Tutorial 1: Lennard-Jones Liquid[*]

ESPResSo Basics

October 10, 2016

Contents

[*]For ESPResSo 3.4-dev-3195-gb4ff7de-dirty

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 produced simulation data.

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 The 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 Python versions

Espresso can be used both, with python 2 and python 3. In these tutorials, we use python 3 print syntax. If you are on python 2, issue the following command to activate it:

```python
from __future__ import print_function
```

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

4 First steps

What is ESPResSo? 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 conditions, interactions, . . .), control the simulation, run analysis, and store and load results. The user has at hand the full reliability 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.

Using the pypresso script in the build directory, python simulation scripts can be run conveniently:

```
./pypresso simulation.py
```

Task 1 *You can check the features, that are compiled in the ESPResSo core by issuing `print(espressomd.features())` after having imported the `espressomd` Module. Features can be switched on or off via the* `myconfig.hpp file. See the chapter on installation in the user's guide.`

```python
import espressomd
print(espressomd.features())
```

5 Overview over a simulation script

Typically, a simulation script consists of the following parts

- System setup (box geometry, thermodynamic ensemble, integrator parameters)
- Placing the particles
- Setup of interactions between particles
- Warm up (bringing the system into a state suitable for measurements)
- Integration loop (propagate the system in time 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. Note that only the core elements of the Lennard-Jones simulation script will be covered in this document. The full script can be found in `scripts/lj_tutorial.py` in the Lennard-Jones tutorial directory.

5.1 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.

```python
import espressomd
```

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 simulation box (`time_step`, and `box_l`). At any time, only one instance of the System class can exist.

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

5.2 Choosing 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 tutorial we use the Langevin thermostat.

In ESPResSo, the thermostat is set as follows:

```python
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. Alternatively, the thermostat can be turned off using

```python
system.thermostat.turn_off()
```

This results in an NVE ensemble.

5.3 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.

```python
# 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 it is not specified, zero is implied.

Particles are added to the simulation as follows

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

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

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

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

5.4 Setting up non-bonded interactions

Non-bonded interactions act between all particles of a given combination of particle types. In this tutorial, we use the Lennard-Jones non-bonded interaction. The interaction 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)
```

5.5 Warmup

In many cases, including this tutorial, particles are initially placed randomly in the simulation box. It is therefore possible that particles overlap, resulting in 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 force 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
    system.integrator.run(100)
    # Obtain minimum distance between particles
    act_min_dist = system.analysis.mindist()

# Disable force cap
system.non_bonded_inter.set_force_cap(0)
```

In this code fragment, 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.

6 Putting it all together: Lennard-Jones liquid simulation

After we have briefly explained the use of ESPResSo, we now come to the Lennard-Jones Liquid Simulation. Before we explain the script step by step, run the `lj_tutorial.py` with `pypresso` to get all generated files.

6.1 Initialization

First, we include necessary modules with **import**.

```
from __future__ import print_function
import espressomd
from espressomd import code_info

import os
import numpy as np

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

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

6.1.1 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
#############################################################
n_part  = 500
density = 0.8442

skin        = 0.1
time_step   = 0.01
eq_tstep    = 0.001
temperature = 0.728

box_l       = np.power(n_part/density, 1.0/3.0)

warm_steps  = 100
warm_n_time = 2000
min_dist    = 0.87

# integration
sampling_interval       = 100
equilibration_interval  = 1000

sampling_iterations     = 100
equilibration_iterations= 5


# Interaction parameters (Lennard Jones)
#############################################################

lj_eps = 1.0
lj_sig = 1.0
lj_cut = 2.5*lj_sig
lj_cap = 5


# System setup
#############################################################
system              = espressomd.System()

if not os.path.exists('data') :
    os.mkdir('data')

system.time_step    = time_step
system.cell_system.skin         = skin

system.box_l = [box_l, box_l, box_l]

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.set_force_cap(lj_cap)

print("LJ-parameters:")
print(system.non_bonded_inter[0, 0].lennard_jones.get_params())

# Thermostat
system.thermostat.set_langevin(kT=temperature, gamma=1.0)
```

6.2 Particles

The particles are initially placed randomly in the simulation box

```
# Particle setup
#############################################################

volume = box_l * box_l * box_l

for i in range(n_part):
    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)

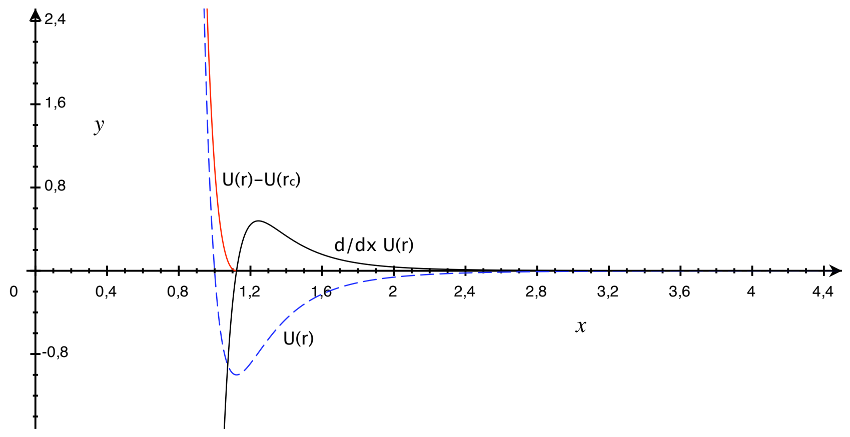

Figure 1: Lennard-Jones Potential with $\epsilon = 1$ and radius $\sigma = 1$. If you use a large cutoff such as 2.5σ, the potential is practically zero at the cutoff. The red curve indicates the Weeks-Chandler-Andersen potential, which is obtained from the Lennard-Jones potential by cutting it off in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

6.3 Removing the overlap between particles (warmup)

This removes the overlap between the randomly placed particles, so the system can be integrated in a stable fashion.

```python
#############################################################
#  Warmup Integration                                       #
#############################################################

print("""
Start warmup integration:
At maximum {} times {} steps
Stop if minimal distance is larger than {}
""".strip().format(warm_n_time, warm_steps, min_dist))

i = 0
act_min_dist = system.analysis.mindist()
while i < warm_n_time and act_min_dist < min_dist :
    system.integrator.run(warm_steps)
    act_min_dist = system.analysis.mindist()
    print("run {} at time = {} (LJ cap= {} ) min dist = {}".
  strip().format(i, system.time, lj_cap, act_min_dist))
    i+=1
    lj_cap += 1.0
    system.non_bonded_inter.set_force_cap(lj_cap)

system.non_bonded_inter.set_force_cap(0)
```

6.4 Integrating the equations of motion, taking measurements

At this point, we have set the necessary environment and warmed up our system. As a last step before starting the actual simulation, we now open the files which we want to output data to during the simulation. Then the simulation is started.

```python
# Record energy versus time
en_fp   = open('data/energy.dat', 'w')

# Record radial distribution function
rdf_fp  = open('data/rdf.dat', 'w')

en_fp.write("#\n#\n#\n# Time\ttotal energy\tkinetic energy\
  tlennard jones energy\ttemperature\n")
```

```python



# Data arrays for simple error estimation
etotal = np.zeros((sampling_iterations,))

# analyzing the radial distribution function
# setting the parameters for the rdf
r_bins = 50
r_min  = 0.0
r_max  = system.box_l[0]/2.0

avg_rdf=np.zeros((r_bins,))
```

In the **energy.dat** file we print out the values for kinetic and potential energies, temperature obtained with the analysis method 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.

Now, the equations of motion are integrated, and measurements are taken. The radial distribution function is averaged over several measurements, to reduce noise.

```python
for i in range(1, sampling_iterations + 1):
    system.integrator.run(sampling_interval)
    energies = system.analysis.energy()

    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)
    avg_rdf+= rdf/sampling_iterations

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

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

Finally, the results are saved.

```python
print("\nMain sampling done\n")

# calculate the variance of the total energy using scipys
  statistic operations
error_total_energy=np.sqrt(etotal.var())/np.sqrt(
  sampling_iterations)

en_fp.write("#mean_energy energy_error %1.5e %1.5e\n" % (etotal.
  mean(), error_total_energy) )

# write out the radial distribution data
for i in range(r_bins):
    rdf_fp.write("%1.5e %1.5e\n" % (r[i], avg_rdf[i]))


en_fp.close()
rdf_fp.close()
```

The radial distribution function, saved here, (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. From the rdf, one can see whether the system is in a liquid or crystal state. For the Lennard-Jones parameters used here, the system is in a liquid state, and particle interactions are significant, as can be seen from the fact that correlations are still visible at a distance of several diameters.

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

Task 3 *Plot the time evolution of energy and temperature, which are written into the* `data` *directory in the file* `energy.dat`. *Make sure that the system is in equilibrium by checking that potential, and kinetic energy and calculated current temperature fluctuate around their mean values and do not show a drift.*

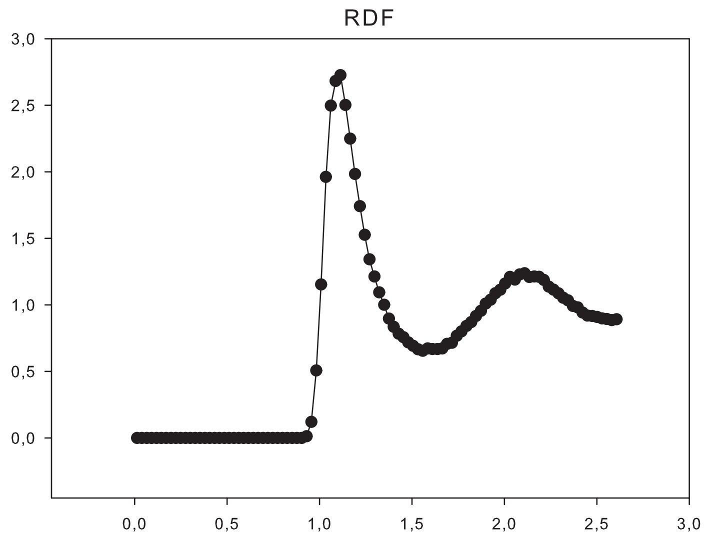

Figure 2: The rdf.dat plot should be similar to this one

Task 4 *Re-run the simulation with a density of 0.1 and a Lennard-Jones cutoff of* $2^{\frac{1}{6}}\sigma$*. Compare the radial distribution function. Now, the correlations at longer distances are gone, as you are simulating a dilute system with purely repulsive particles.*

6.5 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:

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

$$SE = \sqrt{\frac{\sigma}{N}} \tag{2}$$

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

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

# calculate the variance of the total energy and total pressure
  using scipys statistic operations
error_total_energy=np.sqrt(etotal.var())/np.sqrt(
  sampling_iterations)
```

7 Exercise: Binary Lennard-Jones Liquid

A two-component Lennard Jones liquid can be simulated by placing particles of two types (0 and 1) into the system. Depending on the Lennard-Jones parameters, the two components either mix or separate.

Task 5

- *Edit the `lj_tutorial.py` file such that half of the particles are placed with `type=1`. Type 0 is implied for the remaining particles*
- *Specify Lennard-Jones interactions for interactions of type 0 particles with other type 0 particles, of type 1 particles with other type 1 particles, and of type 0 particles with type 1 particles (set parameters for `system.non_bonded_inter[i,j].lennard_jones` where `i,j` can be `0,0`, `1,1`, and `0,1`). Use the same Lennard-Jones parameters for interactions within a component, but use a different `lj_cut_mixed` parameter for the cutoff in the Lennard-Jones interaction between particles of type 0 and particles of type 1. Set this parameter to $2^{\frac{1}{6}}\sigma$ to get de-mixing or to 2.5σ to get mixing between the two components.*
- *Record the radial distribution functions separately for particles of type 0 around particles of type 0, of type 1 around particles of type 1, and finally for particles of type 0 around particles of type 1. This can be done by changing the `type_list` arguments of the `system.analysis.rdf()` command. You can record all three radial distribution functions in a single simulation. It is also possible to write them as several columns into a single file.*
- *Plot the radial distribution functions for all three combinations of particle types. The mixed case will differ significantly, depending on your choice of `lj_cut_mixed`. Explain these differences.*

If you need a hint, look at the `two-component.py` script in your build directory under `doc/tutorials/python/01-lennard_jones/`. Run `two-component-visualization.py` to see a visualization.

8 Exercise: Measuring the particles' mean square displacement

In this task, you will measure the particles' mean square displacement,

$$\mathrm{msd}(t) = \langle (x(t_0+t) - x(t_0))^2 \rangle, \tag{3}$$

using a concept called "observables and correlators". An observable is an object which takes a measurement on the system. It can depend on parameters, such as the ids of the particles to be considered, which are specified, when the observable is instanced.

```python
from espressomd.observables import *
part_pos=ParticlePositions(ids=(0,1,2,3))
```

A correlator is an object which takes results from observables at different times during the simulation, and using a correlation operation, calculates relationships between the observables' values at times t and $t + \delta t$. This can be, e.g., a mean square displacement (expectation value of the square distance a particle has travelled in a certain time) or a velocity autocorrelation (expectation value of the product of the particles' velocities at different time intervals). A correlator for the mean square displacement is instanced as follows

```python
from espressomd.correlators import *

# The correlator works with the part_pos observable. Here, no
  second
# observableis needed

# For short time spans, we record in linear time distances
# for larger time spans, we record in larger and larger steps.
# Use 10 linear measurements 10 time steps apart, each.

# The "square_distance_component_wise" correlation operation
  tells
# the correlator how to calculate a result from the measurements
   of the
# observables at different times

corr=Correlator(obs1=part_pos,
                tau_lin=10,dt=10*time_step,
                tau_max=10000*time_step,
                corr_operation="square_distance_componentwise")
```

Espresso holds a list of correlators to be automatically updated during integration. The correlator is added to this list as follows

```python
system.auto_update_correlators.add(corr)
```

Now the equations of motions can be integrated. This has to be done for a time significantly longer than `tau_max`, to obtain good results. Finally, the results can be obtained as follows

```python
# The 1st column contains the time, the 2nd column the number of
# measurements, and the remaining columns the mean square
  displacement
# for each particle and each Cartesian component
corr.finalize()
result=corr.result()
```

Task 6 *Record the mean square displacement versus time for a Lennard-Jones liquid. Study its dependence on the density and on the friction parameter (γ) of the Langevin thermostat.*

- *Setup up and equilibrate the Lennard Jones liquid as explained in this tutorial*
- *Setup an observable for the particle positions of all particles in the system*
- *Set up a Correlator which records the square distance of the particles for different time intervals. Add it to the list of correlators to be updated automatically.*
- *Integrate the equations of motion*
- *Obtain the results from the correlator, average over all particles and all Cartesian coordinates (column three and above) and write the result to a file*
- *Plot the result*

If you need a hint, look at `msd.py` in your build directory under `doc/tutorials/python/01_lennard-jones/`.

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.