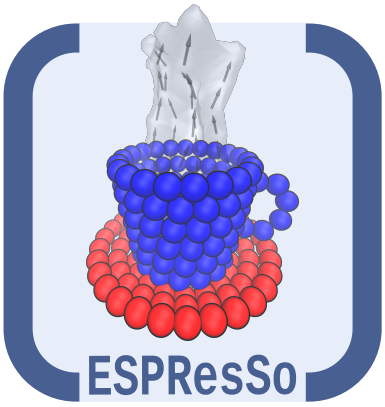


Tutorial 2: A simple charged system*

January 26, 2018

Contents

1 Introduction 1

2 Basic set up 2

3 Running the simulation 3

4 Writing out data 5

5 Analysis 6

6 Further ideas 9

1 Introduction

This tutorial 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 Python tutorial here; however, most of the constructs should be self–explanatory. We also assume that the

*For ESPResSo unknown

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 `pypresso <file>` from the ESPResSo source directory.

2 Basic set up

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 system parameters
n_part = 200
density = 0.7
box_l = (n_part/density)**(1./3.)
```

These variables do not change anything in the simulation engine, but are just standard Python 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 changing the espressomd.System() properties.

```
# Setup system geometry in Espresso
system = espressomd.System()

system.box_l = [box_l, box_l, box_l]
system.periodicity = [1, 1, 1]
```

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

```
# Place particles
q = 1.
ptype = 0

for i in xrange(n_part):
    q *= -1
    ptype = 1-ptype
    system.part.add(id=i, type=ptype,
                    pos=numpy.random.random(3) * system.box_l, q=q)
```

This loop adds `n_part` particles at random positions, one by one. In this construct, the `part` command 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 alternate between creating positive charges of type 0 and negative charges of type 1.

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

```
# Simulation parameters
system.time_step = 0.01
system.skin = 0.3

# Thermostat
temp = 1.
gamma = 1.
thermostat.Thermostat().set_langevin(kT=temp, gamma=gamma)
```

This switches on the Langevin thermostat for the NVT ensemble, with temperature `temp` and friction coefficient `gamma`. The skin depth `skin` is a parameter for the link–cell system which tunes its performance, but shall not 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 a hard core repulsion. Additionally the charges interact via the Coulomb potential:

```
# Lennard-Jones interactions
lj_sig = 1.
lj_eps = 1.
lj_cut = 2.**(1. / 6.)
system.non_bonded_inter[0, 0].lennard_jones.set_params(
    epsilon=lj_eps, sigma=lj_sig, cutoff=lj_cut, shift="auto")
system.non_bonded_inter[1, 0].lennard_jones.set_params(
    epsilon=lj_eps, sigma=lj_sig, cutoff=lj_cut, shift="auto")
system.non_bonded_inter[1, 1].lennard_jones.set_params(
    epsilon=lj_eps, sigma=lj_sig, cutoff=lj_cut, shift="auto")

p3m = electrostatics.P3M(bjerrum_length=10.0, accuracy=1e-3)
system.actors.add(p3m)
```

The first three `non_bonded_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 two lines set the Bjerrum length to the value 10, and then instructs ESPResSo to use the P^3M method for the Coulombic interaction and to try to find suitable parameters for an rms force error below 10^{-3}, with an optimal mesh size. This command will also print the parameters and timings that the tuning found to the screen. Tuning takes some time; if you run many simulations with similar parameters, you might want to save and reload the P^3M parameters. You can obtain the P^3M parameters by `get_params()`:

```
p3m_params = p3m.get_params()
for key in p3m_params.keys():
    print("{} = {}".format(key, p3m_params[key]))
```

3 Running the simulation

Now we can integrate the particle trajectories for a couple of time steps:

```
integ_steps = 200
int_n_times = 20

# Main integration loop
for i in xrange(int_n_times):
    temp = system.analysis.energy()['ideal'] / ((3 / 2.0) * n_part)
    print("t={0}, E={1}, T={2}".format(system.time,
                                       system.analysis.energy()['total'], temp))
    integrate.integrate(integ_steps)
```

This code block is the primary simulation loop and runs 20×`integ_steps` MD steps. Every `integ_steps` time steps, the potential, electrostatic and kinetic energies are printed out. The latter one is printed as temperature, by rescaling by the number of degrees of freedom (3) multiplied by $1/2kT$. Note that energies are measured in kT, so that only the factor $1/2$ remains. Also note that $3/2$ should be written as $3/2.0$; to ensure that Python does not perform integer division (resuting in 1 instead of 1.5).

Note, that in ESPResSo there are usually 3 translational degrees per particle. However, if `ROTATION` is compiled in, there are in addition 3 rotational degrees of freedom, which also contribute to the kinetic energy. You can get this number in the following way:

```
if "ROTATION" in code_info.features():
    deg_free = 6
else:
    deg_free = 3
```

Then all you need to do is to replace the hardcoded 3 by `deg_free`.

However, if you run the simulation, it will still crash: ESPResSo complains about particle coordinates being out of range. The reason for this is simple: Due to the initial random setup, the overlap energy is around a million kT, which we first have to remove from the system. In ESPResSo, this can be accelerated by limiting the maximum forces, i. e. modifying the Lennard–Jones force such that it is constant below a certain distance. Before the main integration loop, we therefore insert this equilibration loop:

```
integ_steps = 200

# Warmup integration loop
for cap in xrange(20, 200, 20):
    print("t={0}, E={1}".format(system.time,
                                system.analysis.energy()['total']))
    system.non_bonded_inter.set_force_cap(cap)
    integrate.integrate(integ_steps)
system.non_bonded_inter.set_force_cap(0)
```

This loop integrates the system with a force cap value of initially 20 and finally 180. The last command switches the force cap off again. With this equilibration, the simulation script runs fine. As a check that the equilibration loop works correctly, you might want to copy the energy and force printing from the main integration loop. You can then observe how the temperature initially overshoots and is then relaxed to its target value by the thermostat.

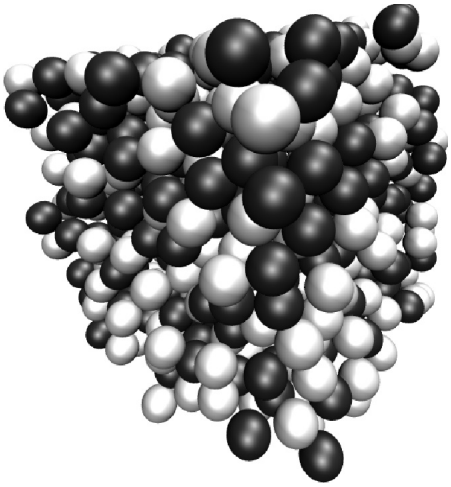


Figure 1: VMD Snapshot of the salt system

4 Writing out data

For later analysis of the simulation results, one will usually like to write out simulation data to configuration files. For this purpose one can use the `cPickle` Python library to write simulation data to a file in an easily parseable form. We add the following lines to the integration loop just after the `integrate` line in order to write out the system configuration files "config_0" through "config_19":

```
# Pickle particle data
with open("config_{}".format(i), "w") as configfile:
    pickle.dump(system.part, configfile, -1)
```

In order to write out `System()` variables such as the boxlength `box_l`, the timestep `time_step` or the skin `skin` to the file "system_config", one can insert the following lines before or after the main loop integration:

```
# Pickle system properties
with open("system_config", "w") as system_config:
    pickle.dump(system, system_config, -1)
```

Reading the system variables is done by the following simple script:

```
# Import system properties
system = espressomd.System()
pickle.load(open("system_config","r"))
```

The `load()` command will set the system variables to the values specified in the file. This includes to set for example the box dimensions for the simulation. Note that it is important to have the box dimensions set before reading the particles, to avoid problems with the periodic boundary conditions. The particles (including all properties) of the first frame can be restored with the following line:

```
# Import of particle properties
pickle.load(open("config_0","r"))
```

The `pickle` mechanism is typically used for one of two main purposes, checkpointing

and offline analysis. Checkpointing is useful for long-running simulations: If you have a simulation that runs for several days, you may want to have it periodically write its current state to a file. That way, the simulation can be resumed if your computer crashes while the simulation is running. Offline analysis refers to analyzing physical parameters of the simulation after the simulation has finished and will be discussed in the next section.

5 Analysis

Having written out pickle files during the simulation, 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 the current configuration can be calculated using the `rdf()` command from the `analysis` submodule:

```
# Calculate rdf
rdf_bins = 100
rdf_list_a = [0]
rdf_list_b = [1]
r, rdf = system.analysis.rdf(rdf_type='rdf', type_list_a=rdf_list_a,
                             type_list_b=rdf_list_b, r_min=0.9,
                             r_max=system.box_l[0] / 2., r_bins=rdf_bins)
```

The shown `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 `rdf_list_a` and `rdf_list_b` to either [0] and [0] or [1] and [1] allows to determine the distribution of equal charges around each other. The result are two numpy arrays containing the of r and $g(r)$ values. After the system properties are loaded, we loop additionally over all pickle files given as command line arguments to average over all configurations.

```
# Import system properties
system = espressomd.System()
pickle.load(open("system_config", "r"))

rdf_bins = 100
rdf_list_a = [0]
rdf_list_b = [1]

# Initialize the total RDF with zero
cnt = 0
avg_rdf = numpy.zeros(rdf_bins)

for filename in sys.argv[1:]:
    # Import of particle properties
    try:
        pfile = open(filename, "r")
        pickle.load(pfile)
        pfile.close()
    except Exception as e:
        print("Could not import particles from '{}'.".format(filename))
        print(e)
        continue

    # Calculate rdf
    r, rdf = system.analysis.rdf(rdf_type='rdf', type_list_a=rdf_list_a,
                                 type_list_b=rdf_list_b, r_min=0.9,
                                 r_max=system.box_l[0] / 2., r_bins=rdf_bins)
    avg_rdf += rdf

    system.part[:].remove()

    cnt += 1

avg_rdf *= 1. / cnt
```

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 `sys.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 again the `1. / cnt`; also here, this is necessary, since `1 / cnt` is interpreted as an integer division, which would result in 0 for cnt > 1. `sys.argv` is a predefined variable: it contains all the command line parameters. Therefore this script should be called like this:

```
./pypresso <script> config_*
```

The printing of the calculated radial distribution functions is simple. Add to the end

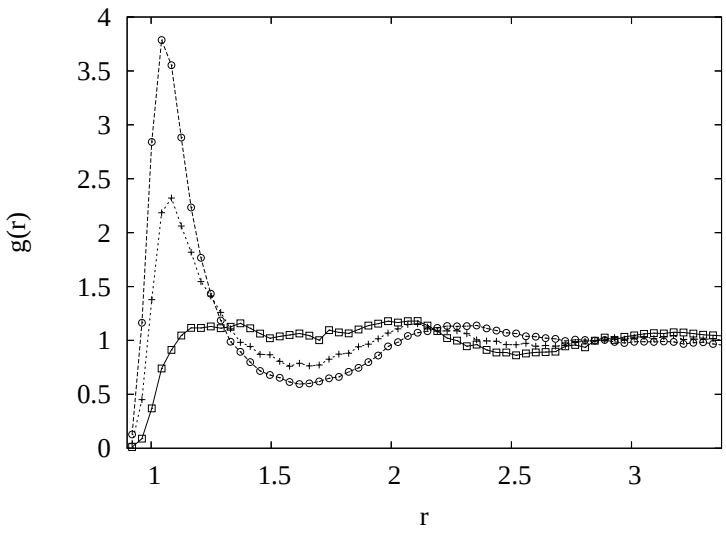


Figure 2: Radial distribution functions $g_{++}(r)$ between equal charges (rectangles) and $g_{+-}(r)$ for opposite charges (circles). The plus symbols denote $g(r)$ for an uncharged system.

of the previous snippet the following lines:

```
# Write averaged data to plot file
plot = open("rdf.data", "w")
plot.write("# r\trdf(r)\n")
for i in xrange(len(rdf)):
    plot.write("{0}\t{1}\n".format(r[i], avg_rdf[i]))
plot.close()
```

This instructs Python to write the `avg_rdf` to the file `rdf.data` in gnuplot–compatible format. Fig. 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 commands

```
p3m = electrostatics.P3M(bjerrum_length=10.0, accuracy=1e-3)
system.actors.add(p3m)
```

and therefore not turning on P^3M. To plot your own results, run `gnuplot` in a Terminal window and type the following:

```
plot "rdf.data" using 1:2
```

If you have written out RDFs for multiple combinations of species, you can plot them into one graph like this:

```
plot "rdf00.data" using 1:2 w l title "g++", \
     "rdf11.data" using 1:2 w l title "g--", \
     "rdf01.data" using 1:2 w l title "g+-"
```

6 Further ideas

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.