ESPResSo++ Documentation

Release latest

Developer team

May 17, 2021

CONTENTS

Welcome to the homepage of the ESPResSo++ project

ESPResSo++ is an extensible, flexible, fast and parallel simulation software for soft matter research. It is a highly versatile software package for the scientific simulation and analysis of coarse-grained atomistic or bead-spring models as they are used in soft matter research.

ESPResSo and ESPResSo++ have common roots and share parts of the developer/user community. However their development is independent and they are different software packages.

ESPResSo++ is free, open-source software published under the GNU General Public License (GPL).

Please cite this, if you used ESPResSo++ in your research H. V. Guzman, N. Tretyakov, H. Kobayashi, A. C. Fogarty, K. Kreis, J. Krajniak, C. Junghans, K. Kremer, T. Stuehn, "ESPResSo++ 2.0: Advanced methods for multiscale molecular simulation", Computer Physics Communications, 238 (2019), pp. 66-76 DOI: 10.1016/j.cpc.2018.12.017 Online access: https://doi.org/10.1016/j.cpc.2018.12.017

J. D. Halverson, T. Brandes, O. Lenz, A. Arnold, S. Bevc, V. Starchenko, K. Kremer, T. Stuehn, D. Reith, "ESPResSo++: A Modern Multiscale Simulation Package for Soft Matter Systems", Computer Physics Communications, 184 (2013), pp. 1129-1149 DOI: 10.1016/j.cpc.2012.12.004 Online access: http://dx.doi.org/10.1016/j.cpc.2012.12.004

Recent publications where ESPResSo++ was used

CHAPTER

ONE

INSTALLATION

The first step in the installation of ESPResSo++ is to download the latest release from the following location:

https://github.com/espressopp/espressopp/releases

On the command line type:

```
tar -xzf espressopp-latest.tgz
```

This will create a subdirectory espressopp-latest

Enter this subdirectory

```
cd espressopp-latest
```

Create the Makefiles using the cmake command. If you don't have it yet, you have to install it first. It is available for all major Linux distributions and also for Mac OS X. (ubuntu,debian: "apt-get install cmake" or get it from http://www.cmake.org)

```
cmake .
```

(the space and dot after *cmake* are necessary)

If cmake doesn't finish successfully (e.g. it didn't find all the libraries) you can tell cmake manually, where to find them by typing:

```
ccmake .
```

This will open an interactive page where all configuration information can be specified. Alternatively, if cmake . complains on missing BOOST or MPI4PY libraries and you had not installed them, you can try

```
cmake . -DEXTERNAL_BOOST=OFF -DEXTERNAL_MPI4PY=OFF
```

In this case, ESPResSo++ will try to use internal Boost and mpi4py libraries.

After successfully building all the Makefiles you should build ESPResSo++ with:

```
make
```

(This will take several minutes)

Before beeing able to use the espressopp module in Python you need to source the ESPRC file:

```
source ESPRC
```

(This sets all corresponding environment variables to point to the module, e.g. PYTHONPATH) You have to source this file every time you want to work with espressopp. It would advisable to e.g. source the file in your .bashrc file ("source <path_to_espressopp>/ESPRC")

In order to use matplotlib.pyplot for graphical output get the open source code from:

http://sourceforge.net/projects/matplotlib

and follow the installation instructions of your distribution.

CHAPTER

TWO

TUTORIAL

2.1 Basic System Setup

ESPResSo++ is implemented as a python module that has to be imported at the beginning of every script:

```
>>> import espressopp
```

ESPResSo++ uses an object called *System* to store some global variables and is also used to keep the connection between some other important modules. We create it with:

```
>>> system          = espressopp.System()
```

Starting a new simulation with ESPResSo++ we should have an idea about what we want to simulate. E.g. how big should the simulation box be or what is the density of the system or what are the interactions and the interaction ranges between our particles.

Let us start with the size of the simulation box:

```
>>> box             = (10, 10, 10)
```

In many cases you will need a random number generator (e.G. to couple to a temperature bath or to randomly position particles in the simulation box). ESPResSo++ provides its own random number generator (for the experts: see boost/random.hpp) so let's use it:

```
>>> rng             = espressopp.esutil.RNG()
```

Our simulation box needs some boundary conditions. We want to use periodic boundary conditions:

```
>>> bc              = espressopp.bc.OrthorhombicBC(rng, box)
```

We tell our system object about this:

```
>>> system.bc       = bc
>>> system.rng      = rng
```

Now we need to decide which parallelization scheme for the particle storage we want to use. In the current version of ESPResSo++ there is only one storage scheme implemented which is *domain decomposition*. Further parallelized storages (e.g. *atom decomposition* or *force decomposition*) will be implemented in future versions.

The *domain decomposition* storage needs to know how many CPUs (or cores, if there are multicore CPUs) are available for the simulation and how to assign the CPUs to the different domains of our simulation box. Moreover the storage needs to know the maximum interaction range of the particles. In a simple Lennard-Jones fluid this could for example be $r_{cut} = 2^{\frac{1}{6}}$. This value together with the *skin* value determines the minimal size for the so called *linked cells* which are used to speed up Verlet list rebuilds (see Frenkel&Smit or Allen&Tildesley for the details).

```
>>> maxcutoff       = pow(2.0, 1.0/6.0)
>>> skin            = 0.4
```

Tell the system about it:

```
>>> system.skin     = skin
```

In the most simple case, if you want to use only one CPU, the *nodeGrid* and the *cellGrid* could look like this:

```
>>> nodeGrid        = (1,1,1)
>>> cellGrid        = (2,2,2)
```

In general you don't need to take care of that yourself. Just use the corresponding ESPResSo++ routines to calculate a reasonable *nodeGrid* and *cellGrid*:

```
>>> nodeGrid        = espressopp.tools.decomp.nodeGrid(espressopp.MPI.COMM_WORLD.size,
↪box,maxcutoff,skin)
>>> cellGrid        = espressopp.tools.decomp.cellGrid(box, nodeGrid, maxcutoff, skin)
```

Now we have all the ingredients we need for the *domain decomposition* storage of our system:

```
>>> ddstorage       = espressopp.storage.DomainDecomposition(system, nodeGrid,␣
↪cellGrid)
```

We initialized the DomainDecomposition object with a pointer to our system. We also have to inform the system about the DomainDecomposition storage:

```
>>> system.storage = ddstorage
```

The next module we need is the *integrator*. This object will do the actual work of integrating Newtons equations of motion. ESPResSo++ implements the well known *velocity Verlet* algorithm (see for example Frenkel&Smit):

```
>>>  integrator      = espressopp.integrator.VelocityVerlet(system)
```

We have to tell the integrator about the basic time step:

```
>>>  dt              = 0.005
>>>  integrator.dt   = dt
```

Let's do some math in between:

Note: For 3D vectors like positions, velocities or forces ESPResSo++ provides a so called *Real3D* type, which simplifies handling and arithmetic operations with vectors. 3D coordinates would typically be defined like this:

```
>>> a = espressopp.Real3D(2.0, 5.0, 6.0)
>>> b = espressopp.Real3D(0.1, 0.0, 0.5)
```

Now you could do things like:

```
>>> c = a + b          # c is a Real3D object
>>> d = a * 1.5        # d is a Real3D object
>>> e = a - b          # e is a Real3D object
>>> f = e.sqr()        # f is a scalar
>>> g = e.abs()        # g is a scalar
```

In order to make defining vectors even more simple include the line

```
>>> from espressopp import Real3D
```

just at the beginning of your script. This allows to define vectors as:

```
>>> vec = Real3D(2.0, 1.5, 5.0)
```

Back to our simulation:

The most simple simulation we can do is integrating Newtons equation of motion for one particle without any external forces. So let's simply add one particle to the storage of our system. Every particle in ESPResSo++ has a unique particle id and a position (this is obligatory).

```
>>> pid = 1
>>> pos = Real3D(2.0, 4.0, 6.0)     # remember to add "from espressopp import Real3D"
>>>                                  # at the beginning of your script
>>> system.storage.addParticle(pid, pos)
```

Of course nothing will happen when we integrate this. The particle will stay where it is. Add some initial velocity to the particle by adding the follow line to the script:

```
>>> system.storage.modifyParticle(pid, 'v', Real3D(1.0, 0, 0))
```

After particles have been modified make sure that this information is distributed to all CPUs:

```
>>> system.storage.decompose()
```

Now we can propagate the particle by calling the integrator:

```
>>> integrator.run(100)
```

Check the result with:

```
>>> print "The new particle position is: ", system.storage.getParticle(pid).pos
```

Let's add some more particles at random positions with random velocities and random mass and random type 0 or 1. The boundary condition object knows about how to create random positions within the simulation box. We can add all the particles at once by creating a particle list first:

```
>>> particle_list = []
>>> num_particles = 9
>>> for k in range(num_particles):
>>>   pid  = 2 + k
>>>   pos  = system.bc.getRandomPos()
>>>   v    = Real3D(system.rng(), system.rng(), system.rng())
>>>   mass = system.rng()
>>>   type = system.rng(2)
>>>   part = [pid, pos, type, v, mass]
>>>   particle_list.append(part)
>>> system.storage.addParticles(particle_list, 'id', 'pos', 'type', 'v', 'mass')
>>> # don't forget the decomposition
>>> system.storage.decompose()
```

To have a look at the overall system there are several possibilities. The easiest way to get a nice picture is by writing out a PDB file and looking at the configuration with some visualization programm (e.g. VMD):

```
>>> filename = "myconf.pdb"
>>> espressopp.tools.pdb.pdbwrite(filename, system)
```

or (if *vmd* is in your search PATH) you could directly connect to VMD by:

```
>>> espressopp.tools.vmd.connect(system)
```

or you could print all particle information to the screen:

```
>>> for k in range(10):
>>>   p = system.storage.getParticle(k+1)
>>>   print p.id, p.type, p.mass, p.pos, p.v, p.f, p.q
```

2.2 Simple Lennard Jones System

Lets just copy and paste the beginning from the "System Setup" tutorial:

```
>>> import espressopp
>>> from espressopp import Real3D
>>>
>>> system         = espressopp.System()
>>> box            = (10, 10, 10)
>>> rng            = espressopp.esutil.RNG()
>>> bc             = espressopp.bc.OrthorhombicBC(rng, box)
>>> system.bc      = bc
>>> system.rng     = rng
>>> maxcutoff      = pow(2.0, 1.0/6.0)
>>> skin           = 0.4
>>> system.skin    = skin
>>> nodeGrid       = (1,1,1)
>>> cellGrid       = (1,1,1)
>>> nodeGrid       = espressopp.tools.decomp.nodeGrid(espressopp.MPI.COMM_WORLD.size,
↪box,maxcutoff,skin)
>>> cellGrid       = espressopp.tools.decomp.cellGrid(box, nodeGrid, maxcutoff, skin)
>>> ddstorage      = espressopp.storage.DomainDecomposition(system, nodeGrid,␣
↪cellGrid)
>>> system.storage = ddstorage
>>>
>>> integrator     = espressopp.integrator.VelocityVerlet(system)
>>> dt             = 0.005
>>> integrator.dt  = dt
```

And lets add some random particles:

```
>>> num_particles = 20
>>> particle_list = []
>>> for k in range(num_particles):
>>>   pid  = k + 1
>>>   pos  = system.bc.getRandomPos()
>>>   v    = Real3D(0,0,0)
>>>   mass = system.rng()
>>>   type = 0
>>>   part = [pid, pos, type, v, mass]
>>>   particle_list.append(part)
```

(continues on next page)

(continued from previous page)

```
>>> system.storage.addParticles(particle_list, 'id', 'pos', 'type', 'v', 'mass')
>>> system.storage.decompose()
```

All particles should interact via a Lennard Jones potential:

```
>>> LJPot = espressopp.interaction.LennardJones(epsilon=1.0, sigma=1.0,␣
↪cutoff=maxcutoff, shift='auto')
```

shift=True means that the potential will be shifted at the cutoff so that potLJ(cutoff)=0 Next we create a VerletList which will than be used in the interaction: (the Verlet List object needs to know from which system to get its particles and which cutoff to use)

```
>>> verletlist = espressopp.VerletList(system, cutoff=maxcutoff)
```

Now create a non bonded interaction object and add the Lennard Jones potential to that:

```
>>> NonBondedInteraction = espressopp.interaction.VerletListLennardJones(verletlist)
>>> NonBondedInteraction.setPotential(type1=0, type2=0, potential=LJPot)
```

Tell the system about the newly created NonBondedInteraction object:

```
>>> system.addInteraction(NonBondedInteraction)
```

We should set the langevin thermostat in the integrator to cool down the random particle system:

```
>>> langevin             = espressopp.integrator.LangevinThermostat(system)
>>> langevin.gamma       = 1.0
>>> langevin.temperature = 1.0
>>> integrator.addExtension(langevin)
```

and finally let the system run and see how it relaxes or explodes:

```
>>> espressopp.tools.analyse.info(system, integrator)
>>> for k in range(100):
>>>   integrator.run(10)
>>>   espressopp.tools.analyse.info(system, integrator)
```

Due to the random particle positions it may happen, that two or more particles are very close to each other and the resulting repulsive force between them are so high that they 'shoot off' in different directions with very high speed. Usually the numbers are then larger than the computer can deal with. A typical error message you get could look like this:

Note: ERROR: particle 5 has moved to outer space (one or more coordinates are nan)

In order to prevent this, systems that are setup in a random way and thus have strong overlaps between particels have to be "warmed up" before they can be equilibrated.

In ESPResSo++ there are several possible ways of warming up a system. As a first approach one could simply constrain the forces in the integrator. For this purpose ESPResSo++ provides an integrator Extension named CapForces. The two parameters of this Extension are the system and the maximum force that a particle can get. The following python code shows how CapForces can be used. Add it to your Lennard-Jones example just after adding the Langevin Extension:

```
>>> print "starting warmup with force capping ..."
>>> force_capping   = espressopp.integrator.CapForce(system, 1000000.0)
```

(continues on next page)

(continued from previous page)

```
>>> integrator.addExtension(force_capping)
>>> # reduce the time step of the integrator to make the integration numerically more␣
↪stable
>>> integrator.dt = 0.0001
>>> espressopp.tools.analyse.info(system, integrator)
>>> for k in range(10):
>>>   integrator.run(1000)
>>>   espressopp.tools.analyse.info(system, integrator)
```

After the warmup the time step of the integrator can be set to a larger value. The CapForce extension can be disconnected after the warmup to get the original full Lennard-Jones potential back.

```
>>> integrator.dt   = 0.005
>>> integrator.step = 0
>>> force_capping.disconnect()
>>> print "warmup finished - force capping switched off."
```

2.2.1 Task 1:

write a python script that creates a random configuration of 1000 Lennard Jones particles with a number density of 0.85 in a cubic simulation box. Warm up and equilibrate this configuration. Examine the output of the command

```
>>> espressopp.tools.analyse.info(system, integrator)
```

after each integration step. How fast is the energy of the system going down ? How long do you have to warmup ? What are good parameters for dt, force_capping and number of integration steps ?

2.3 Advanced Lennard Jones System

This tutorial needs the matplotlib.pyplot and numpy libraries and also VMD to be installed.

```
>>> import espressopp
```

After importing espressopp we import several other Python packages that we want to use for graphical output of some system parameters (e.g. temperature and energy)

```
>>> import math
>>> import time
>>> import matplotlib
>>> matplotlib.use('TkAgg')
>>> import matplotlib.pyplot as plt
>>> plt.ion()
```

We setup a standard Lennard-Jones system with 1000 particles and a density of 0.85 in a cubic siomulation box. ESPResSo++ provides a “shortcut” to setup such a system:

```
>>> N   = 1000
>>> rho = 0.85
>>> L   = pow(N/rho, 1.0/3)
>>> system, integrator = espressopp.standard_system.LennardJones(N,(L, L, L), dt=0.
↪0001)
```

We also add a Langevin thermostat:

```
>>> langevin = espressopp.integrator.LangevinThermostat(system)
>>> langevin.gamma       = 1.0
>>> langevin.temperature = 1.0
>>> integrator.addExtension(langevin)
```

We do a very short warmup in the beginning to get rid of "extremely" high forces

```
>>> force_capping   = espressopp.integrator.CapForce(system, 1000000.0)
>>> integrator.addExtension(force_capping)
>>> espressopp.tools.analyse.info(system, integrator)
>>> for k in range(10):
>>>   integrator.run(100)
>>>   espressopp.tools.analyse.info(system, integrator)
```

Now let's initialize a graph. So that we can have a realtime-view on what is happening in the simulation:

```
>>> plt.figure()
```

We want to observe temperature and energy of the system:

```
>>> T   = espressopp.analysis.Temperature(system)
>>> E   = espressopp.analysis.EnergyPot(system, per_atom=True)
```

x will be the x-axixs of the graph containg the time. yT and yE will be temperature and energy as y-axes in 2 plots:

```
>>> x   = []
>>> yT  = []
>>> yE  = []
>>> yTmin = 0.0
>>> yEmin = 0.0
>>> x.append(integrator.dt * integrator.step)
>>> yT.append(T.compute())
>>> yE.append(E.compute())
>>> yTmax = max(yT)
>>> yEmax = max(yE)
```

Initialize the two graphs ('ro' means red circles, 'go' means green cirlces, see also pyplot documentation)

```
>>> plt.subplot(211)
>>> gT, = plt.plot(x, yT, 'ro')
>>> plt.subplot(212)
>>> gE, = plt.plot(x, yE, 'go')
```

We also want to observe the configuration with VMD. So we have to connect to vmd. This command will automatically start vmd (vmd has to be found in your PATH environment for this to work)

```
>>> sock = espressopp.tools.vmd.connect(system)
>>> for k in range(200):
>>>   integrator.run(1000)
>>>   espressopp.tools.vmd.imd_positions(system, sock)
```

Update the x-, and y-axes:

```
>>>   x.append(integrator.dt * integrator.step)
>>>   yT.append(T.compute())
>>>   yE.append(E.compute())
>>>   yTmax = max(yT)
>>>   yEmax = max(yE)
```

Plot the temperature graph

```
>>>   plt.subplot(211)
>>>   plt.axis([x[0], x[-1], yTmin, yTmax*1.2 ])
>>>   gT.set_ydata(yT)
>>>   gT.set_xdata(x)
>>>   plt.draw()
```

Plot the energy graph

```
>>>   plt.subplot(212)
>>>   plt.axis([x[0], x[-1], yEmin, yEmax*1.2 ])
>>>   gE.set_ydata(yE)
>>>   gE.set_xdata(x)
>>>   plt.draw()
```

In the end save the equilibrated configurations as .eps and .pdf files

```
>>> plt.savefig('mypyplot.eps')
>>> plt.savefig('mypyplot.pdf')
```

2.4 Polymer Melt

We first import espressopp and then define all the parameters of the simulation:

```
>>> import espressopp
>>> num_chains          = 10
>>> monomers_per_chain  = 10
>>> L                   = 10
>>> box                 = (L, L, L)
>>> bondlen             = 0.97
>>> rc                  = pow(2, 1.0/6.0)
>>> skin                = 0.3
>>> dt                  = 0.005
>>> epsilon             = 1.0
>>> sigma               = 1.0
```

Like in the simple Lennard Jones tutorial we setup the system and the integrator. First the system with the excluded volume interaction (WCA, Lennard Jones type)

```
>>> system         = espressopp.System()
>>> system.rng     = espressopp.esutil.RNG()
>>> system.bc      = espressopp.bc.OrthorhombicBC(system.rng, box)
>>> system.skin    = skin
>>> nodeGrid       = espressopp.tools.decomp.nodeGrid(espressopp.MPI.COMM_WORLD.size,
↪box,rc,skin)
>>> cellGrid       = espressopp.tools.decomp.cellGrid(box, nodeGrid, rc, skin)
>>> system.storage = espressopp.storage.DomainDecomposition(system, nodeGrid,␣
↪cellGrid)
>>> interaction    = espressopp.interaction.VerletListLennardJones(espressopp.
↪VerletList(system, cutoff=rc))
>>> potLJ          = espressopp.interaction.LennardJones(epsilon, sigma, rc)
>>> interaction.setPotential(type1=0, type2=0, potential=potLJ)
>>> system.addInteraction(interaction)
```

Then the integrator with the Langevin extension

```
>>> integrator      = espressopp.integrator.VelocityVerlet(system)
>>> integrator.dt   = dt
>>> thermostat      = espressopp.integrator.LangevinThermostat(system)
>>> thermostat.gamma  = 1.0
>>> thermostat.temperature = temperature
>>> integrator.addExtension(thermostat)
```

Know we add the particles. Keep in mind that we want to create a polymer melt. This means that particles are "bonded" in chains. We setup each polymer chain as a random walk.

```
>>> props    = ['id', 'type', 'mass', 'pos', 'v']
>>> vel_zero = espressopp.Real3D(0.0, 0.0, 0.0)
```

In providing bonding information for the particles we "setup" the bonded chains. For this we use the FixedPairList object that needs to know where and in which storage the particles can be found:

```
>>> bondlist = espressopp.FixedPairList(system.storage)
>>> pid      = 1
>>> type     = 0
>>> mass     = 1.0
>>> chain    = []
```

ESPResSo++ provides a function that will return position and bond information of a random walk. You have to provide a start ID (particle id) and a starting position which we will get from the random position generator of the boundary condition object:

```
>>> for i in range(num_chains):
>>>   startpos = system.bc.getRandomPos()
>>>   positions, bonds = espressopp.tools.topology.polymerRW(pid, startpos, monomers_
↪per_chain, bondlen)
>>>   for k in range(monomers_per_chain):
>>>     part = [pid + k, type, mass, positions[k], vel_zero]
>>>     chain.append(part)
>>>   pid += monomers_per_chain
>>>   type += 1
>>>   system.storage.addParticles(chain, *props)
>>>   system.storage.decompose()
>>>   chain = []
>>>   bondlist.addBonds(bonds)
```

Note: try out the command

```
>>> espressopp.tools.topology.polymerRW(pid, startpos, monomers_per_chain, bondlen)
```

to see what it returns

Don't forget to distribute the particles and the bondlist to the CPUs in the end:

```
>>> system.storage.decompose()
```

Finally add the information about the bonding potential. In this example we are using a FENE-potential between the bonded particles.

```
>>> potFENE    = espressopp.interaction.FENE(K=30.0, r0=0.0, rMax=1.5)
>>> interFENE = espressopp.interaction.FixedPairListFENE(system, bondlist, potFENE)
>>> system.addInteraction(interFENE)
```

Start the integrator and observe how the system explodes. Like in the random Lennard Jones system, we have the same problem here: particles can strongly overlap and thus will get very high forces accelerating them to infinite (for the computer) speed.

```
>>> espressopp.tools.analyse.info(system, integrator)
>>> for k in range(nsteps):
>>>   integrator.run(isteps)
>>>   espressopp.tools.analyse.info(system, integrator)
>>>   espressopp.tools.analyse.info(system, integrator)
```

2.4.1 Task 2:

Try to warmup and equilibrate a dense polymer melt (density=0.85) by using the warmup methods that you have learned in the Lennard Jones tutorial.

2.4.2 Hint:

During warmup you can slowly switch on the excluded volume interaction by starting with a small epsilon and increasing it during integration: You can do this by continuously overwriting the interaction potential after some time interval.

```
>>> potLJ          = espressopp.interaction.LennardJones(new_epsilon, sigma, rc)
>>> interaction.setPotential(type1=0, type2=0, potential=potLJ)
```

2.5 AddNewPotential

The aim of the tutorial is to implement a new interaction potential in ESPResSo++. We start with the Gromos fourth-power bond-stretching potential, because its functional form is simple and its implementation is somewhat similar to other potentials already implemented in ESPResSo++. Everything you learn in this tutorial will then be relevent for implementing any other more complicated potential.

Make sure you have a working, compiled version of ESPResSo++ before starting the tutorial.

For those who are not so familiar with C++ or interfacing python and C++, you will find some helpful notes in the appendix.

2.5.1 Steps for adding a new interaction potential

1. Choose the potential and derive the force.
2. Choose the appropriate interaction template from those in `$ESPRESSOHOME/src/interaction`, e.g. `VerletListInteractionTemplate.hpp`, `FixedTripleListInteractionTemplate.hpp`
3. Create the .cpp, .hpp and .py files for your potential, place them in `$ESPRESSOHOME/src/interaction` and modify `$ESPRESSOHOME/src/interaction/bindings.cpp` and `$ESPRESSOHOME/src/interaction/__init__.py`
4. Compile.

These steps are described in more detail below for our tutorial example potential.

2.5.2 Today's tutorial exercise

Step 1

The potential we are implementing today is a two-body bonded potential with the form

$$V(r_{ij}) = \frac{1}{4}k_{ij}(r_{ij}^2 - r_{0,ij}^2)^2$$

noindent where r_{ij} is the distance between particles *i* and *j*. The potential has two input parameters r_0 and k.

Derive the force.

Step 2

This is a 2-body interaction between a predefined (fixed) list of atom pairs. What is the appropriate interaction template to use? Choose one in `$ESPRESSOHOME/src/interaction`

Open the interaction template file. (When you close the file later, close it without saving, or else later on your compile time will be very long, because of the number of dependencies on the interaction template!) Identify the functions `addForces()` and `computeEnergy()`. Many interaction templates also contain functions such as `computeVirial()`, `computeVirialX()` (for calculating the virial in slabs along the x-direction) etc.

Find the function calls:

```
potential->_computeForce(force, dist)
```

in `addForces()` and

```
potential->_computeEnergy(r21)
```

in `computeEnergy()`.

An interaction template can be combined with many different potentials (e.g. harmonic potential, Lennard Jones potential, etc.) Each potential will have its own C++ class containing functions to compute the energy and forces for that particular potential (see e.g. Harmonic.cpp/hpp, LennardJones.cpp/hpp) In turn, each potential can be combined with many different interaction templates.

You don't need to modify the interaction template file today. (Close it without saving!)

Step 3

In this step we create the .cpp, .hpp and .py files for our potential. Let's call the potential FourthPower. The FourthPower.py file will contain the end-user python interface, and in the FourthPower.cpp and FourthPower.hpp files we will create a C++ class for our potential. We will also write a wrapper which will allow the user to call the C++ code from the python interface.

3(a) Interfacing potential class and interaction template

In many cases, it's not necessary to understand the contents of this section in order to implement a new potential. If you like, you can skip directly to Section *3(b) Creating the new potential class*.

Now we need to understand how the interaction template will interface with our new class. This is done via a class template, e.g. in `Potential.hpp`, `AnglePotential.hpp`, `DihedralPotential.hpp` etc.

Still in `$ESPRESSOHOME/src/interaction`, open the file `Potential.hpp`. (When you close the file later, close it without saving, or else later on your compile time will be very long, because of the number of dependencies on the file!)

Find the functions `_computeForce(Real3D& force, const Real3D& dist)` and `_computeEnergy(real dist)` which you identified in the interaction template. Note that `_computeForce(Real3D& force, const Real3D& dist)` calls the function `_computeForceRaw(force, dist, distSqr)` and `_computeEnergy(real dist)` calls `_computeEnergySqr(dist*dist)` which calls `_computeEnergySqrRaw(distSqr)`. The functions `_computeForceRaw()` and `_computeEnergySqrRaw()` are the new functions we need to write for our new potential. They will be member methods of our new C++ class FourthPower.

You don't need to modify anything in `Potential.hpp` today. (Close it without saving!)

3(b) Creating the new potential class

An easy way to implement the new C++ class is to identify a previously implemented potential which somewhat resembles your new potential, e.g. here we could take the Harmonic potential, which is also a 2-body potential, and which has also been interfaced with the FixedPairListInteractionTemplate.

Still in `$ESPRESSOHOME/src/interaction`, copy the files `Harmonic.py`, `Harmonic.cpp` and `Harmonic.hpp` to new files `FourthPower.py`, `FourthPower.cpp` and `FourthPower.hpp`. In the new files, find and replace all occurences of 'Harmonic' with 'FourthPower', and 'HARMONIC' with 'FOURTHPOWER'.

First modify `FourthPower.hpp`.

Note the `#include` statement for `FixedPairListInteractionTemplate.hpp` and `Potential.hpp`, the files you examined in *Step 2* and Step *3(a) Interfacing potential class and interaction template*.

The Harmonic potential had parameters called `K` and `r0`. You can reuse these for the FourthPower potential, along with the setters and getters `setK`, `getK`, `setR0` and `getR0`. For better efficiency, you could also create a new variable which contains the square of `r0`.

Now we need functions `_computeForceRaw()` and `_computeEnergySqrRaw()`, as explained in Step *3(a) Interfacing potential class and interaction template*. Modify these functions to use the functional form of the fourth power potential as derived in *Step 1*. Note that `Real3D dist`, which contains the vector between the two particles, has been defined as $r_{p1} - r_{p2}$ (see `addForces()` in `FixedPairListInteractionTemplate.hpp`).

Next open `Harmonic.py` and `FourthPower.py`.

Here is an example of an end-user's python script to add an interaction using the harmonic potential:

```
harmonicbondslist = espresso.FixedPairList(system.storage)
harmonicbondslist.addBonds(bond_list) #bond_list is a list of tuples [(particleindex_
↪i,particleindex_j),...]
harmonic_potential = espresso.interaction.Harmonic(K=10.0, r0=1.0, cutoff = 5.0,␣
↪shift = 0.0)
harmonic_interaction = espresso.interaction.FixedPairListHarmonic(system,␣
↪harmonicbondslist, potential=harmonic_potential)
system.addInteraction(harmonic_interaction)
```

Compare this to the contents of `Harmonic.py` to understand the python source code.

Our new potential FourthPower can be called by the end-user in a similar way. Since the Harmonic and FourthPower potentials have similar input parameters (`K`, `r0`) and both use the FixedPairListInteractionTemplate, you don't need to make any further modifications to the file `FourthPower.py`, besides replacing 'Harmonic' with 'FourthPower'.

Next open `FourthPower.cpp`.

Here you will find the C++/python interface, in the function `registerPython()`. If you want to understand this function, you will find details in *Exposing a C++ class or struct to python using boost*. You don't need to make any further modifications to this file, besides replacing 'Harmonic' with 'FourthPower'.

3(c) Including the new class in espressopp

Finally, update the files `$ESPRESSOHOME/src/interaction/bindings.cpp` and `$ESPRESSOHOME/src/interaction/__init__.py` (for example by copying and modifying all the lines referring to the Harmonic potential so that they now refer to the FourthPower potential). You need to make three modifications: to include the new .hpp file, to call the new registerPython() wrapper, and to import everything in the new python module.

Step 4

Move to the directory `$ESPRESSOHOME`. Update the makefiles and compile using the commands:

```
cmake .
make
```

2.5.3 Advanced exercise

For an interaction potential of your choosing, follow the steps above to implement it, e.g. a non-bonded two-body interaction, probably using `VerletListInteractionTemplate` and based on the `LennardJones` potential, or a bonded three-body interaction, probably using `FixedTripleListInteractionTemplate.hpp` and based on the `AngularHarmonic` potential.

You will probably have to write setters and getters for the parameters in your potential in your .hpp file, and make the corresponding modifications to the function `registerPython()` in the .cpp file and the python user interface in the .py file.

2.6 Appendices

2.6.1 Exposing a C++ class or struct to python using boost

(See http://www.boost.org/doc/libs/1_56_0/libs/python/doc/tutorial/doc/html/python/exposing.html)

Say we have a C++ struct called World:

```
struct World
{
    World(std::string msg): msg(msg) {}              // constructor
    void set(std::string msg) { this->msg = msg; }   // function set
    std::string greet() { return msg; }              // function greet
    std::string msg;                                 // member variable
};
```

Now we write the C++ class wrapper for struct World to expose the constructor and the functions greet and set to python:

```
{
    class_<World>("World", init<std::string>())
        .def("greet", &World::greet)
        .def("set", &World::set)
    ;
}
```

If there are additional constructors we can also expose them using `def()`, e.g. for an additional constructor which takes two doubles:

```
class_<World>("World", init<std::string>())
    .def(init<double, double>())
    .def("greet", &World::greet)
    .def("set", &World::set)
;
```

We can also expose the data members of the C++ class or struct and the associated access (getter and setter) functions using `add_property()`, e.g. for the variable myValue with access functions `getMyValue` and `setMyValue`:

```
.add_property("myValue",&World::getMyValue,&World::setMyValue)
```

C++ classes and structs may be derived from other classes. Say we have the C++ struct myDerivedStruct which is derived from the struct myBaseStruct:

```
struct myBaseStruct { virtual ~myBaseStruct(); };
struct myDerivedStruct : myBaseStruct {};
```

We can wrap the base class myBaseStruct as explained above:

```
<Base>("Base")
   /*...*/
   ;
```

Now when we want to wrap the class myDerivedStruct, we tell boost that it is derived from the base class myBaseStruct:

```
class_<myDerivedStruct, bases<myBaseStruct> >("myDerivedStruct")
    /*...*/
    ;
```

2.6.2 C++ templates

See http://www.cplusplus.com/doc/oldtutorial/templates/

2.6.3 typedef

typedef declaration allows you to create an alias that can be used anywhere in place of a (possibly complex) type name

```
typedef DataType AliasName;
```

2.6.4 Python notes

Syntax for classes in python

(See also https://docs.python.org/2/tutorial/classes.html)

Here is a python class called DerivedClassName which is derived from two other base classes (BaseClassName1 and BaseClassName1), is initialised with two variables x and y which have default values 1 and 2, and contains a function myFunction.

```
class DerivedClassName(BaseClassName1, BaseClassName2):
    """docstring"""          #a way of providing some documentation for the class
    def __init__(self,x=1,y=2): #takes two variable which have default values 1 and 2
        self.x = x
        self.y = y
    def myFunction(self):
        return self.x * self.y
```

PMI

PMI = parallel method invocation. For more details see the file `$ESPRESSOHOME/src/pmi.py`

2.7 Adaptive Resolution Simulations

2.7.1 Theory and Background

ESPResSo++ provides functionality to run adaptive resolution simulations using the Adaptive Resolution Simulation Scheme (AdResS). In AdResS molecules in different regions in a simulation box are described by different non-bonded force fields, typically atomistic (AT) and coarse-grained (CG). These different subregions are interfaced and coupled via a hybrid region, where the interaction smoothly changes. Molecules can diffuse between the different regions and change their interaction on the fly.

There are two different AdResS approaches: The force-based scheme, in which forces are interpolated, as well as the energy-based scheme (Hamiltonian AdResS or H-AdResS) which interpolates on the level of potential energies. In force-based AdResS (see, for example, Praprotnik et al., J. Chem. Phys. 123, 224106 (2005) as well as Annu. Rev. Phys. Chem. 59, 545 (2008)), we have for the net force between the molecules α and β

$$\mathbf{F}_{\alpha|\beta} = \lambda(\mathbf{R}_\alpha)\lambda(\mathbf{R}_\beta)\mathbf{F}^{\mathrm{AT}}_{\alpha|\beta} + (1 - \lambda(\mathbf{R}_\alpha)\lambda(\mathbf{R}_\beta))\,\mathbf{F}^{\mathrm{CG}}_{\alpha|\beta},$$

where $\mathbf{F}^{\mathrm{AT}}_{\alpha|\beta}$ is an AT force field based on the individual atoms belonging to the molecules α and β and λ is a position dependent resolution function smoothly changing from 1 in the AT region to 0 in the CG region via the hybrid buffer

region. It is evaluated based on the molecules' center of mass positions $\mathbf{R}_\alpha$. Note that there can of course also be bonded interactions, but these are typically not interpolated, as they are computationally usually much cheaper to evaluate than the non-bonded forces. For the sake of clarity, we omit them here.

In H-AdResS (see Potestio et al., Phys. Rev. Lett. 110, 108301 (2013)), interpolation is performed directly on potential energies in the Hamiltonian as

$$H = \sum_\alpha \sum_{i \in \alpha} \frac{\mathbf{p}_{\alpha i}^2}{2m_{\alpha i}} + \sum_\alpha \left\{ \lambda(\mathbf{R}_\alpha) V_\alpha^{\mathrm{AT}} + (1 - \lambda(\mathbf{R}_\alpha)) V_\alpha^{\mathrm{CG}} \right\},$$

where the first term corresponds to the kinetic energy and we again omitted intramolecular interactions. The forces obtained from this Hamiltonian are

$$\begin{aligned} \mathbf{F}_{\alpha i} = & \sum_{\beta \neq \alpha} \sum_{j \in \beta} \left\{ \frac{\lambda_\alpha + \lambda_\beta}{2} \mathbf{F}_{\alpha i|\beta j}^{\mathrm{AT}} + \left(1 - \frac{\lambda_\alpha + \lambda_\beta}{2} \right) \mathbf{F}_{\alpha i|\beta}^{\mathrm{CG}} \right\} \\ & - \left[V_\alpha^{\mathrm{AT}} - V_\alpha^{\mathrm{CG}} \right] \nabla_{\alpha i} \lambda_\alpha . \end{aligned}$$

The last term, the so-called drift force, comes from applying the position gradient on the position-dependent resolution function λ. It acts only in the hybrid region and unphysically pushes molecules from one region to the other. Therefore, it needs to be corrected. On the other hand, force-based AdResS, contrary to H-AdResS, does not allow a Hamiltonian formulation at all.

Usually, the force fields used in the different regions of the adaptive simulation setup have significantly different pressures given the same temperature and particle density. This pressure gradient leads, in addition to the drift force in H-AdResS, to particles being pushed across the hybrid region. Eventually, the system would evolve to an equilibrium state with a inhomogeneous density profile across the simulation box. Therefore, correction forces needs to be applied in the hybrid region to counter these effects. In H-AdResS one can use a so-called free energy correction (FEC), which on average cancels the drift force in the hybrid region (see Potestio et al., Phys. Rev. Lett. 110, 108301 (2013)). The FEC corresponds to the free energy difference between the subsystems and can, for instance, be derived from Kirkwood thermodynamic integration. An alternative approach which is typically used to cancel the pressure gradient in force-based AdResS is the so-called thermodynamic force (see Fritsch et al., Phys. Rev. Lett. 108, 170602 (2012)). It is derived by constructing the correction directly from the distorted density profile which is obtained without any correction and then refined iteratively.

2.7.2 ESPResSo++ code

Several measures had to be taken to implement adaptive resolution simulations in ESPResSo++. On top of the normal particles, which serve as the CG particles in AdResS, another layer of extra AT particles is introduced such that one has access to both atomistic and CG particles throughout the whole system. A mapping between the two defines which atoms belong to which CG bead. The resolution function λ is implemented as a particle property of the CG particles that is updated after each integration step based on the new positions. This happens in an extension to the Velocity Verlet integrator. The actual adaptive resolution scheme is then implemented via new interaction templates that define how forces and energies are computed in force-based and energy-based AdResS. These templates use for particle pairs in the atomistic region the actual atoms, this is the AT particles, for the force and energy computation while in the CG region they use the CG particles. In the hybrid region, both are used, as defined in the equations above. The drift term of H-AdResS is implemented similarly. Furthermore, the AdResS integrator extension makes sure that the atomistic particles in the CG region travel along with the CG particles and that similarly the CG particles in the AT region are properly updated according to the new atomistic positions after each integration step. The FEC as well as a module to apply the Thermodynamic Force are implemented as integrator extensions.

In the following, we explain the new features step by step (more details about parameters etc. can be found in the documention of the different classes).

Adress Domain Decomposition

When setting up the storage we have to use an appropriate domain decomposition that accomodates storage and proper interprocessor communication of both AT and CG particles.

```
# (H-)AdResS domain decomposition
system.storage = espressopp.storage.DomainDecompositionAdress(system, nodeGrid,␣
↪cellGrid)
```

Atomistic and Coarse-Grained particles

When adding particles to the storage, we have to define them as atomistic or coarse-grained. This has been implemented as the particle property "adrat". If it is 0, the particle is coarse-grained. If it is 1, it is an atomistic particle.

```
# add particles to system
system.storage.addParticles(allParticles, "id", "pos", "v", "f", "type", "mass",
↪"adrat")
```

When adding the particles as above, it is important that a set of atomistic particles belonging to one CG particle appears in the list of particles `allParticles` always after the corresponding CG particle.

Next, the FixedTupleListAdress defines which atomistic particles belong to which coarse-grained particles.

```
# create FixedTupleList object and add the tuples
ftpl = espressopp.FixedTupleListAdress(system.storage)
ftpl.addTuples(tuples)
system.storage.setFixedTuplesAdress(ftpl)
```

In this example, `tuples` is a list of tuples, where each tuple itself is another short list in which the first element is the CG particle and the other elements are the AT particles belonging to it. Note that in ESPResSo++ the CG particle is positioned always in the center of mass of its atoms.

Having set up the FixedTupleList, we can also set up an AdResS fixed pair list that defines bonds between AT particles within individual molecules. This is done in the following way:

```
# add bonds between AT particles
fpl = espressopp.FixedPairListAdress(system.storage, ftpl)
fpl.addBonds(bonds)
```

where `bonds` is a list of bonds between AT particles within CG molecules. Similarly, triple lists for angles, quadruple lists for dihedrals etc. are set up. Compared to conventional bonds, angles, etc. between different normal CG particles one just adds the suffix `Adress` to the appropriate list object and provides it also with the FixedTupleList (`ftpl` in the example). Note that you can define several different such fixed pair lists and you can, for example, also in AdResS simulations still use the normal `FixedPairList` to define bonds between regular CG particles.

AdResS Verlet List

Next, we construct the AdResS Verlet list object for non-bonded interacting particle pairs:

```
# AdResS Verlet list
vl = espressopp.VerletListAdress(system, cutoff=0.8, adrcut=1.4,
                                 dEx=1.5, dHy=1.0,
                                 adrCenter=[Lx/2, Ly/2, Lz/2], sphereAdr=False)
```

We have to provide the cutoffs of the list as well as the sizes of the atomistic and hybrid regions. The parameter `cutoff` corresponds to the cutoff used for CG particle pairs with both particles being in the CG region, while `adrcut` is the cutoff for all other particle pairs (at least one particle of the pair is in the AT or hybrid region). We want to stress that this pair list is build based on the CG particles' positions. Hence, for the AT and hybrid region one needs in some situations to provide a Verlet list cutoff (adrcut) slightly larger than the actual maximum interaction range of the potential, in order to not lose interactions between some atom pairs. Let us clarify this with an example: Thinking of a pair of water molecules, both coarse-grained into single beads, these CG beads could be farther apart than the interaction cutoff. Two hydrogen atoms pointing towards each other, however, could in fact still be in interaction range. Therefore, an appropriate buffer needs to be provided.

The `sphereAdr` flag decides how to geometrically set up the change in resolution. If it's true, the AT region is a spherical region positioned at `adrCenter` with radius `dEx`. If `sphereAdr` is false, the resolution changes along the x-axis of the system and `dEx` corresponds to half the width of the AT region. `dHy` always is the full width of the hybrid region. Instead of providing a 3D position for `adrCenter` as above, one can also provide a particle ID of a CG particle. In this case, the atomistic region will follow the movement of the particle. This should be only done, however, for force-based AdResS, since it would break the Hamiltonian character of H-AdResS, and also only when using a spherical adaptive geometry. Then, however, it is even possible to provide a list of particle IDs, in which case the AT region corresponds to the overlap of the spherical regions defined by the individual particles provided in the list. It will deform accordingly while these particle move.

Interactions

When adding interactions to the system we have to use the corresponding interaction templates. Here is how to set up a non-bonded interaction in a H-AdResS system:

```
# H-AdResS non-bonded interaction: WCA potential between AT particles
# and tabulated potential between CG particles
interNB = espressopp.interaction.VerletListHadressLennardJones(vl, ftpl)
potWCA  = espressopp.interaction.LennardJones(epsilon=1.0, sigma=1.0, shift='auto',
                                  cutoff=rca)
potCG = espressopp.interaction.Tabulated(itype=3, filename=tabCG, cutoff=rc) # CG
interNB.setPotentialAT(type1=1, type2=1, potential=potWCA) # AT
interNB.setPotentialCG(type1=0, type2=0, potential=potCG) # CG
system.addInteraction(interNB)
```

First, we define the appropriate interaction type, in H-AdResS this is `VerletListHadressLennardJones`. Next we define the actual potentials. Then we associate them with the H-AdResS interaction and add the interaction to the system. For force-based AdResS the only change required would be to use the `VerletListAdressLennardJones` interaction.

Note that the here used interaction, `VerletListHadressLennardJones`, couples only Lennard-Jones-type potentials with tabulated ones. However, there exist more such interaction templates for other potentials and potential combinations.

AdResS Integrator Extension

Finally, we have to set up the AdResS integrator extension:

```
# AdResS integrator extension
adress = espressopp.integrator.Adress(system, verletlist, ftpl, regionupdates = 1)
integrator.addExtension(adress)
```

It takes as arguments the Verlet list and the fixed tuple list. Additionally, for the case of a moving and/or deforming AdResS region based on one or more particles, the parameter `regionupdates` specifies how regularly we want to update the shape of the AdResS region in number of steps. This is to avoid as much as possible of the additional communication required to inform different processors of the change of the AdResS region. The parameter defaults to 1 and is not used at all for static AdResS regions.

Having set up the AdResS extension, we can distribute all particles in the box and place the CG molecules in the centers of mass of the atoms which they belong to. This can be done conveniently via

```
# distribute atoms and CG molecules according to AdResS domain decomposition,
# place CG molecules in the center of mass
espressopp.tools.AdressDecomp(system, integrator)
```

Free Energy Compensation

When using H-AdResS, we probably want to also employ a FEC. This can be done as follows:

```
# set up FEC
fec = espressopp.integrator.FreeEnergyCompensation(system, center=[Lx/2, Ly/2, Lz/2])
fec.addForce(itype=3, filename="table_fec.tab", type=1)
integrator.addExtension(fec)
```

The FEC takes as arguments the system object as well as the center of the AT region. Then we add the actual force, which needs to be provided in a table (first column: resolution λ, second: energy, third: force). `itype` defines which type of interpolation should be used for values between the ones provided in the table. 1 corresponds to linear interpolation, 2 to akima splines, 3 to cubic splines. We suggest to use cubic splines. The FEC is applied on CG particles and distributed among the atoms belonging to the CG particle. `type` specifies the CG particle type for which this correction should be applied. One can, for example, use different FECs for different molecules types.

Thermodynamic Force

When using force-based AdResS, or, alternatively, in addition to the FEC in H-AdResS, we can use the thermodynamic force. It can be set up in the following way, very similar to the FEC before:

```
# set up Thermodynamic Force
thdforce = espressopp.integrator.TDforce(system, verletlist)
thdforce.addForce(itype=3,filename="table_tf.tab",type=1)
integrator.addExtension(thdforce)
```

It works largely as for the FEC with the following differences: The table should not provide resolution values in the first column but actual distance values, this is, the distance from the (closest) AT region center. This allows to extend the application of the thermodynamic force slightly beyond the borders of the hybrid region where the resolution is constant. Furthermore, the Thermodynamic Force needs the verletlist as argument.

It is also possible to define a thermodynamic force, which is suited for an adaptive resolution setup with an AT region that is constructed via the overlap of several spherical regions. In this case, the extension needs more information:

```
# set up Thermodynamic Force
thdforce = espressopp.integrator.TDforce(system, verletlist, startdist = 0.9,
                                         enddist = 2.1, edgeweightmultiplier = 20)
thdforce.addForce(itype=3,filename="table_tf.tab",type=1)
integrator.addExtension(thdforce)
```

It gets three more parameters, `startdist`, `enddist` and `edgeweightmultiplier`. `startdist` explicitely says at which distance from the center of the closest AT region defining particle the thermodynamic force starts to act and `enddist` says where it ends. Hence, these value should correspond to what is actually written in the table. `edgeweightmultiplier` is a parameter that speficies how precisely the thermodynamic force should be applied in the overlap regions of different spheres. For most applications, however, 20 should provide reasonable results (for details, see Kreis et al., J. Chem. Theory Comput. 12, 4067 (2016)). The 3 additional parameters are of course also present with some default values in the basic case, but they are ignored unless we have an AT region that is constructed via the overlap of several spherical regions.

2.7.3 Examples

We have provided several example scripts and setups that are available in the ESPResSo++ source code at `examples/adress`. Most of them are based on published papers.

The reader is strongly encouraged to play around with them and test what happens when the setups are modified. Possible questions to ask are provided at the end of the following subsections, which explain the individual examples in more detail.

Force-AdResS: Tetrahedral Liquid

Subfolder: `fadress_tetraliquid`. This example consists of the system that was used in the initial work introducing the force-based adaptive resolution method (see Praprotnik et al., J. Chem. Phys. 123, 224106 (2005) and Phys. Rev. E 73, 066701 (2006)). A liquid composed of artificial tetrahedral molecules, i.e. each molecule consists of 4 bonded atoms arranged in a tetrahedral geometry, is coupled to a CG model which describes the molecules as individual beads.

Questions: The geometry is set in such a way that the resolution changes along the x-axis of the box. Try changing the setup such that the AT region is of spherical shape. You can also try removing the thermostat. Does the system conserve energy? Also vary the size of the atomistic region and see what happens. Can you also make the system all-atomistic or all-CG? You can also try to compare computational times.

Force-AdResS: A Protein in Water

Subfolder: `fadress_protein`. This system is an aqueous solution of the regulatory protein ubiquitin. The atomistic protein and the atomistic water around it is coupled to a coarse-grained water model, which maps water molecules farther away from the protein to single beads. The CG water interaction was parametrized with iterative Boltzmann inversion (IBI). This system is similar to the setup which was used by Fogarty et al. (J. Chem. Phys. 142, 195101 (2015)) to study the structure and dynamics of a protein hydration shell.

Questions: The setup is significantly more complicated than the previous system. Try to understand the script. You can also have a look into the the actual source code and try to understand, for example, how the gromacs parser works. The example is set up as a fully atomistic simulation by setting the size of the atomistic region to a value larger than the simulation box. Try to change the script such that it is an actual adaptive setup. Do not forget the thermodynamic force! Furthermore, how is the high-resolution region positioned now?

Force-AdResS: Self-Adjusting Adaptive Resolution Simulations

Subfolder: `fadress_selfadjusting`. This setup demonstrates how force-based adaptive resolution simulations with self-adjusting high-resolution regions can be set up (Kreis et al., J. Chem. Theory Comput. 12, 4067 (2016)). The system is a polyalanine-9 molecule in aqueous solution. A spherical AT region is associated with each atom of the peptide such that the overall AT region formed by the overlap of all these spheres elegantly envelops the peptide. The peptide starts in an extended configuration and as it folds, the AT region surrounding it adjusts itself accordingly. At the outside, we use again a coarse-grained IBI single-bead model for the water molecules.

Questions: Can you change the system such that fewer atoms are associated with AT region, for example, only the heavy atoms? Can you change the update frequency of the shape of the AT region?

H-AdResS: Tetrahedral Liquid

Subfolder: `hadress_tetraliquid`. This is the system used by Potestio et al. in the paper that proposed the H-AdResS method (Phys. Rev. Lett. 110, 108301 (2013)). It is again a simple system composed of tetrahedral molecules that change their resolution and become individual beads in the CG region. The interpolation occurs along the x-axis. This example has three subfolders.

The first folder `hadress_tetraliquid_plain` runs a simple H-AdResS simulation without any free energy correction. Hence, the drift force strongly pushes molecules from one region to the other. The script contains analysis routines which measure both a density and a pressure profile along the direction of resolution change while the simulation is running. Gathering enough statistics takes a while, but we have also provided reference profiles which are obtained after a sufficiently long simulation. Have a look at them and try to interpret them.

The second folder `hadress_tetraliquid_FEC` contains the same setup but with a free energy correction. For this, two tables are provided, `table_FEC_Helmholtz.dat` and `table_FEC_Gibbs.dat`. They were derived via Kirkwood thermodynamic integration. The first one is based on the Helmholtz free energy difference per particle between the two subsystem, and the second one corresponds to the Gibbs free energy difference per particle. Two density and pressure profiles obtained while applying these correction are also shown. Try to interpret them.

The third folder `hadress_tetraliquid_KTI` contains a simple implementation of Kirkwood thermodynamic integration (KTI) which could in principle, when run for long enough, be used to derive the FEC. This is not an adaptive resolution simulation. Instead, we tell the AdResS integrator extension that we want to run KTI. Then, the extension does not modify the resolution values associated with the different molecules and we can change them by hand during the simulation. In this way, we can set up a simulation in which we change the resolution of all molecules in the system every few steps and slowly proceed from a complete CG system to an all-atom one. Have a look and try to understand what is going on.

There are many more interesting things you can try out: Are the H-AdResS simulations energy conserving? Add the commented Langevin thermostat and compare. Also vary the timestep. Additionally, you can change the size of the hybrid region. What happens if it becomes smaller or larger? Furthermore, what happens if you change the system from H-AdResS to force-based AdResS?

H-AdResS: Water

Subfolder: `hadress_water`. This is a slightly more advanced H-AdResS system in which an atomistic model is coupled to a coarse-grained one, mapping the three water atoms onto single beads.

Questions: Feel free to play around with the system. You could also try to figure out, how the gromacs parsers sets up the interactions and chooses the right H-AdResS interactions.

2.7.4 Adaptive Resolution Simulations with Multiple Time Stepping

Coarse-grained (CG) potentials are typically significantly softer than atomistic (AT) force fields and the corresponding equations of motions can be solved using a larger time step. This suggests the use of multiple time stepping (MTS) techniques in adaptive resolution simulations, in which both AT and CG potentials are present simultaneously. For simulations in which the CG region is much larger than the AT one, this promises a significant speed-up compared to calculations in which a single short time step is used for the whole system. ESPResSo++ provides a RESPA-based MTS scheme (J. Chem. Phys. 97, 1990 (1992)), in which the CG interactions are integrated on a slow timescale and all AT interactions (bonded and non-bonded) on a faster timescale. The scheme can be used both with force-based AdResS and energy-based H-AdResS.

Note that MTS within AdResS simulations can be interpreted as spatially adaptive MTS, in which the integration time scales of the different forces within and between molecules depend on its positions in the simulation box. Large time steps are used in one domain while short time steps are used in another domain of the box. This is in contrast to usual MTS applications, in which the same multiple time stepping is applied everywhere in the system and the separation is usually just between bonded and non-bonded interactions, but without any spatial dependency.

AdResS with Multiple Time Stepping: Implementation

The AdResS-MTS scheme is implemented in ESPResSo++ by two modifications. On the one hand, a new interaction type (*NonbondedSlow*) and a new integrator that implements the RESPA scheme are provided. The integrator calculates all interactions of type *NonbondedSlow* on the long time scale, while all other interactions are treated on the fast timescale. On the other hand, a set of new interaction templates for adaptive resolution interactions are provided. The fast AT and the slow CG adaptive resolution interactions are now implemented in separate interaction templates that have different types (*Nonbonded* and *NonbondedSlow*), which can be exploited by the MTS integrator. Furthermore, the user can specify whether the Thermodynamic Force and the Free Energy Compensation are applied on the slow or fast time scale. Code examples are below:

```
# set up the atomistic part of a force-based adaptive resolution interaction. This␣
↪interaction template incorporates both a Lennard-Jones
# term and a Reaction Field term for the force to compute both the Van der Waals and␣
↪electrostatic forces during one loop over
# the atomistic- and hybrid-region particle pairs
non_bonded_interaction_at = espressopp.interaction.
↪VerletListAdressATLenJonesReacFieldGen(verletlist, ftpl)
potLJ = espressopp.interaction.LennardJones(epsilon=epsilon, sigma=sigma, shift='auto
↪', cutoff=interaction_cutoff_at)
potQQ = espressopp.interaction.ReactionFieldGeneralized(prefactor=138.935485, kappa=0.
↪0, epsilon1=1.0, epsilon2=80.0, cutoff=interaction_cutoff_at, shift="auto")
non_bonded_interaction_at.setPotential1(type1=1, type2=1, potential=potLJ)
non_bonded_interaction_at.setPotential2(type1=1, type2=1, potential=potQQ)
non_bonded_interaction_at.setPotential2(type1=1, type2=0, potential=potQQ)
non_bonded_interaction_at.setPotential2(type1=0, type2=0, potential=potQQ)
system.addInteraction(non_bonded_interaction_at)
```

```
# set up the coarse-grained part of a force-based adaptive resolution interaction
non_bonded_interaction_cg = espressopp.interaction.
↪VerletListAdressCGTabulated(verletlist, ftpl)
potCG = espressopp.interaction.Tabulated(itype=3, filename="table_ibi.dat",␣
↪cutoff=interaction_cutoff_cg)
non_bonded_interaction_cg.setPotential(type1=typeCG, type2=typeCG, potential=potCG)
system.addInteraction(non_bonded_interaction_cg)
```

```
# set up the RESPA VelocityVerlet Integrator (timestep is the short time step,
# and multistep is an integer multiplier to construct the long time step as the
```

(continues on next page)

(continued from previous page)

```
# product of the short time step with the multiplier)
integrator = espressopp.integrator.VelocityVerletRESPA(system)
integrator.dt = timestep
integrator.multistep = multistep
```

```
# add AdResS extension. It also needs to know about the multiple time stepping␣
↪(multistep parameter)
adress = espressopp.integrator.Adress(system, verletlist, ftpl, multistep=multistep)
integrator.addExtension(adress)
```

```
# add Thermodynamic Force and specify whether the force is applied
# together with the slow (slow=True) or fast (slow=False) forces
thdforce = espressopp.integrator.TDforce(system, verletlist, slow=False)
thdforce.addForce(itype=3, filename="table_tf.xvg", type=typeCG)
integrator.addExtension(thdforce)
```

AdResS with Multiple Time Stepping: Examples

Subfolders within the AdResS examples folder: `multiple_time_stepping_fadress` for a force-based AdResS MTS simulation and `multiple_time_stepping_hadress` for an H-AdResS MTS simulation of liquid water. In both examples, the system is a box of liquid water in which the resolution changes along the x-axis. The atomistic model is the SPC/Fw force field with a Reaction Field approach to treat electrostatics. The force-based AdResS example uses a tabulated iterative Boltzmann inversion-based potential in the CG region, while the H-AdResS example employs a simple truncated Harmonic potential to describe the CG interactions.

Have a look at the examples and modify the different time steps. For example, you can test the effect of different time step configurations on the energy conservation in H-AdResS or you investigate whether it makes a difference to apply the corrections on the slow or fast time scale.

2.7.5 Path Integral-AdResS

The path integral (PI) formalism can be used in molecular simulations to account for the quantum mechanical delocalization of light nuclei. It is frequently used, for example, when modeling hydrogen-rich chemical and biological systems, such as proteins or DNA. In the PI methodology, quantum particles are mapped onto classical ring polymers, which represent delocalized wave functions. This renders the PI approach computationally highly expensive (for a detailed introduction see, for example, *M. E. Tuckerman, Statistical Mechanics: Theory and Molecular Simulation*). However, in practice the quantum mechanical description is often only necessary in a small subregion of the overall simulation.

Recently, a PI-based adaptive resolution scheme was developed that allows to include the PI description only locally and to use efficient classical Newtonian mechanics in the rest of the system (*J. Chem. Phys.* **147**, 244104 (2017) and *J. Chem. Theory Comput.* **12**, 3030 (2016)). In this approach the ring polymers are forced to collapse to classical, point-like particles in the classical region. This is achieved by introducing a position-dependent and adaptively changing particle mass which controls the spring constants between the ring polymer beads. Note that this does not necessarily affect the separate "kinietic" masses which are typically introduced in Path Integral Molecular Dyamics.

The method is based on an overall Hamiltonian description and it is consistent with a bottom-up PI quantization procedure. It allows for the calculation of both quantum statistical as well as approximate quantum dynamical quantities in the quantum subregion using ring polymer or centroid molecular dynamics. The methodology is implemented in the ESPResSo++ package and it also makes use of multiple time stepping. For technical details, please see the original publications.

PI-AdResS Implementation

PI-AdResS is implemented in ESPResSo++ by the addition of further particle properties, this is, a variable mass parameter to control the ring polymers' spring constants and a path integral bead (pib) number indicating which imaginary time slice or Trotter number a particle corresponds to. A system is typically set up in such a way that the physical atoms correspond to the coarse-grained ESPResSo++ particles, while ESPResSo++'s atoms, which are linked to the coarse-grained particles using the fixed tuple list, are the actual beads of the ring polymer. A coarse-grained ESPResSo++ particle then correspond to the ring polymer centroid, which are therefore used for the construction of the Verlet list, and control whether a ring polymer is send to another CPU in parallel simulations.

Furthermore, new interaction templates were implemented to accommodate the calculation of interactions between atoms in a path integral-based manner and a new multiple time stepping integrator was developed (see the user manual for detailed documention of the classes and the example for usage in practice).

```
# path integral-based adaptive resolution interaction that employs a tabulated␣
↪potential for the potential in the
# path integral region and a Lennard Jones potential in the classical region (where␣
↪the ring polymers are collapsed)
non_bonded_interaction = espressopp.interaction.
↪VerletListPIadressTabulatedLJ(verletlist, fixedtuplelist, TrotterNumber, speedup_in_
↪CL_region)
```

```
# path integral-based adaptive resolution multiple time stepping integrator
integrator = espressopp.integrator.PIAdressIntegrator(system, verletlist, timestep_
↪short, multiplier_short_to_medium, multiplier_medium_to_long, nTrotter, realkinmass,
↪ constkinmass, temperature, gamma, centroidThermostat, CMDparameter, PILE,␣
↪PILElambda, CLmassmultiplier, freezeCLrings, KTI)
```

PI-AdRresS Example: Liquid Water

Subfolder: `piadress_water` within the AdResS example folder. This system is a box of liquid water and the resolution changes along the x-axis. In the center of the box, the molecules behave quantum mechanically with extended ring polymers, while elsewhere the ring polymers collapse to pointlike particles, making them behave classically and allowing for an efficient force computation. In the PI region, a tabulated potential is used, which was specifically developed for PI-based simulations, while in the classical region a simple WCA potential is employed. The setup is similar to those used in *J. Chem. Phys.* **147**, 244104 (2017).

Have a look at the example and try to understand and play around with the many available options for the PI-based adaptive resolution setup. Use the user manual and the original publication as reference.

2.8 Thermodynamic integration

2.8.1 Theoretical explanation

Thermodynamic integration (TI) is a method used to calculate the free energy difference between two states A and B. For the theoretical background, see e.g. http://www.alchemistry.org. In this tutorial, we show how to perform TI calculations with ESPResSo++. We calculate the free energy of solvation of methanol in water. The complete python script is available in the ESPResSo+ source code under `examples/thd_integration_solvation`

To do TI, we define states A and B, with potentials U^A and U^B. We then construct a pathway of intermediate states between A and B by defining a parameter λ that takes values between 0 and 1 and writing the system potential U as a

function of λ, U^A and U^B. The free energy difference between the states A and B is then given by

$$\Delta A = \int_0^1 \left\langle \frac{dU(\lambda)}{d\lambda} \right\rangle_\lambda \mathrm{d}\lambda$$

In practise, we discretise λ and perform a series of MD simulations with different λ values between 0 and 1, sampling $\frac{dU(\lambda)}{d\lambda}$ in each simulation.

To calculate the solvation free energy of methanol in water, we use a box of water containing one methanol molecule. We simulate desolvation via two separate TI calculations. (Note that the procedure described here is decoupling, and solute-solute interactions will be treated differently if you're doing annihilation instead of decoupling, see Note 1.)

Step 1: free energy change for switching off the Coulombic interactions

State A: methanol has full non-bonded (Coulomb and Lennard Jones) interactions with the solvent

State B: methanol has only Lennard Jones interactions with the solvent

---

Step 2: free energy change for switching off the Lennard Jones interactions

State A: methanol has only Lennard Jones interactions with the solvent

State B: methanol has no interaction with the solvent

Step 1 can be done using a linear function of λ:

$$U(\lambda_C) = (1 - \lambda_C)U_C^A + U_{unaffected}$$

where U_C^A is the solute-solvent Coulombic interaction in state A. In ESPResSo++ the charges used for state A are the particle charges contained in the particle property `charge`. The charges in state B are zero, so $U_C^B(q)$ does not appear in the expression. (The case where A and B both have non-zero charges is not implemented in ESPResSo++). The term $U_{unaffected}$ is all other parts of the potential that don't change with λ_C including all bonded interactions, any solute-solute Coulombic interactions, solvent-solvent Coulombic interactions and all Lennard-Jones interactions. The parameter λ_C goes from 0 to 1 in Step 1.

Step 2 must be done using a softcore potential because of the singularity in the Lennard-Jones potential at $r_{ij} = 0$.

$$\begin{aligned} U(\lambda_L) &= \sum_{i,j} U_L(r_{ij}, \lambda_L) + U_{unaffected} \\ U_L(r_{ij}, \lambda_L) &= (1 - \lambda_L)U_H^A(r_A) + \lambda_L U_H^B(r_B) \\ r_A &= (\alpha\sigma_A^6\lambda^p + r_{ij}^6)^{1/6} \\ r_B &= (\alpha\sigma_B^6(1 - \lambda)^p + r_{ij}^6)^{1/6} \end{aligned}$$

The terms $U_H^A(r_A)$ and $U_H^B(r_B)$ are the normal Lennard-Jones 12-6 hardcore potentials:

$$U_H^A(r_A) = 4.0\epsilon_A\left(\frac{\sigma_A}{r_A}^{12} - \frac{\sigma_A}{r_A}^{6}\right)$$

The sum $\sum_{i,j} U_L(r_{ij}, \lambda_L)$ is over all solute-solvent interactions. The term $U_{unaffected}$ is all other parts of the potential that don't change with λ_L including any solute-solute Lennard-Jones interactions and solvent-solvent Lennard-Jones interactions, which are treated using standard hardcore Lennard-Jones. (In this particular example of methanol, there are no solute-solute Lennard-Jones interactions). Finally α and p are adjustable parameters of the softcore potential.

The ESPResSo++ C++ code allows for different values of ϵ_A, ϵ_B, σ_A and σ_B for every pair of atomtypes interacting via this potential. In this example, we will set ϵ_B to 0 (we are switching off the Lennard-Jones interaction). The parameter λ_L goes from 0 to 1 in Step 2.

2.8.2 ESPResSo++ code

We must perform many separate simulations, each with a different λ value. It is convenient to define a list of λ values in the python script and use an index to access a different element of the list in each separate simulation. The script for the first simulation contains these lines:

```
# Parameters for Thermodynamic Integration
stateBIndices = [1,2,3,4,5,6] #indices of the methanol atoms
lambdaVectorCoul = [0.00, 0.05, 0.10, 0.15, 0.20, 0.25, 0.30, 0.35, 0.40, 0.45, 0.50,
                    0.55, 0.60, 0.65, 0.70, 0.75, 0.80, 0.85, 0.90, 0.95, 1.00, 1.000,
                    1.000, 1.000, 1.000, 1.000, 1.000, 1.000, 1.000, 1.000, 1.000,
                    1.000, 1.000, 1.000, 1.000, 1.000, 1.000, 1.000, 1.000, 1.000,
                    1.000, 1.000, 1.000, 1.000, 1.000, 1.000, 1.000, 1.000, 1.000,
                    1.000, 1.000, 1.000, 1.000, 1.000, 1.000, 1.000, 1.000, 1.000,
                    1.000, 1.000, 1.000]
lambdaVectorVdwl = [0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
                    0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.025,
                    0.050, 0.075, 0.100, 0.125, 0.150, 0.175, 0.200, 0.225, 0.250,
                    0.275, 0.300, 0.325, 0.350, 0.375, 0.400, 0.425, 0.450, 0.475,
                    0.500, 0.525, 0.550, 0.575, 0.600, 0.625, 0.650, 0.675, 0.700,
                    0.725, 0.750, 0.775, 0.800, 0.825, 0.850, 0.875, 0.900, 0.925,
                    0.950, 0.975, 1.000]
lambdaIndex = 0
lambdaTICoul = lambdaVectorCoul[lambdaIndex]
lambdaTIVdwl = lambdaVectorVdwl[lambdaIndex]
```

The list `lambdaVectorCoul` contains the values of λ_C and the list `lambdaVectorVdwl` contains the values of λ_L. The total number of simulations to do Step 1 and Step 2 will be `len(lambdaVectorCoul)` or `len(lambdaVectorVdwl)`. We must make a copy of the python script for each simulation, changing each time the value of `lambdaIndex`.

Next we set up the Coulombic interactions, assuming we already have created a `system` and a `verletlist`. The electrostatics method used is generalised reaction field.

```
#atTypes - list of all atomtypes (integers) used in the pairs interacting via this␣
↪potential
#epsilon1,epsilon2,kappa - reaction field parameters
#annihilate=False means decoupling is used (see Note 1)
#ftpl - a FixedTupleListAdResS object (see AdResS tutorial)
#for non-AdResS simulations, simply set adress=False, and the parameter ftpl is not␣
↪needed
qq_adres_interaction = gromacs.setCoulombInteractionsTI(system, verletlist, nbCutoff,
                                                        atTypes, epsilon1=1, epsilon2=80,
                                                        kappa=0, lambdaTI=lambdaTICoul,
                                                        pidlist=stateBIndices,
                                                        annihilate=False, adress=True,␣
↪ftpl=ftpl)
```

Now we set up the softcore Lennard Jones interaction.

```
#atomtypeparameters - dictionary of format {atomtype: {'eps': epsilon, 'sig': sigma}}
#                     where atomtype is integer and epsilon and sigma are real
#defaults - dictionary containing a key 'combinationrule' with value 1 if the contents
#           of atomtypeparameters need to be converted from c6,c12 format to
#           epsilon,sigma format; can also be an empty dictionary if no conversion␣
↪needed
#sigmaSC, alphaSC, powerSC - parameters of the softcore potential
alphaSC = 0.5
```

(continues on next page)

(continued from previous page)

```
powerSC = 1.0
epsilonB = 0.0
sigmaSC = 0.3
lj_adres_interaction = gromacs.setLennardJonesInteractionsTI(system, defaults,
                                        atomtypeparameters, verletlist, nbCutoff,
                                        epsilonB=epsilonB, sigmaSC=sigmaSC,␣
↪alphaSC=alphaSC,
                                        powerSC=powerSC, lambdaTI=lambdaTIVdwl,
                                        pidlist=stateBIndices, annihilate=False,
                                        adress=True, ftpl=ftpl)
```

We open an output file. In the first line we write the values of λ_C and λ_L for this simulation.

```
dhdlF = open("dhdl.xvg","a")
dhdlF.write("#(coul-lambda, vdw-lambda) = ("+str(lambdaTICoul)+", "+str(lambdaTIVdwl)+
↪")\n")
```

During the MD run, every x number of MD steps, we return to the python level and calculate the derivatives of the energies with respect to λ.

```
dhdlCoul = qq_adres_interaction.computeEnergyDeriv()
dhdlVdwl = lj_adres_interaction.computeEnergyDeriv()
dhdlF.write(str(time)+" "+str(dhdlCoul)+" "+str(dhdlVdwl)+"\n")
```

After all simulations, we can now average $\frac{dU(\lambda)}{d\lambda}$ for each value of λ_C or λ_L, integrate over λ_C and λ_L, add the values ΔA_C and ΔA_L, and take the negative (because the procedure described here is desolvation and we want the free energy of solvation).

2.8.3 Some notes

1. This example given here uses decoupling (solute-solvent interactions are a function of λ, solute-solute interactions are not affected by changes in λ). In ESPResSo++ it is also possible to do annihilation, where both solute-solvent and solute-solute interactions are a function of λ, by setting `annihilate=True` when creating the non-bonded interactions.
2. The procedure described here is desolvation. To get the free energy of solvation, we take the negative of the value obtained after integration.
3. The example Python code snippets here use the helper functions `gromacs.setLennardJonesInteractionsTI` and `gromacs.setCoulombInteractionsTI` contained in `$ESPRESSOHOME/src/tools/convert/gromacs.py`, but this is not necessary. You can do TI with ESPResSo++ without the Gromacs parser by directly calling `espresso.interaction.LennardJonesSoftcoreTI` and `espresso.interaction.ReactionFieldGeneralizedTI`. See the documentation of these two classes.

CHAPTER

THREE

USER INTERFACE

3.1 analysis

3.1.1 espressopp.analysis.AnalysisBase

Overview

List of classes based on AnalysisBase:

espressopp.analysis.LBOutput

Overview

—

Details

espressopp.analysis.LBOutputScreen

espressopp.analysis.LBOutputVzInTime

espressopp.analysis.LBOutputVzOfX

Details

3.1.2 espressopp.analysis.Observable

Overview

List of classes based on Observable:

Details

3.2 bc

3.2.1 espressopp.bc.BC

Overview

Details

espressopp.bc.OrthorombicBC

espressopp.bc.SlabBC

3.3 check

3.4 esutil

3.5 external

3.6 integrator

3.6.1 espressopp.integrator.LatticeBoltzmann

Overview

Details

3.6.2 espressopp.integrator.LBInit

Overview

Details

espressopp.integrator.LBInitPopUniform

espressopp.integrator.LBInitPopWave

espressopp.integrator.LBInitConstForce

espressopp.integrator.LBInitPeriodicForce

3.7 interaction

3.7.1 Angular

espressopp.interaction.AngularPotential

Overview

Details

espressopp.interaction.AngularCosineSquared

espressopp.interaction.AngularHarmonic

espressopp.interaction.Cosine

3.7.2 Bonded

3.7.3 Charged

3.7.4 Constrained

3.7.5 Dihedral

3.7.6 Manybody

3.7.7 Pair

3.7.8 Tabulated

3.7.9 Wall

3.7.10 Other

3.8 io

3.9 espressopp

3.10 standard_system

3.11 storage

3.12 tools

3.12.1 information and analysis

Overview

—

Details

espressopp.tools.analyse

espressopp.tools.info

espressopp.tools.timers

espressopp.tools.vmd

3.12.2 initializing particles

Overview

—

Details

espressopp.tools.lattice

espressopp.tools.replicate

espressopp.tools.topology

espressopp.tools.velocities

espressopp.tools.warmup

3.13 Logging mechanism

ESPResSo++ uses Loggers

Logging can be switched on in your python script with the following command:

```
>>> logging.getLogger("*name of the logger*").setLevel(logging.*Level*)
```

Level is one of the following:

ERROR	for errors that might still allow the application to continue
WARN	for potentially harmful situations
INFO	informational messages highlighting progress
DEBUG	designates fine-grained informational events

Example:

```
>>> import espressopp
>>> import logging
>>> logging.getLogger("Storage").setLevel(logging.ERROR)
```

To log everything (WARNING: this will produce **lots** of output):

```
>>> logging.getLogger("").setLevel(logging.DEBUG)
```

The following loggers are currently available:

- Configurations
- Observable
- Velocities
- BC

- Logger
- FixedListComm
- FixedPairList
- FixedQuadrupleList
- FixedTripleList
- FixedTupleList
- Langevin
- MDIntegrator
- AngularPotential
- DihedralPotential
- Interaction
- InterpolationAkima
- InterpolationCubic
- InterpolationLinear
- InterpolationTable
- Potential
- CellListAllPairsIterator
- DomainDecomposition.CellGrid
- DomainDecomposition
- DomainDecomposition.NodeGrid
- Storage
- DomainDecompositionAdress
- StorageAdress
- VerletList
- VerletList

CHAPTER

FOUR

CREDITS

4.1 ESPResSo++ Developers

The core of the developer team comes from the Polymer Theory Group of Prof. Kurt Kremer at the Max Planck Institute for Polymer Research in Mainz.

A full list of active and former developers is available at the main website of ESPResSo++.

4.2 FAQ

A short list of frequently asked questions is given here.

4.3 Getting Help

If you have any questions do not hesitate to contact us.