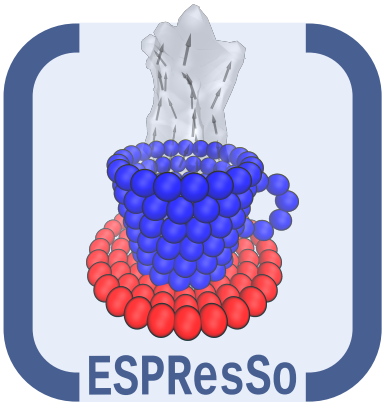


Tutorial 1: Lennard-Jones Liquid*

ESPResSo Basics

H.-J. Limbach M. Süzen K. Grass M. Sega
A. Arnold N. Gribova

January 26, 2018

Contents

1 Introduction **2**

2 Background **2**
2.1 Lennard-Jones Potential 2
2.2 Units 3

3 First steps **3**

4 Overview over a simulatino script **4**
4.1 System setup 4
4.2 Chosing the thermodynamic ensemble, thermostat 4
4.3 Placing and accessing particles 5
4.4 Setting up non-bonded interactions 5
4.5 Warmup 6
4.5.1 System Setup 7

*For ESPResSo unknown

4.6 Warmup 9
4.7 Integrating the equaitons of motion, taking measurements 10
4.8 Simple Error Estimation on Time Series Data 12
4.9 Online analysis of correlations 14
4.10 Other Useful Scripts 14

5 Binary Lennard Jones liquid **15**

1 Introduction

Welcome to the basic ESPResSo tutorial!

In this tutorial, you will learn, how to use the ESPResSo package for your research. We will cover the basics of ESPResSo, i.e. how to set up and modify a physical system, how to run a simulation, and how to load, save and analyze the data.

The more advanced features and algorithms available in the ESPResSo package are described in additional tutorials.

2 Background

Today's research on Soft Condensed Matter has brought the needs for having a flexible, extensible, reliable and efficient (parallel) molecular simulation package. For this reason ESPResSo (Extensible Simulation Package for Research on Soft matter) [1] has been developed at Max Planck Institute for Polymer Research, Mainz, and Institute for Computational Physics at the University of Stuttgart in the group of Prof. Dr. Christian Holm[2, 3]. The Espresso package is probably the most flexible and extensible simulation package in the market. It is specially developed for coarse-grained molecular dynamics (MD) simulation of polyelectrolytes but not necessarily limited to this. It can be used even in simulating granular media for example. ESPResSo has been nominated for the Heinz-Billing-Preis for Scientific Computing in 2003 [4].

2.1 Lennard-Jones Potential

A pair of neutral atoms or molecules is subject to two distinct forces in the limit of large separation and small separation: an attractive force at long ranges (van der Waals force, or dispersion force) and a repulsive force at short ranges (the result of overlapping electron orbitals, referred to as Pauli repulsion from Pauli exclusion principle). The Lennard-Jones potential (also referred to as the L-J potential, 6-12 potential or, less commonly, 12-6 potential) is a simple mathematical model that represents this behavior. It was proposed in 1924 by John Lennard-Jones. The L-J potential is of the form $V(r) = 4\epsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}]$ where ϵ is the depth of the potential well and σ is the (finite) distance at which the inter particle potential is zero and r is the distance between the particles. The $(\frac{1}{r})^{12}$ term describes repulsion and the $(\frac{1}{r})^{6}$ term describes attraction. The Lennard-Jones potential is an approximation. The form of the repulsion term

has no theoretical justification; the repulsion force should depend exponentially on the distance, but the repulsion term of the L-J formula is more convenient due to the ease and efficiency of computing r^{12} as the square of r^6.

2.2 Units

Novice users must understand that Espresso has no fixed unit system. The unit system is set by the user. Conventionally, reduced units are employed, in other words LJ units. [1]

3 First steps

What is ESPResSo? It is not a coffee, indeed. It is an extensible, efficient Molecular Dynamics package specially powerful on simulating charged systems. In depth information about the package can be found in the relevant sources [1, 4, 2, 3].

ESPResSo consists of two components. The simulation engine is written in C and C++ for the sake of computational efficiency. The steering or control level is interfaced to the kernel via an interpreter of the Python scripting languages.

The kernel performs all computationally demanding tasks. Before all, integration of Newton's equations of motion, including calculation of energies and forces. It also takes care of internal organization of data, storing the data about particles, communication between different processors or cells of the cell-system.

The scripting interface (Python) is used to setup the system (particles, boundary onditions, interactions, ...), control the simulation, run analysis, and store and load results. The user has at hand the full rl3sibili5y and functionality of the scripting language. For instance, it is possible to use the SciPy package for analysis and PyPlot for plotting. With a certain overhead in efficiency, it can also be used to reject/accept new configurations in combined MD/MC schemes. In principle, any parameter which is accessible from the scripting level can be changed at any moment of runtime. In this way methods like thermodynamic integration become readily accessible.

Note: This tutorial assumes that you already have a working ESPResSo installation on your system. If this is not the case, please consult the first chapters of the user's guide for installation instructions.

Task 1 *You can check the features, that are compiled in the ESPResSo core by issueing* `print(code_info.features())` *after having imported the* `code_info` *Module:*

```
1 from espressomd import code_info
2 print(code_info.features())
```

[1] If we have charges there is additionally a concept of Bjerrum length, consult Espresso original paper for more details.

4 Overview over a simulatino script

Typically, a simulatino script consists of the following parts

- System setup (box geometry, thermodynamic ensemble, integrator parameters)
- Placeing the particles
- Setup of interactions between particles
- Warmup (bringing the system into a state suitable for measurements)
- Integration loop (propagate the system in teim and record measurements)

In the following sections, it will be shown, how these steps can be taken. Once the basics are covered, we apply them to the simulation of the Lennard-Jones liquid.

4.1 System setup

The functionality of ESPResSo for python is provided via a python module called `espressomd`. At the beginning of the simulation script, it has to be imported.

```
1 import espressomd
2 from espressomd import integrate
```

For convenience we also import the `integrate` submodule.

The next step would be to create an instance of the System class. This instance is used as a handle to the simulation system. It can be used to manipulate the crucial system parameters like the time step and the size of the simulatio box (`time_step`, and `box_l`). At any time, only one instance of the System class can exist.

```
1 system = espressomd.System()
2 system.time_step = time_step
3 system.box_l = [box_l_x, box_l_y, box_l_z]
```

4.2 Chosing the thermodynamic ensemble, thermostat

Simulations can be carried out in different thermodynamic ensembles such as NVE (particle Number, Velocity, Energy) or NVT (particle Number, Velocity, Temperature) as well as NPT-isotropic (particle Number, Pressure, Temperature). The ensemble is maintained by a thermostat. In this tutorail, the NVE-ensemble will be maintained by means of the Langevin thermostat.

In ESPResSo, the teermostat is set as follows:

```
1 system.thermostat.turn_off()
```

This results in an NVE ensemble.

```
1 system.thermostat.set_langevin(kT=1.0, gamma=0.5)
```

Use a Langevin thermostat (NVT ensemble) with temperature set to 1.0 and damping coefficient to 0.5

4.3 Placing and accessing particles

Particles in the simulation can be accessed via the `part`-property of the System class. Individual particles are referred to by an integer id, e.g., `system.part[0]]`. It is also possible to use common python iterators and slicing operations to access several particles at once.

```
# access position of single particle
print system.part[0].pos

# Iterate over particles
for p in system.part:
    print(p.pos)
    print(p.v)

#Obtain all particle positions
cur_pos = system.part[:].pos
```

Particles can be grouped into several types, so that, e.g., a binary fluid can be simulated. Particle types are identified by integer ids, which are set via the particles' `type` attribute. If no type is specified, 0 is implied.

Particles are added to the simulatino as follows

```
system.part.add(id=0, type=0, pos=[x,y,z])
```

Here, `id` and `type` can be ommited, in which case an unused particle id is assigned automatically and type 0 is impied.

Many objects in ESPResSo have a string representation, and thus can be displayed via python's `print` method:

```
print system.part[0]
```

4.4 Setting up non-bonded interactions

Non-bonded interactions act between all particls of a given combination of particle types. In this tutorial, we use the Lennard-Jones non-bonded interaction. The interactaion of two particles of type 0 can be setup as follows:

```
lj1_eps      = 1.0
lj1_sig      = 1.0
lj1_cut      = 1.12246
lj1_shift    = 0.0
lj1_offset   = 0.0
system.non_bonded_inter[0, 0].lennard_jones.set_params(epsilon=lj_eps
  , sigma=lj_sig ,
```

```
cutoff=lj_cut, shift=lj_shift)
```

4.5 Warmup

In many cases, including this tutorial, particles are initially placed randomly in the simulation box. It is therefore possible that particles overlap, i.e., that there is a huge repulsive force between them. In this case, integrating the equations of motion would not be numerically stable. Hence, it is necessary to remove this overlap. This is done by limiting the maximum force between two particles, integrating the equations of motion, and increasing the limit step by step. This is done as follows

```
# Obtain minimum distance between particles
act_min_dist = system.analysis.mindist()
lj_cap=10
while i < warm_n_time and act_min_dist < lj_sigma*0.9 :
    # Set the force cap
    system.non_bonded_inter.set_force_cap(lj_cap)
    lj_cap += 1.0
    # Integrate the equation of motion
    integrate.integrate(100)
    # Obtain minimum distance between particles
    act_min_dist = system.analysis.mindist()

# Disbale force cap
system.non_bonded_inter.set_force_cap(0)
\end{lstlistingg}
In this code fragmetn, you can also see, how the analysis
  routines can be used to obtain information about the
  simulation system, and how to integrate the equation of
  motion.

\section{Putting it all together: Lennard-Jones Liquid
  Simulation}

After we have briefly explained the use of \es{}, we now come
  to the
Lennard-Jones Liquid Simulation. Before we explain the script
  step by step, run the
\texttt{lj\_tutorial.py} with \texttt{pypresso} to get all
  generated files.

\subsection{Iniialization}
First, we include necessary modules with \lstinline|import|.
{\small\vspace{0,2cm}
```

```
\begin{lstlisting}
import espressomd
from espressomd import integrate

from __future__ import print_function
import cPickle as pickle
import os
import numpy as np

print("""
=======================================================
=                    lj_tutorial.py                   =
=======================================================

Program Information:""")
print(code_info.features())
```

4.5.1 System Setup

At first, we must configure the environment and set the needed parameters. It is good practice to define all simulation parameters as variables in a single location.

```
# System parameters
############################################################
#Number of particles and density
n_part  = 108
density = 0.8442

skin        = 0.1
time_step   = 0.001
eq_tstep    = 0.0001
temperature = 0.728

box_l       = np.power(n_part/density, 1.0/3.0) + 2*skin

warm_steps  = 100
warm_n_time = 2000
min_dist    = 0.87

# integration
sampling_interval       = 10
equilibration_interval  = 1000

sampling_iterations     = 10000
equilibration_iterations= 20

# Interaction parameters (Lennard Jones)
```

```
26 ############################################################
27
28 lj_eps = 1.0
29 lj_sig = 1.0
30 lj_cut = 2.5
31 lj_cap = 20
32
33 # System setup
34 ############################################################
35 system                = espressomd.System()
36
37 if not os.path.exists('data') :
38     os.mkdir('data')
39
40 system.time_step      = time_step
41 system.skin           = skin
42
43 system.box_l = [box_l, box_l, box_l]
44
45 system.non_bonded_inter[0, 0].lennard_jones.set_params(
46     epsilon=lj_eps, sigma=lj_sig,
47     cutoff=lj_cut, shift="auto")
48 system.non_bonded_inter.set_force_cap(lj_cap)
49
50 print("LJ-parameters:")
51 print(system.non_bonded_inter[0, 0].lennard_jones.get_params())
52
53 # Thermostat
54 system.thermostat.set_langevin(kT=temperature, gamma=1.0)
55 \end{lstlsisting}
56
57 \subsection{Particles}
58 \begin{lstlisting}
59 # Particle setup
60 ############################################################
61
62 volume = box_l * box_l * box_l
63
64 for i in range(n_part):
65     system.part.add(id=i, pos=np.random.random(3) * system.box_l)
```

Task 2 *Study the file* `lj_tutorial.py`. *This system mimics the case study 4 of section 4, in the book [5]. How can one define truncated-shifted potential in* `lj_tutorial.py`*? (keep in mind that Espresso has already a factor of 4 at shifted part with cut off* $r_c = 2.5$*)*

$$U(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

$$U(r)^{tr\text{-}sh} = \begin{cases} U(r) - U(r_c) & r_c > r \\ 0 & r_c < r \end{cases}$$

(To find the solution look at line 26. Look at picture 1 to see a plot of the potential)

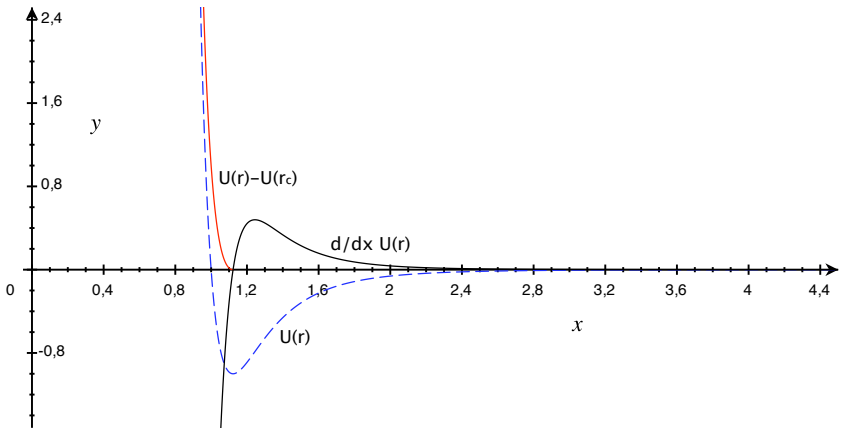


Figure 1: Lennard Jones Potential with $\epsilon = 1$ and radius $\sigma = 1$. If you use a large cutoff such as 2.5σ, the potential is practically zero at the cutoff. The red curve indicates the Weeks-Chandler-Andersen potential, which is obtained from the Lennard-Jones potential by cutting it off in its minimum at $r_c = \sqrt[6]{2}$ and shifting it up.

4.6 Warmup

After we have set the necessary environment we must warmup our system before we run the simulation. We set particles at random positions so some particles can overlap. In this situation ESPResSo will crash with an error: particle out of range. To remove the overlap between particles, we cap forces by setting the Lennard-Jones force constant

below a certain distance. Therefore we use the set_force_cap command. We do the procedure `warm_n_times` times for `warm_steps` steps and stop only if the minimal distance between particles is also larger than `min_dist`, that was set earlier. To turn the capping off, we set `set_force_cap(0)`. Then we equilibrate our system until all relevant physical observables are fluctuating around their mean values. In the case of Lennard Jones it is enough to monitor energy. To equilibrate we use the langevin thermostat set to the target temperature.

```
1  ###########################################################################
2  #  Warmup Integration                                                     #
3  ###########################################################################
4
5  print("""
6  Start warmup integration:
7  At maximum {} times {} steps
8  Stop if minimal distance is larger than {}
9  """.strip().format(warm_n_time, warm_steps, min_dist))
10
11 i = 0
12 act_min_dist = system.analysis.mindist()
13 while i < warm_n_time and act_min_dist < min_dist :
14     integrate.integrate(warm_steps)
15     act_min_dist = system.analysis.mindist()
16     print("run {} at time = {} (LJ cap= {} ) min dist = {}".strip().
         format(i, system.time, lj_cap, act_min_dist))
17     i+=1
18     lj_cap += 1.0
19     system.non_bonded_inter.set_force_cap(lj_cap)
20
21 system.non_bonded_inter.set_force_cap(0)
22
23 print("\nWarm up finished\n")
24
25 system.time_step = eq_tstep
26
27 for i in range(equilibration_iterations):
28     integrate.integrate(equilibration_interval)
29     energies = system.analysis.energy()
30     print("eq run {} at time {}\n".format(i, system.time))
31
32 print("\nEquilibration done\n")
33 # Switch thermostat off NVE msd measurement
34 system.thermostat.turn_off()
```

4.7 Integrating the equaitons of motion, taking measurements

After we have set the necessary environment and warmed up our system, we can now start with the actual simulation. To analyse our data after the simulation has completed,

we open some files for writing the data in.

```
print("\nSampling\n")
system.time_step = time_step

en_fp   = open('data/energy.dat', 'w')
sys_fp  = open('data/sim_info.pickle', 'w')
part_fp = open('data/part.pickle', 'w')
msd_fp  = open('data/msd.dat', 'w')
rdf_fp  = open('data/rdf.dat', 'w')

en_fp.write("#\n#\n#\n# Pressure    Kinetic Potential    Temperature\n
  #\n")

#start positions of all particles for MSD calculation on the fly
start_pos=system.part[:].pos

# save system setup
pickle.dump(system, sys_fp, -1)
pickle.dump(system.part, conf_fp, -1)

msd = np.zeros((sampling_iterations,))

# save start particle configuration
pickle.dump(system.part, part_fp, -1)

for i in range(1, sampling_iterations + 1):
    integrate.integrate(sampling_interval)
    energies = system.analysis.energy()
    pressure = system.analysis.pressure()

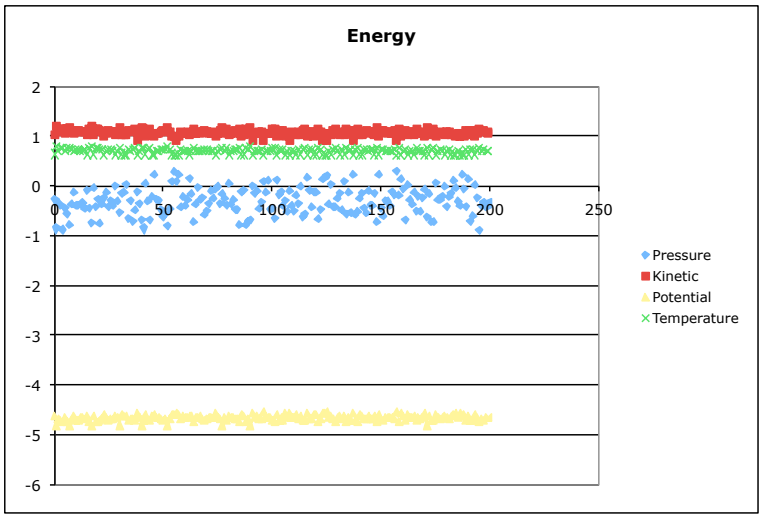
    kinetic_temperature = energies['ideal']/( 1.5 * n_part)

    en_fp.write("%i\t%1.5e\t%1.5e\t%1.5e\t%1.5e\t%1.5e\n" % (i,
      pressure['total'], energies['total'], energies['ideal'],
      energies['total'] - energies['ideal'], kinetic_temperature))
```

In the energy.dat file we print out the values for pressure, kinetic and potential energies, temperature obtained with the analysis submodule system.analysis.energy(). See the code in the snippet above, which contains the main sampling loop of the script.
`kinetic temperature` here refers to the measured temperature obtained from kinetic energy and the number of degrees of freedom in the system. It should fluctuate around the preset temperature of the thermostat.

Task 3 *Plot the time evolution of pressure and energy, which are written into the* `data` *directory in the file* `energy.dat`.

The result plot for `energy.dat` should be similar to this one:



As we can see the system is in equilibrium because pressure, potential and kinetic energy per particle and calculated current temperature fluctuate around their mean values.

4.8 Simple Error Estimation on Time Series Data

A simple way to estimate the error of an observable is to use the common standard deviation ($\sqrt{(}\sigma)$) and the standard error of the mean (SE) for N *uncorrelated* samples:

$$\sigma = \langle x^2 - \langle x \rangle^2 \rangle \tag{1}$$

$$SE = \sqrt{(}\sigma)/\sqrt{(}N) \tag{2}$$

```
# Data arrays for simple error estimation
etotal = np.zeros((sampling_iterations,))
ptotal = np.zeros((sampling_iterations,))
for i in range(1, sampling_iterations + 1):
    energies = system.analysis.energy()
    pressure = system.analysis.pressure()

    etotal[i-1] = energies['total']
    ptotal[i-1] = pressure['total']

# calculate the variance of the total energy and total pressure
    using scipys statistic operations
```

```
12 error_total_energy=np.sqrt(etotal.var())/np.sqrt(
     sampling_iterations)
13 error_total_pressure=np.sqrt(ptotal.var())/np.sqrt(
     sampling_iterations)
14
15 en_fp.write("#mean_energy energy_error mean_pressure
     pressure_error\n#%1.5e %1.5e %1.5e %1.5e" % \
```

4.9 Online analysis of correlations

This section will be updated once the correlator functionality is available in the python interface.

4.10 Other Useful Scripts

The radial distribution function (RDF) describes the distribution of particles around the center of a fixed particle, as a function of the particle-particle distance. This of course assumes that the particle distribution is isotropic around the particles.

The rdf is computed on the fly in the current script, the data is then written to `data/rdf.dat`.

In order to compute the rdf with `espressomd` one needs the following code fragments:

```
# analyzing the radial distribution function
# setting the parameters for the rdf
r_bins = 30
r_min  = 0.0
r_max  = system.box_l[0]/2.0

avg_rdf=np.zeros((r_bins,))

# calculate the rdf in the main sampling loop
for i in range(1, sampling_iterations + 1):
    integrate.integrate(sampling_interval)
    r, rdf = system.analysis.rdf(rdf_type="rdf", type_list_a
      =[0], type_list_b=[0], r_min=r_min, r_max=r_max, r_bins=
      r_bins)
    # system.analysis.rdf() return the bin positions as numpy
      array, here 'r'
    # and the corresponding value of the radial distribution
      function also as
    # numpy array in 'rdf'

    avg_rdf+= rdf/sampling_iterations
    # simply averaging over the calculated rdf and saving it to
       the ''average'' rdf
    # 'avg_rdf'


# after the sampling is done write out the radial distribution
  data and then close
# the file
for i in range(r_bins):
    rdf_fp.write("%1.5e %1.5e\n" % (r[i], avg_rdf[i]))
```

```
26
27 rdf_fp.close()
```

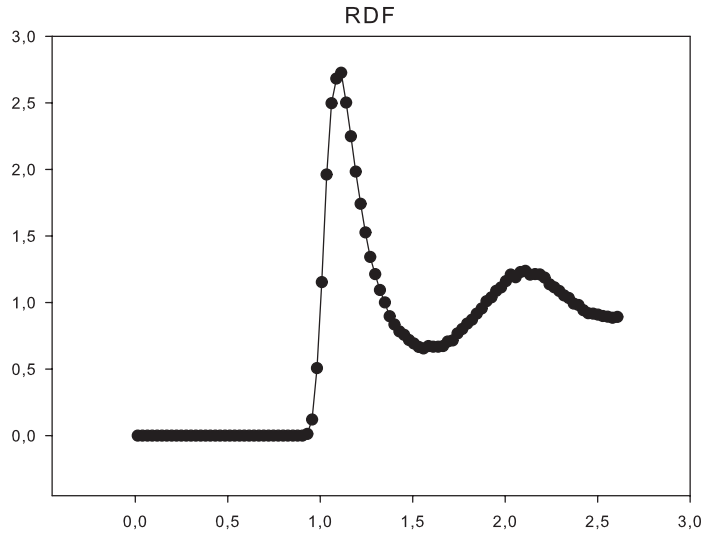


Figure 2: The rdf.dat plot should be similar to this one

5 Binary Lennard Jones liquid

In ESPResSo it is possible to simulate particles of various sizes, meaning different values of the Lennard Jones paramter σ. For this purpose there are different mixing rules that define how particles of different size and different "affinity", Lennard Jones ϵ interact. The most commonly used mixing rules are the Lorentz-Berthelot rules:

$$\sigma_{ij} = \frac{\sigma_{ii} + \sigma_{jj}}{2} \tag{3}$$

$$\epsilon_{ij} = \sqrt{\epsilon_{ii}\epsilon_{jj}}\,. \tag{4}$$

Here σ_{ii} and ϵ_{ii} are the Lennard Jones paramter for the interactions between particles of type i with themselves and σ_{ij} and ϵ_{ij} the interaction parameters between those two species.

Task 4 *Edit the* `lj_liquid.py` *such that it simulates a binary liquid with particles of different sizes (Lennard Jones σs). Beware that every pairwise interaction needs to be declared!*

Notes

- Beware that the system is quite dense already if you choose to use very big particles for the second species, you can also vary the density.
- When setting up the particles be sure about the desired composition of the liquid (a good starting point might be a 50/50 mixture).
- Now you can also look at the radial distrubtion function between particles of different species. Make sure you set up the files and analysis routines.
- Of course you can also vary the ϵ parameter between both species.
- Note also, that there are many differnt mixing rules for the Lennard Jones paramters, you can also choose your own.

References

[1] http://espressomd.org/.

[2] HJ Limbach, A. Arnold, and B. Mann. ESPResSo; an extensible simulation package for research on soft matter systems. *Computer Physics Communications*, 174(9):704–727, 2006.

[3] A. Arnold, O. Lenz, S. Kesselheim, R. Weeber, F. Fahrenberger, D. Röhm, P. Košovan, and C. Holm. ESPResSo 3.1 — molecular dynamics software for coarse-grained models. In M. Griebel and M. A. Schweitzer, editors, *Meshfree Methods for Partial Differential Equations VI*, volume 89 of *Lecture Notes in Computational Science and Engineering*, pages 1–23. Springer Berlin Heidelberg, 2013.

[4] A. Arnold, BA Mann, HJ Limbach, and C. Holm. ESPResSo–An Extensible Simulation Package for Research on Soft Matter Systems. *Forschung und wissenschaftliches Rechnen*, 63:43–59, 2003.

[5] Daan Frenkel and Berend Smit. *Understanding Molecular Simulation*. Academic Press, San Diego, second edition, 2002.