parallel.

Output format (variant (1)*)*

{ *total_pressure pressure_square*
  { `ideal` *ideal_gas_pressure* }
  { { *bond_type pressure pressure_square* }
    ⋮
  }
  { { *nonbonded_type pressure pressure_square* }
    ⋮
  }
  { `coulomb` *pressure pressure_square*}
}

specifying the pressure, its square, the ideal gas pressure, the contributions from bonded interactions, the contributions from non-bonded interactions and the electrostatic contributions.

8.1.14. Pressure tensor

Syntax

`analyze p_IK1` *bin_volume ind_list flag_all*

Required features

Description

Computes the pressure-tensor and all its contributions in the system. Note that the tensorial entries $p^{(k,l)}$ are directly derived from

$$p^{(k,l)} = \sum_{j>i \in bin} F_{ij}^{(k)} \cdot r_{ij}^{(l)} \tag{8.1}$$

where *e.g.* $F_{ij}^{(k)}$ denotes the kth entry of the difference vector between the force vectors of particles i and j, while the ideal pressure is determined from the particles' velocities, *i.e.*

$$p_{ideal}^{(k,l)} = \sum_{i \in bin} v_i^{(k)} \cdot v_i^{(l)} \tag{8.2}$$

contrary to `analyze pressure ideal` which uses the temperature. In both cases, the sum is taken over only those particles whose identities have been provided in *ind_list* if

flag_all=0, otherwise i loops over *ind_list* while j represents all particles in the system. Since this command is intended to follow the Kirkwood-scheme IK1, all particles given in *ind_list* should represent a bin geometry with volume *bin_volume* for the algorithm to make sense, although this is not checked; hence it is preferable to obtain *ind_list* from `analyze bins`.

This command is executed on the master node only, hence a higher number of nodes only slows it down due to the increasing amount of communication processes required between the nodes to gather all needed particle information. Furthermore, all interactions including electrostatic ones are treated with the virial ansatz given above, which makes non-periodic systems highly recommendable, and which slows electrostatic systems down due to the long-range interactions.

Output format

{ *total_pressure* *pressure_square*
  { `total` *pxx* *pxy* *pxz* *pyx* ... }
  { `ideal` *pxx* *pxy* *pxz* *pyx* ... }
  { `bonded` *pxx* *pxy* *pxz* *pyx* ... }
    { *bond_type* *pxx* *pxy* *pxz* *pyx* ... }
    ⋮
  }
  { `nonbonded` *pxx* *pxy* *pxz* *pyx* ... }
}

specifying the pressure, its square, its tensorial form, and the tensorial form of its components ideal, bonded, nonbonded, coulombic pressure (if applicable); the bonded component is further split up into its different contributions (FENE, ANGLE, HARMONIC) sorted according to the interaction type number specified by the `inter` command (see section 5 on page 31).

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`

Required features

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*]

Required features

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 61).

Output format
{ *re error_of_re re2 error_of_re2* }

Radius of gyration

Syntax

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

Required features

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 61).

Output format
{ *rg error_of_rg rg2 error_of_rg2* }

Hydrodynamic radius

Syntax

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

Required features

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 61).

Output format
{ *rh error_of_rh* }

Internal distances

Syntax

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

Required features

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 61).

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]
```

Required features

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 61).

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]
```

Required features

Description
Returns the average bond-length within a chain, averaged over all chains in the system; hence, it gives the same result as *idf(0)* from `analyze internal_dist`. 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 the facing page).

Form factor

Syntax

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

Required features

Check this!

Description

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

$$S(q) = \frac{1}{chain\_length} \sum_{i,j=1}^{chain\_length} \frac{\sin(qr_{ij})}{qr_{ij}} \tag{8.3}$$

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

Output format

{
  { *q* *S(q)* }
  ⋮
}

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

Chain radial distribution function

Syntax

```
analyze rdfchain r_min r_max r_bins [chain_start n_chains chain_length]
```

Required features

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 *r_min* and *r_max* and it is divided into *r_bins* 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*]

Required features

Description

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

Variant (1) returns

- the mean-square displacement of the beads in the chain (`<g1>`)
- the mean-square displacement of the beads in the center of mass of the chain (`<g2>`)
- or the motion of the center of mass (`<g3>`)

averaged over all stored configurations (see section 8.3).

Variant (2) returns all of these observables for the current configuration, as compared to the reference configuration. The reference configuration is set, when the option `-init` is used.

Output format (variant (1)*)*
{ *gi(0*dt) gi(1*dt)* ... }

Output format (variant (2)*)*
{ *g1(t) g2(t) g3(t)* }

8.3. Storing configurations

Some observables (*i.e.* non-static ones) require knowledge of the particles' positions at more than one or two times. Therefore, it is possible to store configurations for later analysis. Using this mechanism, the program is also able to work quasi-offline by successively reading in previously saved configurations and storing them to perform any analysis desired afterwards.

Note that the time at which configurations were taken is not stored. The most observables that work with the set of stored configurations do expect that the configurations are taken at equidistant timesteps.

Note also, that the stored configurations can be written to a file and read from it via the `blockfile` command (see section 9.1 on page 66).

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*

Required features

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`

Required features

Description

Variant (1) returns all stored configurations, while variant (2) returns only the number of stored configurations.

Output format (variant (1)*)*

{
  {*x1 y1 z1 x2 y2 z2 ...* }
  ⋮
}

8.4. 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*]

Required features

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*

Required features

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*

Required features

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* [*a_tau*] [`plot`]
(2) `uwerr` *data nrep f* [*a_tau* [*f_args*]] [`plot`]

Required features

Description

Calculates the mean value, the error and the error of the error for an arbitrary numerical time series accordings to Wolff [5].

Arguments

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

How exactly does the Tcl-list look like?

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

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

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

- f is a user defined Tcl function returning a double with first argument a vector which has as many entries as data has columns. If f is given instead of the column, the corresponding derived quantity is analyzed.
- *f_args* are further arguments to f.
- *s_tau* is the estimate $S = \tau/tau_{\text{int}}$ as explained in section (3.3) of [1]. 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`

Required features

Description

Variant (1) writes the global variables *varname1 varname2* ... (which are known to the `setmd` command (see section 6.1 on page 43) 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`

Required features

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`

Required features

Description
How is a Tcl-range specified?

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_id* `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`

Required features

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`

Required features

Description

This command writes all configurations currently stored for off-line analysis (see section 8.3 on page 61) to *channel*.

9.1.6. Writing arbitrary blocks

Syntax

| (1) `blockfile` *channel* `write start` *tag*
| (2) `blockfile` *channel* `write end`

Required features

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.

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

Required features

Description

reads the start part of a block and returns the block title.
reads the blocks data and returns it.
reads one block, checks wether it contains data of the given type and reads it.
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 <tag> <rest of block>`

3. if the file is at end, it returns `eof`

If `blockfile <channel> read auto` finds a block, it tries to load the corresponding procedure as described above. `blockfile <channel> read <block>` checks for a block with tag *block* and then again executes the corresponding blockfile_read_auto_*tag*, if it exists.

If that fails, blockfile executes `blockfile_arg1_arg2`, if it exists, with the all arguments given to blockfile. For example

```
blockfile channel write particles "id pos" all
```

results in the evaluation of

```
blockfile_write_particles channel write particles "id pos" all
```

If the next block in a blockfile is a particle block, e. g.

```
{particles {id pos type q}
           {0 27.251 62.31 58.707 1 1.0}
           {1 27.226 61.483 58.146 0 0.0}
}

blockfile <channel> read auto
```

will call

```
blockfile_read_auto_particles <channel> read auto
```

, which then will delete all particles and insert the two particles above.

In the contrary that means that for a new blocktype you will normally implement two procedures:

`blockfile_write_<tag> {channel write <tag> param...}`

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 informations to and from disk from and to ESPResSo.

- ```
  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 variables accessible by The setmd command, all interactions known to The inter command, the full current state of the random number generator (The t_random command or The bit_random command), all informations (i.e. id, position, type-number, charge, velocity, forces, bonds) on all particles, and all particle configurations appended (using e. g. analyze append) for offline-analysis purposes to the file 'checkpoint.block.gz' which is even being compressed on-the-fly (if you don't want that, use `set out [open "checkpoint.block" "w"]` instead). Note that interactions must be stored before particles before bonding informations, 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 informations in the blockfile (again, if you don't have a compressed file, use

  ```
  set out [open "checkpoint.block" "r"]
  ```

  instead) overwriting the current settings in ESPResSo.

9.2. Checkpointing

The following procedures may be used to save/restore checkpoints to minimize the hassel involved when your simulations crashes after long runs. The scripts are located in scripts/auxiliary.tcl and use The blockfile command as file format.

- `checkpoint_set <destination> [<# of configs> [<tclvar> [<ia_flag> [<var_flag> [<ran_flag>]]]]]`

  creates a checkpoint with path/filename *destination* (compressed if *destination* ends with '.gz'), saving the last *# of configs* 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 *ia_flag* / *var_flag* / *ran_flag* 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 *var_flag*. 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.

- `checkpoint_read <origin>`

  restores all the checkpoints whose filenames are listed in *origin* in the order given therein, consequently putting the simulation into the state it was in when tc_-

checkpoint_set was called. If parts of the configs array are given in the files listed in *origin*, it is assumed that they represent a fraction of the whole array.

- `polyBlockWrite <path> <param_list> <part_list>`

  writes out the current 'ESPResSo' configuration as an AxA-blockfile, including parameters, interactions, particles, and bonds. *path* should contain the filename including the full path to it. *param_list* gives a tcl-list of the 'ESPResSo'-parameters (out of) to be saved; if an empty list '' is supplied, no parameters are written. If 'all', all parameters available through The setmd command are written. Defaults to the full parameter set. *part_list* gives a string of the particle-properties (out of pos — type — q — v — f) to be saved to disk; if an empty string ' "" 'is provided, no particles, no bonds, and no interactions are written. Defaults (if omitted) to all particle-properties. Depending on the file-name's suffix, the output will be compressed (if *path* ends with '.gz'), too. Note, that 'polyBlockWrite' in combination with tcl_convertMD2Deserno replaces the (undocumented) function 'polywr': To save the current configuration to a Deserno-compatible file (e. g. for use with 'poly2pdb') you may now use tcl_polyBlockWrite to save your current configuration to a blockfile, and convert that with tcl_convertMD2Deserno afterwards, or you directly write a Deserno-compatible file by invoking

  ```
  convertMD2Deserno "-1" <output-filename>
  ```

  out of ESPResSo to save your current active configuration. However, this last paragraph now has only historical meaning (see Writing pdb/psf files).

- `polyBlockWriteAll <destination> [<tcl-var> [<rdm> [<configs>]]]`

  does even more than tcl_polyBlockWrite, i.e. it saves all current interactions, particles, bonds, ESPResSo-variables to *destination*, but in addition it also saves the tcl-variables specified by *tcl-var* (if 'all', then all the variables in the active script are stored), it saves the state of the random number generator if *rdm* is 'random' (= complete state) or 'seed' (= only the seeds), and it saves all the particle configurations used for analysis purposes if *configs* is all but '-'. Using '-' as value usually skips that entry. 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 atomic 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.

`writepsf <file> { <N_P> <MPC> <N_CI> <N_pS> <N_nS> }|-molecule`

writes the current topology to the file ¡file¿ (here ¡file¿ is not a channel since additional information cannot be written anyways). *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.

`writepdb <file>`

writes the corresponding particle data.

`writepdbfoldchains <file> { < chain_start> <n_chains> <chain_length> <box_l> }`

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 writepdbfoldtopo instead.

`writepdbfoldtopo <file> { <shift> }`

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`

`writevsf` *channelId*
[¡short—verbose¿] [radius ¡*radii*—auto¿] [typedesc *typedesc*]

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.

[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.

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"}`

9.4.2. `writevcf`

`writevcf` *channelId*
[¡short—verbose¿] [¡folded—absolute¿] [pids ¡*pids*—all¿]

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`

`vtfpid` *pid*

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. `imd`: Online-visualisation with VMD

IMD (Interactive Molecular Dynamics) is the protocol 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 up and using of IMD is described.

9.5.1. IMD in the script

In your simulation, the IMD connection is setup up using `imd connect <port>`

where *port* is an arbitrary port number (it has to be between 1024 and 65000). Normally 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 (see lj_liquid.tcl).

Now while the simulation is running, you should execute

```
imd positions <flag>
```

from time to time, which will transfer the current coordinates to VMD, if it is connected. If not, nothing happens and imd connect just consumes a small amount of CPU time. The optional flag argument can take values -unfolded or -fold_chains. By specifying -unfolded the unfolded coordinates for each particle will be given to VMD. Specifying -fold_chains causes imd to call the routine analyze_fold_molecules which folds chains according to their centers of mass and retains bonding connectivity. Note that this routine requires the chain structure to be specified first using the analyze command.

```
imd listen <seconds>
```

can be used to let the simulation wait for *seconds* seconds or until IMD has connected. This is normally only useful in demo scripts, if you want to see all frames of the simulation.

If your simulations terminates,

```
imd disconnect
```

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.

Additionally, you have to provide VMD with the structural information for your system. Therefore your program has to write out psf-/pdb-files using tcl_writepsf and tcl_writepdb.

That hassle is greatly reduced by using the built-in auxiliary script

```
prepare_vmd_connection [<filename> [<wait> [<start>]]]
```

which writes out the necessary 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. If *start* is 1 (the default), it does all that by itself; otherwise it writes those steps out to a script-file 'vmd_start.script and waits for *wait* seconds (default: 0) for you to connect.

9.5.2. Using IMD in VMD

So after your simulation runs and has written the psf/pdb files, you start VMD. Then click on "Molecule", choose fileformat "psf and pdb", and select your psf/pdb files in the corresponding entries. Now click on "Load Molecule". You should see the snapshot you saved in the psf/pdb files.

Then execute "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 the molecule before .

For more information on how to use VMD to extract more information or hide parts of configuration, see the VMD Quick Help.

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.

For error conditions such as the examples given above, an 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

10.1. Finding particles and bonds

10.1.1. `countBonds`

`countBonds` *particle_list*

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 **??** on page ??).

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`

`findPropPos` *particle_property_*
property

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`.

```
[lindex [part $i] [findPropPos [part $i] type]]
```

for example returns the particle type id of particle $i without fixing where exactly that information has to be in the output of `[part $i]`.

`findBondPos` *particle_proper*

Returns the index of the bonds within varparticle_property_list, which is expected to have the same format as [part *particle_number*]; hence its output is the same as [findPropPos ¡particle_property_list¿ bonds]. If the particle does not have any bonds, -1 is returned.

10.1.3. `timeStamp`

`timeStamp` *path prefix postfix suffix*

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
  - `EPSILONO`

    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)

- 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. Under the hood

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

11.1. 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.

12. 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

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 18). Too activate `FEATURE`, add the following line to the header file:

```
#define FEATURE
```

B.1. General features

- `PARTIAL_PERIODIC` By default, all coordinates in ESPResSo are periodic. With `PARTIAL_PERIODIC` turned on, the ESPResSo global variable `periodic` (see section 6.1 on page 43) 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 5.3 on page 37 for details on these algorithms.

Docs for rotation missing

- `ROTATION` Switch on rotational degrees of freedom for the particles, as well as the corresponding quaternion integrator. See section **??** on page ?? for details.
- `DIPOLES` This activates the dipole support in P^3M. Currently, a mixing of dipoles and charges is not possible, *i.e.* all particles have to have charge $q = 0$. Requires `ELECTROSTATICS` and `ROTATION`.
- `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 29 for possible constraint forms.
- `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.

Docs missing

- `COMFORCE`

sing

- COMFIXED
- MOLFORCES — Docs missing
- BOND_CONSTRAINT Turns on the RATTLE integrator which allows for fixed lengths bonds between particles. — How to use it?

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

- 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 ??). — Docs missing
- 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 5.1 on page 31):

- TABULATED Enable support for user–defined interactions.
- LENNARD_JONES Enable the Lennard–Jones potential.
- LJ_WARN_WHEN_CLOSE This adds an additional check to the Lennard–Jones potential that prints a warning of particles come too close so that the simulation becomes unphysical.
- MORSE Enable the Morse potential.
- LJCOS Enable the Lennard–Jones potential with a cosine–tail.
- LJCOS2
- BUCKINGHAM Enable the Buckingham potential.
- SOFT_SPHERE Enable the soft sphere potential.

If you want to use bondangle potentials, you currently need to choose the type by the feature (see section **??** on page ??). 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.
- 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 [2]: 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 23, 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] Gary S. Grest and Kurt Kremer. Molecular dynamics simulation for polymers in the presence of a heat bath. *Phys. Rev. A*, 33(5):3628–31, 1986.

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

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

[4] 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`.

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

Index