LAMMPS

A ready-to-use introductory guide

F. Camerin, G. Del Monte, J. Ruiz-Franco

Department of Physics
University of Rome, La Sapienza

October 2019

Table of contents

How to use this guide

This ready-to-use guide is meant as a quick reference to start up a computer simulation with the LAMMPS simulation package.

We first provide in Chapter 1 technical information on how to install and compile LAMMPS on a working node with all the packages that are required.

We then explain how to prepare the init and input files with which a simulation can be run. The init file contains the initial settings of the system while the input script encloses the simulation details. These are described in depth in Chapter 2 and Chapter 3, respectively. There, we evidence commonly used commands, eventual warnings and those commands that are related to each other. Those listed in Chapter 3 are the basic ones which can be used in any kind simulation, regardless of the system studied. Special cases, that require additional commands, are described in Chapter 5.

Finally, Chapter 4 explains how to launch a simulation in parallel.

The complete online manual can be found here.

Being a work-in-progress project, errors may be found throughout the text. We apologize for this.

Chapter 1

Install LAMMPS

Within this section we will indicate briefly how to build and compile a version of LAMMPS on our working node. We will describe all the steps to build LAMMPS with a series of packages allowing to study statical and dynamical properties of atomic and molecular systems. We will also provide the instructions to build some accelerated packages to exploit the parallel computation on multi-core systems and GPUs. Also the USER-VTK package will be compiled, which allows LAMMPS to dump information about configurations in a format readable by the molecular visualization software VisIt. Here are the steps to be followed:

1. Download and unzip the LAMMPS package downloaded as tarball with the command

   ```
   tar -xzvf lammps*.tar.gz
   ```

   Then enter in the created folder (lammps-<release_date>) and give the shell command:

   ```
   cp src/MAKE/MACHINES/Makefile.ubuntu src/MAKE/
   ```

   Alternatively copy an already unzipped folder (usually its name is again lammps-<release_date>) containing the source code to the home directory. In that case some other preliminary steps are necessary:

   (a) move to directory `src/` and give the shell command

   ```
   make clean-all
   ```

   (b) move to directory `lib/gpu/` and give the command

   ```
   make -f Makefile.linux clean
   ```

2. Within directory `src/` open the file `pair_table.cpp` with a text editor and comment out the last line (`return . . . .`), substituting it with:

   ```
   return NULL ;
   ```

This step aims to avoid errors when using different cut-off values when using for example hybrid pair styles with Coulomb interactions and a long-range solver, such as pppm or ewald.

3. Build with the **GPU package**:

   (a) first, install the `nvidia-cuda-toolkit` package, available on the NVIDIA web-site or by typing the command:

   ```
   sudo apt-get install nvidia-cuda-toolkit
   ```

   we have also to install the GPU drivers. On Ubuntu the open-souce package `nouveau` is installed by default, but probably only with the proprietary Nvidia drivers we can have the maximum performance. To install them we have to add the appropriate repository:

   ```
   sudo add-apt-repository ppa:graphics-drivers/ppa
   ```

   and then we can install the recommended latest stable version with the command:

   ```
   sudo ubuntu-drivers autoinstall
   ```

   or select the another version manually with the utility `Software&Updates`. Sometimes after this process the system could not work properly anymore, due to incompatibility problems. If this happens, you can restart the system and press the keys `Alt + F2` or `Alt + F1` before logging in, to open a root terminal. In this terminal we can give the commands `sudo apt-get purge nvidia-*` and `sudo reboot` to restore the situation.

   (b) Give the commands:

   ```
   export PATH=$PATH:/usr/local/cuda/bin
   export CUDADIR=/usr/local/cuda
   export CUDAHOME=/usr/local/cuda
   export LD_LIBRARY_PATH=/usr/local/cuda/lib64
   ```

   then open the file `lib/gpu/Makefile.linux` and modify the line `CUDA_ARCH = -arch=sm_`*XX*, where *XX* is a two digits number. The latter are the two digits of the Compute Capability of the GPU. [1] Then within directory `lib/gpu/` give the command:

   ```
   make -f Makefile.linux
   ```

[1] More information about this procedure can be found in the file `lib/gpu/README` and on the respective web-pages of the LAMMPS user-guide.

(c) If the system is not able to find the CUDA libraries we probably should modify the paths where it has to search for them, contained in the file `lib/gpu/Makefile.lammps`. Anyway, this shouldn't be needed. It is worth noting that the procedure could not work using the version 10.0 of Cuda (probably because the place of some libraries is different with respect to the older versions).

4. Build with the **OPT package** (this package contains some optimised code for the `compute` commands relative to several pair-styles):

   Only in case one uses an Intel compiler (needed if one wants to build LAMMPS with the USER-INTEL package) add the option `-restrict` to `CCFLAGS` in the file `src/MAKE/Makefile.ubuntu`, otherwise no preliminary changes must be done.

5. Build with the **USER-OMP package** (an optimisation package which allows to use multiple threads per mpi process):

   Open the file `src/MAKE/Makefile.ubuntu` and add the flag `-fopenmp` to `CCFLAGS` and `LINKFLAGS`. The flags will be different in case of using the Intel compiler (see the corresponding section of the online manual).

6. Build with the **USER-VTK package**:

   (a) install the vtk library with the command:

   `sudo apt-get install libvtk7-dev` (or other versions)

   (b) Open the file `lib/vtk/Makefile.lammps`, which has to contain three similar lines:

   ```
   vtk_SYSINC = -I/usr/include/vtk-7.1
   vtk_SYSLIB = -lvtkCommonCore-7.1 -lvtkIOCore-7.1
        -lvtkCommonDataModel-7.1 -lvtkIOXML-7.1
        -lvtkIOLegacy-7.1 -lvtkIOParallelXML-7.1
   vtk_SYSPATH = -L/usr/lib/x86_64-linux-gnu
   ```

   `vtk_SYSINC = -I(...)` indicates the directory where the VTK library has been installed. `vtk_SYSLIB` indicates to the compiler the name of the libraries (they change upon the installed version of VTK). `vtk_SYSPATH = -L(...)` is the path where the compiler has to search for the libraries. These directories could be detected, for example, with the command:

   ```
   find / -type f -name *libvtkCommon*
   ```

   to see where the libraries actually are.

7. Build with the **USER-INTEL package**:

   In this case an Intel compiler is needed, otherwise the package will not work. It has not been tested yet. If an Intel compiler is already installed, give the command:

   ```
   cp src/MAKE/OPTIONS/Makefile.intel_cpu_openmpi src/MAKE/
   ```

   and modify that Makefile instead of Makefile.ubuntu in steps 2-6.

8. Go to the directory `src/` and do (according to what packages you want to install):

   ```
   make yes-user-omp
   make yes-molecule
   make yes-kspace
   make yes-gpu
   make yes-opt
   make yes-user-vtk
   make yes-user-intel
   ```

   `make ubuntu -j`*N* (`make intel_cpu_openmpi`, in case the Intel compiler is used)

   where in the last command *N* is the number of processors the compiler is allowed to use. An executable `lmp_ubuntu` is created (`lmp_intel_cpu_openmpi` in case you compile the Intel makefile).

9. Make a directory in the home called `.bin`, enter into it and write:

   ```
   ln -s $HOME/lammps-<release_date>/src/lmp_ubuntu $PWD/lmp
   ```

   A link called lmp is created.

10. Within directory `$HOME` open the file `.bashrc` and write at the end:

    ```
    PATH=/home/<user_name>/.bin:$PATH
    ```

    Then save and do:

    ```
    source .bashrc
    ```

If errors appear during compilation ask the administrator of the system to install the missing files.

Chapter 2

The init file

The init file requires a specific format.

It is made of sections that are added depending on the system treated. The header and the `Atoms` section **must** be present in the file. Moreover, single sections may require specific information always depending on the system.

Follow these steps *to the end* to understand what should be included in the init file:

1. choose a system among these ones in the **third column** of the table. Warning: if your system does not correspond to the ones listed you may need to combine multiple styles (see below);

2. the **second column** of the same table tells you the quantities that have to be provided in the init file either as atom-specific quantities in the section `Atoms` or as specific sections of the init file;

3. the **first column** of the the same table tells you the `atom_style` that has to be reported in the input script (see 3.1.2). Warning: in case you need to combine multiple styles, you will have to use `atom_style hybrid`, see 3.1.3;

4. check the "Atoms" section in this page (just search "Atoms section:"), to see the per-atom quantities that have to be provided in the init file for each atom for the style chosen. Warning: in case you are using an hybrid atom style you have to provide, for each atom, `atom-id atom-type x y z sub-style1 sub-style2` where `sub-style1` and `sub-style2` are the information of the first and of the second atom style you are combining that are NOT `atom-id atom-type x y z`. Specific examples are reported in Chapter 5;

5. other details required by the chosen atom style that are not indicated in the Atoms section, have to be included in the init file as separate sections, always listed in this page;

6. the Velocity section can always be included for all the atom styles;

7. the Header of the init file (that is the first part that has to be included in the init file) contains, depending on the system:

   (a) first line:
   `description of the script` (not read in)

   (b) number of "components" of the system:
   ```
   # atoms
   # bonds
   ...
   ```

   (c) type of "components" of the system (**the number of types allows to distinguish how different atoms in the system interact among themselves**):
   ```
   # atom types
   # bond types
   ...
   ```

   (d) box dimensions:
   ```
   -25.  25 xlo xhi
   -25.  25 ylo yhi
   -25.  25 zlo zhi
   ```

   (e) masses for each type of atom:
   ```
   Masses
   1 1
   2 1
   ...
   ```

   Check here which components you have to include (just search "Format of header of data file");

8. the next part of the init file, following the Header, is the Atoms section. The others, if any, follow in turn.

Respect blank lines!

An example of a full init file for bead-spring polymer with stiffness (that includes also Bonds and Angles other than the Atoms sections) is reported here.

Chapter 3

Prepare the input script

The input script can be divided into six sections:

1. General settings
2. Configurations and trajectories for post-processing
3. Interaction Potentials
4. On-the-fly computations
5. Thermodynamic (and other) information
6. Minimize/Run

Below are listed all the commands that are usually common for all kind of simulations. Additional commands required to treat specific cases are explained in Chapter 5.

For each command, we provide the syntax required by LAMMPS, a short explanation, some suggestions on how to use that command in common situations and related commands that could complement its function.

3.1 General settings

3.1.1 `units`

`units` units

Define the units of the simulation.

Specify the units of the system. Use `lj` for typical adimensional units.

3.1.2 `atom_style`

`atom_style` style

It defines the type of system you are dealing with.

Use atomic for single particles, bond for molecules. Depending on the chosen style, the information provided in the init data, and thus the way they are read by the program, differ (see init data section). See section 'how to...?' for further details on the hybrid atom style, in case you need to mix multifold styles. A table with the systems treated by LAMMPS: is given here. Alternative to: 3.1.3

3.1.3 `atom_style hybrid`

`atom_style hybrid` sub-style1 sub-style2

It allows to couple multiple built-in atom styles to define the system in the simulation.

In later sections we show a specific example where charged polymers are simulated. In that case, for example, one need to couple the charge with the bond style. Warning: the use of `atom_style hybrid` implies a specific format for the init file (see Chapter 2). Alternative to: 3.1.2

3.1.4 `boundary`

`boundary` x y z

Box boundary conditions.

Use `p p p` for periodic boxes in 3D.

3.1.5 `neighbor`

`neighbor` skin style

Define the skin distance to construct the neighbor Verlet list.

The default, which has been successful up to now, is 0.3 bin.
Related to:

- `neigh_modify every` 1 `delay` 0 `check` yes ...
  This command sets the procedure to construct the Verlet list. The keyword `every` sets the number of steps between two subsequent constructions, whether `delay` allows a list construction only if a certain number of steps have been passed since the previous construction. With `check` yes the list is actually constructed only if one atom has moved farther than half the skin value.

3.1.6 `read_data`

`read_data` filename

Read the init file (where details of the system are provided).

For details on how to build the init file see the dedicated section. Alternative to: 3.1.7

3.1.7 `read_restart`

`read_restart` filename

Read the binary restart file (which contains details of the system AND details of the simulation such as potentials, thermodynamic outputs etc.).

Use to restart a simulation in case it stopped unexpectedly or after an equilibration procedure. Warning: NOT ALL the settings of a simulation are read in: those that are included and those that are not are described in its LAMMPS' page. Alternative to: 3.1.6.

3.1.8 `reset_timestep`

`reset_timestep` n

Set the timestep counter to n.

3.1.9 `group`

`group` group-id `type` type-id1 type-idn

Define groups of atoms of the same type. Other styles besides `type` may be used (e.g. `molecule`).

Define groups to perform subsequent operations on a specific group of atoms.

3.2 Configurations and trajectories for post-processing

3.2.1 `dump`

`dump` dump-id group-id `custom` n filename `id x y z vx vy vz...`

Print information for the specified group of atoms every n step.

The `custom` style allows customized output. In this example, the atom-id, the positions and the velocities for each atom are printed (see dump page for other options besides positions and velocities). This file can be used

for post-processing, it is not a binary file.
Related to:

- `dump_modify` dump-id `format line "%d %.10f ..."`
  Depending on the output specified, the format has to be adjusted.
- `dump_modify` dump-id `pbc yes`
  Print configurations taking into account pbc, such that all the atoms have a position into the simulation box.
- `dump_modify` dump-id `sort id`
  Atoms are ordered according to their id.

To save the **trajectory** of a group of atoms (the format is read by VMD), use:

- `dump` dump-id group-id `atom` n filename`.lammpstrj`
- `dump_modify` dump-id `pbc yes`

Sometimes it is convenient to save the system information on a specific timescale. In particular, the log scale allows to observe the behavior at microscopic and macroscopic times in correlation functions. The set of n logarithmically spaced measures is called a cycle. The time step will be defined as:

$$t_n = A \cdot base^n \tag{3.1}$$

where A is the number of cycles and n is an incremental index. Thus, we store each t_n in a file called *timestep.dat* that will be later read to specify the step in which the system configuration has to be dumped. The script reads as follow:

Listing 3.1 Log-timescale definition for dump command

```
1 variable      f file timestep.dat
  variable      s equal next(f)
  dump          conf all custom 100 dumpconf*.dat x y z vx vy ↩
    ↪ vz
  dump_modify   conf every v_s
```

1. The *timestep.dat* file contains a list that stores the temporary steps at which the configuration has to be saved. This file is pointed by the variable *f*.
2. We make a loop on the values that *f* points to and are stored in the *s* variable.
3. We specify the stored information.
4. By means of the argument *every* we indicate that the path is stored in all the temporal steps indicated by means of the variable *v_s*.

Warning: The first value stored in *s* comes from the first iteration, so the first line of the *timestep.dat* file will not be read. We also note that at the end of the file there must be a blank line in order to to allow for reading the last timestep from the list.

3.2.2 `restart`

`restart` n filename

Write a binary restart file every n timesteps to a file called filename.ntimesteps.

3.3 Interaction potentials

3.3.1 `pair_style`

`pair_style` style args

Define the potential between **non-bonded** atoms.

A list of the available potentials (with the args required) can be found in the command webpage. In this case, we are assuming the non-bonded interactions are all equivalent. To include multiple pair potentials or table potentials see the next commands. Alternative to: 3.3.2, 3.3.3.
Related to:

- `pair_modify shift yes`
  The potential is shifted to 0 at the cut-off radius.

3.3.2 `pair_style hybrid`

`pair_style hybrid` style1 args style2 args

This allows to use multiple pair potentials in the same simulation.

Each pair style keeps its argument. The pair coefficient command requires the pair potential name to be specified after the type-ids. Multiple instances of the same sub-style can be used (for example several lj/cut styles with different cut-offs). In such a case a further number has to be specified in the `pair_coeff` command, indicating which instance of the sub-style it refers to.
Warning: this pair-style has accelerated versions with packages USER-OMP, OPT, GPU, KOKKOS, USER-INTEL. Currently (December 2018) the GPU package is not able to treat multiple instances of the same sub-style (the package KOKKOS compiled for GPU should work instead, but no tests have been done up to now). Clearly the best performances can be obtained only if all the used sub-styles have an accelerated version.
Related to:

- `pair_style hybrid/overlay` style1 args style2 args
  In this case the potentials are superposed in an additive fashion. With the pair-style `hybrid` only one sub-style can be assigned to each pair of atom-types, and a new `pair_coeff` command for the same pair will overwrite the previous ones. With the pair-style `hybrid/overlay`, instead, several `pair_coeff` commands for the same pair allow for the superposition of the related sub-styles.

3.3.3 `pair_style table`

`pair_style table` linear n keyword

This allows to specify a potential that is not already built-in. It creates interpolation tables from potential energy and force values listed in a file as a function of the distance.

Use `pair_style table linear` n for linear interpolation among the points provided. By passing to the program a quite large number of points, the results are satisfactory even without using more sophisticated techniques. Pair styles table can be used as hybrid pair style. Warnings:

- the format required by the table can be found here (just search "format of a tabulated file")
- If the potential is used to minimize the energy of a system, some atoms could be found at a distance smaller than the one provided in the table: either the first point is set at smaller distance or the minimization is carried out by employing a built-in LAMMPS potential.
- With multiple tables in the same simulations, better use the same number of points and same values of the positions in all of them
- Add parts of the potential step by step
- The command special_bonds also works (and has to be applied) with tables
- If you are making consistency checks between your tabulated potential and the potentials that LAMMPS already uses, verify via `pair_write` the actual shape of the latter (for instance, the LAMMPS FENE potential does not diverge at the maximum bond length but, at a certain point, it is truncated and remains constant; this also depends on the LAMMPS version).

Related to:

- `pair_coeff` type-id1 type-id2 filename keyword (cutoff)
  The keyword is related to the table format and the filename refers to the file where the table is saved.

3.3.4 `pair_coeff`

`pair_coef` type-id1 type-id2 coeffs

Specify the coefficients and the cut-offs of the related pair-style.

Depending on the potential chosen the coefficients vary and they are listed in the potential web page. In case the potential is the same for all the combinations of types, * can be used for both type-ids; otherwise single types have to be specified inserting multiple pair_coeff commands.

3.3.5 `pair_write`

`pair_coeff` type-id1 type-id2 n `r` rmin rmax filename

Print a potential built-in or read by LAMMPS using n points linearly distributed from a distance rmin to a distance rmax to a file called filename.

Other styles than `r` may be used. The file is not overwritten thus it has to be removed to avoid later appendings.

3.3.6 `bond_style`

`bond_style` style

Define the potential of the **bonded** atoms.

A list of bond styles is provided in the style web page. Warning: check if your bond style requires the `special_bonds` command.
Related to:

- `special_bonds` style
  This applies a reweighting of the interactions such that the bond_style is computed only for the truly connected atoms. Warning: if you need to compute the radial distribution function restrictions may apply. Check the command web page.

3.3.7 `bond_coeff`

`bond_coeff` coeffs

Define the coefficients of the bond potential.

3.3.8 `fix`

`fix` fix-id group-id style args

This fix allows to specify the integration scheme to be applied to the entire group whose name is group-id.

The **NVE** ensemble is reproduced by simply using

- `fix` fix-id group-id `nve`

Simulations in the NVE ensemble can be performed by limiting the displacement of the particles by

- `fix` fix-id group-id `nve/limit` xmax

where *xmax* is the maximum allowed displacement for every particle in a timestep. This could be convenient in the case of minimizing the initial energy of a system from a configuration where the positions of the particles were randomly drawn (see below).
The **temperature can be fixed** using a Nose-Hoover thermostat as:

- `fix` fix-id group-id `nvt temp` start-temp stop-temp damp-temp `tchain` nchains
  The dump-temp is the relaxing time of the NH chains.

Other thermostats may be used. Warning1: their specific `fix` HAS to be coupled to the basic integration scheme, that is `fix` fix-id group-id `nve`. The Nose-Hoover thermostat does not require this operation. Warning2: One has to be careful in setting correctly the parameters tchain and dump-temp because wrong values may produce

unwanted effect in the simulated temperature. One can start trying with tdamp equal to 100 times the given timestep and then, if it doesn't work, increase the value. The default value for tchain is 3 (a higher value gives in principle a better thermalization of the system). We report here the Langevin and Berendsen thermostats:

- `fix` fix-id group-id `langevin` start-temp stop-temp damp-temp seed keywords
- `fix` fix-id group-id `temp/berendsen` start-temp stop-temp damp-temp

Warning 3: In general, the damp factor is proportional to the viscosity. However, Lammps considers an inverse proportion, and hence, decreasing the damp factor we are increasing the viscosity. Warning 4: To integrate the equation of motions, fix langevin must be preceded by fix nve.
The DPD thermostat works instead as a pair_style as:

- `pair_style dpd/tstat` start-temp stop-temp cutoff seed

The basic integration scheme, that is `fix` fix-id group-id `nve`, has to be added in this case.

3.4 On-the-fly computations

3.4.1 `compute`

`compute` compute-id group-id quantity

Compute quantity while running; this can be for instance the center of mass or the gyration radius (`com, gyration`).

To output this quantity c_compute-id can be added to the thermo_style custom list of outputs or averaged over time by means of fix ave/time.
Related to:

- `fix` fix-id group-id `ave/time` nevery nrepeat nfreq compute-id `ave running file` filename `overwrite`
  This fix takes as input what's been computed by compute-id and average over time. The quantity is averaged over time every nfreq times by taking the previous nrepeat values every nevery steps. The output file is overwritten every time this operation is performed.

3.4.2 `compute chunk/atom`

`compute` compute-id group-id `chunk/atom` style args

It assigns the atoms in the group group-id a different number (chunk) according to the style chosen. This allows to distinguish atoms in a certain region of the system. For instance, one could create several 3D rectangular or spherical bins: atoms in different bins will be assigned a different chunk.

In case the radial density profile has to be computed the style args would be `bin/sphere` x y z rmin rmax nbins; This assigns to the atoms the same ID in case they are found in the same shell of a sphere centered in x y z. Nbins are created from a radius rmin to a radius rmax. It allows the calculation of a quantity that varies

radially. Warning: to output an average quantity the compute has to be coupled to a fix ave/chunk that perform the average.
Related to:

- `fix` fix-id group-id `ave/chunk` nevery nrepeat nfreq compute-chunk-atom-id style `ave running file` filename `overwrite`
  This fix takes as input what's been computed by compute-chunk-atom-id and output the quantity defined by style according to the spatial partition performed by compute-chunk-atom-id. The quantity is averaged over time every nfreq times by taking the previous nrepeat values every nevery steps. The output file is overwritten every time this operation is performed. Style examples are: vx ,vy, vz, fx, fy, fz, density/number or quantities defined in other computes or variables.

3.4.3 `variable`

`variable` name style arguments

This is a powerful command which can be used to store virtually every kind of information (strings, scalars, vectors, ...), according to the style, which specifies the variable's kind.

Styles `string` and `index`

Variables with style `string` can contain a single string, whereas `index` variables can be arrays of strings. After the definition, a variable `index` assumes the value of the first string, and it can be changed to the following values through the command `next`. When no more values are available, the last `next` command acts as `variable` name `delete`, which cancel the variable.

Listing 3.2 Example of use of index variables.

```
1 . . .
  variable init_name index molecule.dat
  read_data ${input_fname}
  . . .
  variable N_of_steps index 100000
6 run ${N_of_steps}
  . . .
```

Another worthy difference among these two kinds of variable is that the `string` type can be re-defined throughout the script, whether the `index` type cannot, and commands attempting to do that are ignored. In this way some parameters can be passed to the script from command-line options. This is particularly useful if we have to **launch several simulations of the same kind**, varying the init-file and the number of simulation steps, for example (see the dedicated Chapter 4).

Style `loop`

This kind of variable is defined to allow the creation of loop cycles within the input file (script). The command's syntax is:

`variable` i `loop` N

`variable` i `loop` N_i N_f

In the first case an array of values from 1 to N is assigned to the variable i the same way as for an index-type variables, whether in the second case the range of assigned numbers starts from N_i instead of 1.

Listing 3.3 gives an example of cycle in an input script. The two commands `label` indicate the start point and the end point of the cycle. The command `if` controls if the last iteration has been reached that is when the command `jump SELF loop_end` is executed and the cycle is exited.`jump` tells LAMMPS to read commands from another input file, starting from the line signed by a label. The argument `SELF` refers to the actual input script. After each iteration the variable i is incremented through the command `next` and then the cycle is restarted.

Listing 3.3 Example of cycle within an input script.

```
label loop_start
variable i loop 10
. . .
if "$i == 10" then "jump SELF loop_end"
next i
jump SELF loop_start
label loop_end
variable i delete
. . .
```

Styles `equal`, `vector` and `atom`

These kinds of variable are defined through the command:

`variable` var `equal` scalar_expression

`variable` var `vector` vector_expression

`variable` var `atom` atom_expression

where for expression is intended a formula which is evaluated every time the variable is used. For `equal`-style variables this expression has to produce a scalar quantity (such as the kinetic energy `ke`, for example), otherwise an error occurs. For `vector`-style ones it has to produce a global vector (such as the center of mass `com` of a group of atoms, or the stress tensor obtained by means of the compute `pressure`). Finally, for `atom`-style variables it has to give a per-atom vector (such as the x-component of the velocity `vx`).

The expression may contain references to several predefined quantities and built-in functions, as well as other variables, and the output of computes and fixes defined in the input script, which can be accessed via `c_compute-id` and `f_fix-id`. An extensive list of the names of all these quantities and functions can be found here.

Evaluation of variables

At a certain point of the input script, we may need to get the value of a variable, for example to define a formula that has to be passed to a `compute` or `fix` command, to calculate some parameter, or to print some thermodynamic value. Given a variable `var`, there are two ways to evaluate it:

- writing `v_var` (or `v_var[i]`, if it is a vector, i being an integer number), which leads to the sudden evaluation of the variable;
- writing `${var}`, which leads to the conversion of the variable in a string and the subsequent substitution within the line of the script, which is finally parsed.

These two methods produces different results when `var` is an `equal`-, `vector`- or `atom`-style variable.
To clarify how they are employed we can imagine to use the `equal`-style variable `zappa` (that contains for example a thermodynamic formula) within a specific `compute` command, which is periodically called during the simulation. When using `v_zappa`, every time the compute is called, the formula is evaluated, updating the value of `zappa`, which is finally used in the computation. Contrarily, when using `${zappa}` the variable is evaluated only at the definition of the compute, when it is substituted with its value at that moment and no longer updated. For this reason `v_zappa` cannot be used in all the fixes and computes. Warning: some computes and fixes do not accept the use of variables. If this is the case, in the command webpage there is no explicit reference to them.

Immediate variables

Immediate variables are expressions which are not associated to a name, but are suddenly evaluated and used. They can be formulated as if they were equal-style variables and are enclosed within parenthesis preceded by the symbol `$`: `$( ··· expression ··· )`.

3.4.4 `fix recenter`

`fix` fix_name group1-ID `recenter` x y z `shift` group2-ID

This allows to constrain the center of mass of atoms belonging to group1 to the position (x,y,z). With the keyword `shift` we specify the group of atoms that are shifted along with group1 (usually all). The values x, y and z could be also set to `INIT` (which indicates their initial value) or `NULL` (in such a case the shifting operations are not performed in the respective direction). Warning: this fix does not work for big displacements. In this case, use `displace_atoms` before, see section 5.1.

3.5 Thermodynamic (and other) information

3.5.1 `thermo`

`thermo` n

The thermodynamic quantities are printed every n timesteps to screen and in a file called log.lammps.

Related to:

- `thermo_style custom step temp etotal pe ke press c_compute id`
  The output can be custom. In this case for example the timestep, the temperature, the total, potential, kinetic energy, the pressure and the output of the compute called compute-id are printed.

3.6 Minimize/run

3.6.1 `timestep`

`timestep n`

Define the value of the timestep for the simulation.

3.6.2 `minimize`

`minimize` energy-tolerance force-tolerance max-iterations max energy/force check

Perform an (non-physical) energy minimization of the system.

It can be used before run in order to avoid too high energies or atoms lost at the beginning of the simulation. The use of 1.0e-4, 1.0e-6, 10000, 10000 typically gives satisfactory minimizations. Warning: in case the energy minimization is performed using a table potential errors and lost atoms may occur following a too short inner and outer cutoffs. To avoid this, it is more convenient to minimize the energy of the system using a built-in potential and subsequently change it. Other possibility would be to run an nve simulation limiting the displacement of the particles by fix nve/limit (see above for a complete description). For a satisfactory minimization, we also recommend to include

`fix` fix ID group-ID momentum N keyword values ...

Thus, we reset the momentum of the particles every N timesteps (see the corresponding section of the online manual). In that case, we can use any kind of potential during the minimization.

3.6.3 `run`

`run n`

Perform the simulation for n timesteps.

Chapter 4

Launch a simulation

A simulation is started by

```
lmp < in.script
```

or, equivalently, by

```
lmp -in in.script
```

You may want to use multiple processor of your node. You do this with mpirun

```
mpirun -np #processors lmp < in.script
```

Best performances are reached using isolated nodes, with the total number of processors employed not higher than the number of physical processors in your node (just count the processors, not the total number of cores). This information can be found via the `lscpu` command.

Other useful flags (to be given after the `lmp` command):

```
-e screen -l none
```

to avoid the production of the log file, redirecting all the information to the standard output, together with the thermodynamics output;

```
-var <vname>
```

to overwrite the definition of a variable in the input file;

```
-sf omp -pk omp <N_threads_per_mpi_process>
-sf gpu -pk gpu <N_gpus>
```

to avoid changing the input file when using accelerated versions of the pair/bond styles, fixes, computes.

Chapter 5

Study cases

5.1 Equilibration procedure and initial settings

One could setup a simulation following several routes:

1. from a binary restart file (see also 3.1.7), that already contains both system and simulation details. Warning: not all the information of the system are saved in the restart file, check the restart page to see which are and are not included;
2. from a series of atoms already listed in an init file. In this case, the init file is prepared following Chapter 2, and read in via 3.1.6; other atoms can be included successively (see below);
3. from a pre-defined box via an init file and no atom in the box yet. The init file contains 0 atoms, n. atom types, but the box is defined and read in via 3.1.6;
4. directly via an input script, starting from scratch, by defining a region and creating a simulation box in the input script (see below).

Atoms can be created or added to others already existing via

`create_atoms` type style args keyword value

To create n atoms randomly in the box use `create_atoms` type `random` n seed `NULL` where type always refer to the type-id that the new atoms will adopt. Also convenient is the creation of atoms in specific regions of the box. This is achieved by substituting `NULL` with the region-id. Of course, one or multiple regions have to be previously defined via the `region` command:

`region` region-id style args

where the style depend of the shape that region should assume. See the command webpage for further details.
The `region` command is also useful to create a simulation box from scratch. In fact, the region-id previously defined is taken by `create_box` as input:

`create_box` n-atom-types region-id keyword value

Non-orthogonal boxes can be created. This is needed for example in simulations where shear is applied, see **??**.
Once atoms are created into the simulation box, a useful (sometimes compulsory) operation to be performed is the energy minimization, that is a one-time non-physical displacement of the atom that LAMMPS perform in order to reduce the energy due to their possible overlap. To perform this operation see 3.6.2.

Sometimes atoms or groups of atoms have to be displaced by large quantities (for example if the initial configuration is taken from a different set of simulations). To this aim, use

`displace_atoms` group-ID style arguments keyword value ...

It allows to manipulate the positions of the particles. For example at the beginning of the simulation one may need to bring some molecule at the center of the simulation box. This can be done with the command `displace_atoms` all `move` c_moleculeCOM[*], where the moleculeCOM is the result of the `compute com` applied only on the atoms in the molecule. For more information about the other possibilities visit the documentation page.

To manipulate the velocities of the particles, use

`velocity` group-ID style arguments keyword value ...

For example at the beginning of the simulation one may need to cancel the center of mass velocity and the total angular momentum, which can be done with the commands `velocity` all `zero linear` and `velocity` all `zero angular`. For more information about the other possibilities visit the documentation page.

5.2 Umbrella sampling

The umbrella sampling procedure is typically performed by imposing an additional spring potential among two objects in a simulation. In LAMMPS this can be done with the command fix spring, that apply a spring force between two groups of atoms. The syntax is the following:

```
fix fix-id group-id1 spring couple group-id2 k x y z R0
```

where k is the spring constant while x y z define the vector that connect the two groups. For normal purposes, x y z define the equilibrium distance of the spring and the value of R0 should be set to 0. This would be an additional fixed distance that the coupled object would be away from the equilibrium point (it seems that R0 plays the role of a fixed-distance wire connected to the spring that effectively keeps the two groups at a distance R0 one each other imagining a the spring in a fully compressed state).
To monitor the spring lengths one can use the following script lines:

Listing 5.1 Example of use of index variables.

```
compute commgel1 mgel1 com
compute commgel2 mgel2 com

variable comdistancex equal c_commgel1[1]-c_commgel2[1]
variable comdistancey equal c_commgel1[2]-c_commgel2[2]
variable comdistancez equal c_commgel1[3]-c_commgel2[3]

variable comdistance equal sqrt(v_comdistancex^2+ ↩
    ↪ v_comdistancey^2+ v_comdistance^2)
```

and add `v_comdistance` to the thermo line to output the distance of the spring at certain time steps.

Warning: be careful in the choice of the thermostat! The Nosè-Hoover thermostat introduces a bias in the sampling of the spring lengths while it appears that Langevin and DPD (with explicit solvent) do not have influences in this respect.

5.3 Polydispersity

In this chapter, we will present how it is possible to introduce directly or by an input file the polydispersity in our script, considering first a simple case of a Lennard-Jones binary

system, and then applying it to the case of systems where the potential is the sum of two contributions. Finally, we will show how we can introduce the polydispersity following a Gaussian distribution.

Listing 5.2 General setting

```
units           lj
boundary        p p p
atom_style      sphere

region          box block -5.0 5.0 -5.0 5.0 -5.0 5.0
create_box      2 box
create_atoms    1 random 1000 76702 NULL
```

5.3.1 Binary systems

A system with $N_{particles}$ is called binary system when there are two characteristic diameters, σ_1 and σ_2 beign $\sigma_1 > \sigma_2$. As we observe in Listing 5.2, once defined the units and the boundary conditions, the atom_style is defined as *sphere* because it allows us to specify as attributes the diameter, the mass and the angular velocity. For simplicity, let us consider a simulation box of $L_x = L_y = L_z = 10\sigma_1$ (we will define σ_2 later) and we introduce 1000 particles. This is specified in the second part of Listing 5.2, where the simulation box is defined by region. After, by create_box we declare the presence of 2 different types, although Lammps initializes particle types to 1, i.e., there are no distinctions. Then, we introduce 1000 particles in random positions using create_atoms.

Listing 5.3 Setting attributes

```
set             atom 300* type 2
set             type 1 diameter 1.0
set             type 2 diameter 0.5
set             type * mass 1
```

Now, we must specify the attributes of our particles. It is performed by set, as we can read in Listing 5.3:

1. Atoms labelled from 300 onwards are considered type 2.
2. Atoms of type 1 have diameter $\sigma_1 = 1.0$
3. Atoms of type 2 have diameter $\sigma_1 = 0.5$

4. All atoms have mass $m = 1.0$, independently of particle type.

In this way our system consists of $N_1 = 300$ and $N_2 = 700$ for types 1 and 2, respectively.

The following step is to take into account how the presence of polydispersity affects the coefficients that determine the interaction between particles. In this example, we have considered that particles interact by means of a Lennard-Jones potential:

$$V\left(r_{ij}\right) = 4\varepsilon\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] \tag{5.1}$$

where ε is the depth of the potential well, $\sigma_{ij} = 0.5\left(\sigma_i + \sigma_j\right)$ and r_{ij} is the distance between the particles i and j. In Listing 5.4, we show the effects of the polydispersity on the interaction coefficients.

Listing 5.4 Specifying the interaction among particles of the same and different types

```
1 pair_style        lj/cut 2.5
  pair_coeff        1 1 1.0 1.0 2.5
  pair_coeff        2 2 1.0 0.5 1.25
  pair_coeff        1 2 1.0 0.75 1.875
```

In the first line we define the interaction and its cut-off r_{cut}. Generally, the defined values by pair_style are overwritten by the value introduced in pair_coeff, in this case r_{cut}. However, we believe that it is good practice to explicitly write the value of the cutoff that would be applied if the system were monodisperse. That way, future revisions of the script will be easier. The next three lines define the interaction between particles among equal and different types.

- Type 1 - Type 1: In this case, $\varepsilon = 1.0$, $\sigma_{ij} = 1.0$ and $r_{cut} = 2.5\sigma_{ij} = 2.5$.
- Type 2 - Type 2: In this case, $\varepsilon = 1.0$, $\sigma_{ij} = 0.5$ and $r_{cut} = 2.5\sigma_{ij} = 1.25$.
- Type 1 - Type 2: In this case, $\varepsilon = 1.0$, $\sigma_{ij} = 0.75$ and $r_{cut} = 2.5\sigma_{ij} = 1.875$.

However, it is possible to simply define the interaction coefficients between particles of the same type, and then use the command pair_modify, which defines how σ_{ij} is calculated, and hence, it infers the interactions among different types from interactions between equal types.

Listing 5.5 Specifying the interaction by mix arithmetic

```
1 pair_style        lj/cut 2.5
  pair_coeff        1 1 1.0 1.0 2.5
  pair_coeff        2 2 1.0 0.5 1.25
  pair_modify       mix arithmetic
```

In Listing 5.5 we observe that pair_modify has as argument *mix arithmetic*, which indicates $\sigma_{ij} = 0.5\left(\sigma_i + \sigma_j\right)$ (see pair_modify page for other arguments). Several pair_style present a default pair_modify, as is the case of lj/cut which contains pair_modify geometric. We believe that it is a good habit not to leave anything by default in such a way that for the maintenance of the script in future uses is much more intuitive.

On the other hand, coefficients can be entered per file in two ways. The first would be to specify the interaction coefficients in the input file, as specified in Listing 5.6.

Listing 5.6 Input file for a polydisperse system

```
*LAMMPS Description
/*white space*/
/*white space*/
# atoms
# bonds
...
/*white space*/
2 atom types
/*white space*/
lvalue hvalue xlo xhi
lvalue hvalue ylo yhi
lvalue hvalue zlo zhi
/*white space*/
Pair Coeffs
/*white space*/
        1 1.000000 1.000000 2.500000
        2 1.000000 0.500000 1.250000
/*white space*/
Atoms
/*white space*/
1 1 1.000000 1.000000 0.672549 -1.935030 -0.406199
...
300 2 0.500000 8.000000 -2.922060 0.000386 -0.226118
...
```

In this way, the input script will look like this:

Listing 5.7 Specifying the interaction by input files

```
pair_style          lj/cut 2.5
pair_modify         mix arithmetic
read_data           input.file
```

It is important to note that the use of sphere as atom_style introduces additional information in the definition of the atoms in the init file. This information is the density of the particle $\frac{m_i}{\sigma_i^3}$, which is located in the fourth column.

The second way is using the command include so that, all settings for the force filed is put into a separate file. We consider this option better because it is much more flexible, easier to debug, and you still have a clean main input file. In this case, script overall view will be:

Listing 5.8 General setting

```
units           lj
boundary        p p p
atom_style      sphere

region          box block -5.0 5.0 -5.0 5.0 -5.0 5.0
create_box      2 box
create_atoms    1 random 1000 76702 NULL

include         Coeff.dat
```

where include acquires as argument the file name, in this case *Coeff.dat*, where we enclose Listing 5.4 or Listing 5.5 (see include page for more information).

5.3.2 Gaussian distribution for a random polydispersity

In a polydisperse system, the distribution of diameters follows a Gaussian distribution, whose variance determines the degree of polydispersity [%]. In the following script, we show how to introduce polidispersity through a Gaussian distribution.

Listing 5.9 Gaussian distribution for a random polydispersity

```
units           lj
boundary        p p p
atom_style      sphere

variable        i loop 1000
```

```
variable                sigma equal normal(1.0,0.1,200)

region                  box block -5.0 5.0 -5.0 5.0 -5.0 5.0
create_box              1000 box
create_atoms            1 random 1000 76702 NULL

label types
set                     atom ${i} type ${i}
set                     type ${i} diameter ${sigma}
set                     type ${i} mass 1
next i
jump SELF types
```

After the *Initialization*, we have defined two variables: the *i* loop variable, which will run until it reaches the number of particles, and the *sigma* variable that points to the *normal* function, which generates random numbers from a Gaussian distribution of mean $\mu = 1$ and variance $\delta = 0.1$ (the third value in the argument *normal* is the seed for the random generator). Then, the attributes are set by a loop cycle:

1. We attribute to each particle a different type (for simplicity and efficiency, the type will match the id of the particle).

2. Each type is associated with a diameter extracted from the random distribution pointed by the variable *sigma*.

3. Independently of the type, all particle have unitary mass.

5.4 Charged molecules

In the following we will instead discuss some code used to simulate a system consisting of charged molecules. As a specific example we will treat a coarse-grained bead-spring model of a charged polymer with free counter-ions balancing its bear charge. All the particles interact via a truncated and shifted Lennard-Jones potential (also known as Week-Chandler-Anderson potential, WCA) representing the steric repulsion; the polymeric beads interact also through a short-range attractive well, accounting for the hydrophobic attraction; finally, all charged particles interact via the Coulomb potential.

5.4.1 Initial settings

As usual, within the first part of the input file we define the simulation box, the unit system, and the kind of atoms we want to simulate (see Sec. 5.1). With the command `atom_style` we set the kind of information we want to store for atoms; in our case we need to use the style `hybrid`, which gives us the possibility to store information on both charge and bonds for each atom.

Listing 5.10 Simulation initial setting.

```
###     Box and units
                          . . .
3 ###
atom_style hybrid charge bond
###
###     Verlet list
                          . . .
8 ###     Read in atoms
                          . . .
```

Then we set the parameters for computing the Verlet-lists, discussed in section 3.1.5, and finally we get the information about atoms and bonds, which can be generated through the commands `create_atoms` and `create_bonds`, or can read in by an external init-file. The choice of the atom style unavoidably affect the structure of the section `Atoms` of the init-file. This part of the file contains a line for each atom, which brings information about the position and other attributes, depending on the specific pair-styles used. In our case we have to store (orderly) the id, the atom-type, the position's coordinates, the value of the charge (which has to be expressed as a floating point number, to avoid errors in the execution of the script), and the molecule's id. An example is given in listing (5.11).

Listing 5.11 Starting file example.

```
1                                . . .

  Atoms

    1  1  3.1453643336  38.1152721225  27.5177108967  0.0000000000  1
6   2  1  5.5701782840  -36.6845193937  26.8732446459  0.0000000000  1
    3  1  17.4161391890  10.6782565472  51.7533524675  0.0000000000  1

                                 . . .
```

5.4.2 Interaction potentials

Subsequently we define atom groups, needed to set the interactions among them.
With the `pair_style hybrid/overlay`[1] command we define all the interactions among the atoms in the system, with the respective cut-offs and optional parameters. In our case we use the shifted Lennard-Jones potentials to zero at the cut-off to obtain the WCA potential. This effect is obtained through the command `pair_modify shift yes`, which acts only on the Lennard-Jones potentials. In this example we have counter-ions with a radius smaller than the polymeric beads, then we have to use different `lj/cut` sub-styles to model the steric repulsion among counter-ion-counter-ion, counter-ion-bead, bead-bead. In this case of study we need to simulate a solvophobic attraction among the polymer beads, which is not a predefined pair-style potential of LAMMPS, so we have to read it from a tabulated file using the sub-style `table`, specifying the number of points in the table and the interpolation method.
The `pair_coeff` command allows to set the parameters of the various interactions (for information about each style visit the documentation page) and the set of atoms among which it acts.
The `kspace_style` command sets the k-space solver (here we use the P^3M algorithm for Coulomb interactions), which needs the estimated relative accuracy on forces calculation, and automatically determines the algorithm's internal parameters on the basis of this value and the selected cut-off. For such a system it has been evaluated that the true relative accuracy

[1] For this system the accelerated package GPU cannot be used, because errors will occur. For uncharged molecules, instead, it will produce correct calculations, but the performances will be quite the same obtained with the parallelisation through the shell-command `mpirun -np 8`, because an accelerated version of the FENE bonding potential does not yet exist, leading to a bottle-neck in the overall parallelisation. For more information about the use of the pair-style `hybrid` see section 3.3.2.

is about one order of magnitude greater than the estimated one. Once we have selected the desired accuracy we have to optimise the value of the Coulomb cut-off to look for the faster balancing among *k*-space and real-space computations. One can also include the short part of the Coulomb potential (comprising of the dumping function, see the theory about the Ewald sums) in the table, differently from the example shown below.
Finally, also the bonding potential has been tabulated. We need to tell LAMMPS not to compute the pair interaction among bonded monomers, because the WCA and solvophobic contributions are already included into the bonding potential. This is done through the `special_bonds`

Listing 5.12 Definition of the interaction potentials.

```
###
# Define groups ( atom type 1 is uncharged monomer, atom type 2 is counter-ion, atom type 3 is charged monomer )
###
group all type 1 2 3
group mgel type 1 3
group cions type 2
group ions type 3

# Pair interaction between non-bonded atoms
pair_style hybrid/overlay table linear 10000 lj/cut 1.122462 lj/cut 0.617354 lj/cut 0.112246 coul/long 8.0
pair_modify shift yes

pair_coeff 1 1 lj/cut 1 1.0 1.0
pair_coeff 3 3 lj/cut 1 1.0 1.0
pair_coeff 1 3 lj/cut 1 1.0 1.0
pair_coeff 1 2 lj/cut 2 1.0 0.55
pair_coeff 2 3 lj/cut 2 1.0 0.55
pair_coeff 2 2 lj/cut 3 1.0 0.1
pair_coeff 2 2 coul/long
pair_coeff 2 3 coul/long
pair_coeff 3 3 coul/long
pair_coeff 1 3 table tab_potentials.dat SOLVOPHOBIC 1.5
pair_coeff 1 1 table tab_potentials.dat SOLVOPHOBIC 1.5
pair_coeff 3 3 table tab_potentials.dat SOLVOPHOBIC 1.5
```

```
# kspace solver for long-range interactions
kspace_style pppm 1.0e-4

# Pair interaction between bonded atoms
bond_style table linear 1527
special_bonds lj 0.0 1.0 1.0 coul 1.0 1.0 1.0
## special_bonds fene is equivalent to:
#                lj 0.0 1.0 1.0 coul 0.0 1.0 1.0
## lj acts on lj interactions and all other pair styles, such as table

bond_coeff 1 tab_bonds.dat SOLVOPHOBIC_WCA_FENE
```

5.4.3 Preparation of the simulation

Before starting the simulation, in our example we displace the atoms in order to position the center of mass of our molecule (`mgel`) on the center of the simulation box (which is thought to be the origin of the reference system). To do that we need to calculate the center of mass of the group `mgel` through the command `compute mgel_com mgel com`. The actual computation happens only after the start of the simulation, if requested, therefore we need to tell LAMMPS to calculate it (in this case through the command `thermo_style`) and then give the command `run 0`, which actually does not evolve the positions and velocity of the particles, but calculates all the thermodynamic quantities. In this way we can use the compute `mgel_com` in the command `displace_atoms` without producing errors.
For a matter of convenience we constrain the center of mass of the molecule to stay fixed over all the duration of the simulation (we cannot do that if interested in the calculation of quantities such as the diffusivity). This is accomplished through the command `fix recenter` (whose fix is called here `hook`) which shifts periodically the coordinates of all the particles in order to leave the molecule's center of mass in the initial position (see section 3.4.4 for further information). Finally, the total velocity and the total angular momentum of both the molecule and counter-ions (and in turn of the whole system) are cancelled via the command `velocity`.

Listing 5.13 Preparation of the system.

```
compute mgel_com mgel com
thermo_style custom step c_mgel_com[*]
timestep 0.001          # integration algorithm
```

```
4 fix md_nvt all nvt temp 1.0 1.0 0.03 tchain 30
  reset_timestep 0
  run 0
  displace_atoms all move $(-1.0*c_mgel_com[1]) $(-1.0*c_mgel_com ↩
    ↪ [2]) $(-1.0*c_mgel_com[3])
  fix hook mgel recenter INIT INIT INIT shift all
9
  velocity mgel zero linear
  velocity mgel zero angular
  velocity cions zero linear
  velocity cions zero angular
```

Within the listing 5.14 we show an example of some computations one can do during the simulation.

A quantity often used to study molecules is the radius of gyration, which can be calculated by the `compute gyration`. To obtain the running average of this quantity we use the command `fix ave/time` (see section 3.4.1), which performs the average of the quantity `c_mgyr`. With `mode scalar` it is specified that this quantity is a scalar, `ave running` is self-explained, `file gyration.dat` tells where this average has to be saved, and `overwrite` imply that this file is re-written every time a new value of the average is calculated.

Finally we compute the radial density profile of the polymeric beads as a function of the distance from the center of mass of the molecule. To do this we have to build an hystogram that counts the number of particles within spherical shells. The hystogram is constructed by the command `compute chunk/atom bin/sphere`. The thickness of the shells is fixed, whereas their number is calculated automatically (see section 3.4.3). Also in this case we let LAMMPS to print a running average of this profile through the command `fix ave/chunk` (see section 3.4.2 for a more comprehensive explanation). In this case the keyword `norm` imposes the normalisation of the average hystogram.

Listing 5.14 Setting of the on-the-fly computations.

```
  ##   Computation of the gyration radius
2 compute mgyr mgel gyration
  fix gyr_radius mgel ave/time 1 1 2000 c_mgyr file gyration.dat ↩
    ↪ overwrite mode scalar ave running

  ##   Density profile around the center of mass
  #     Now I compute rmax and nbins for the density profiles :
7 variable dr equal 0.2175
```

```
variable r_max equal $(xhi-xlo)*0.5    # if the box is a cube
variable nbins equal floor(v_r_max/v_dr)
compute mgel_hysto mgel chunk/atom bin/sphere 0.0 0.0 0.0 0.0 $ ↩
   ↪ {r_max} ${nbins}
fix mgel_profile mgel ave/chunk 10 10 4000 mgel_hysto density/ ↩
   ↪ number file monomers_profile.dat overwrite norm sample  ↩
   ↪ ave running
                        . . .
```

5.4.4 Counter-ions generation

We devote this last section to the description of the procedure used to generate counter-ions, if at the beginning we have only the positions and charges of the molecule's beads.
In this example we first create counter-ions as neutral beads, placing them randomly outside a sphere containing the molecule. We compute the variable `sphere_rad` as the maximum distance of the beads from the center of mass of the molecule (which is in the origin), through the command `compute reduce max` applied to the atom-type vector `center_dist`. Then with the command `region` we define the place where counter-ions have to be generated, as the intersection of the space outside the sphere containing the molecule and the simulation box.
We finally generate counter-ions with the command `create_atoms`, and then we minimize the energy of the system (without the electrostatic contribution) to cancel eventual dangerous superpositions.

Listing 5.15 Generation of counter-ions.

```
## The center of mass of the molecule is in the origin
# Here we compute the maximum distance of the beads from it
variable center_dist atom sqrt(x^2+y^2+z^2)
compute max_dist mgel reduce max v_center_dist
variable sphere_rad equal c_max_dist

region simul_box block EDGE EDGE EDGE EDGE EDGE EDGE
## this command define the region wherein
#                counterions will be generated
region mgel_out sphere 0.0 0.0 0.0 v_sphere_rad side out
region mgel_outside intersect 2 mgel_out simul_box

create_atoms 2 random $(count(ions)) 9345419 mgel_outside
```

```
## Groups are static containers, when adding particles
#               they must be updated
group cions type 2
group all type 1 2 3
set type 2 mol 2

set type 2 charge 0.0
set type 3 charge 0.0
minimize 1.0e-4 1.0e-6 1000 10000
```

To make a first equilibration of the system we suggest an alternative to the use of the command `minimize`. Using the command `minimize` would lead almost all counterions to penetrate within the network, trying to minimize the potential energy forming dipoles. The excluded volume of counterions could in principle produce a lot of stress on chains in the network, which during the minimization process could result in the disentanglement events due to numerical errors, implying an unwanted change in the network topology. To avoid such situations we can make a cycle wherein we gradually raise the charge of ions and counter-ions to the values $1e^*$ and $-1e^*$. In that way counter-ions will penetrate the molecule's network smoothly. Listing 5.16 shows an example of this procedure (for an explanation of this sketch of script see section 3.4.3).

Listing 5.16 First equilibration of the charged system.

```
variable inloop_steps index 10000
label equilibration_loop
variable i loop 10
set type 2 charge $(-0.1*v_i)
set type 3 charge $(0.1*v_i)
run ${inloop_steps}
if "$i==10" then "jump SELF break"
next i
jump SELF equilibration_loop
label break
variable i delete
```