ESPResSo++ Documentation

Release 1.7

Torsten Stuehn

December 03, 2013

CONTENTS

ESPResSo++ is an extensible, flexible, fast and parallel simulation software for soft matter research.

It is the successor of the ESPResSo simulation package. ESPResSo++ is a highly versatile software package for the scientific simulation and analysis of coarse-grained atomistic or bead-spring models as used in soft matter research. It also supports charged systems.

Contents:

CHAPTER

ONE

OVERVIEW

CHAPTER

TWO

THE ESPRESSO++ DEVELOPER TEAM

Current developers:

Torsten Stuehn (Max Planck Institute for Polymer Research, Germany) Vitalii Starchenko (Max Planck Institute for Polymer Research, Germany) Konstantin Koschke (Max Planck Institute for Polymer Research, Germany) Livia Moreira (Max Planck Institute for Polymer Research, Germany) Raffaello Potestio (Max Planck Institute for Polymer Research, Germany) Karsten Kreis (Max Planck Institute for Polymer Research, Germany) Sebastian Fritsch (Max Planck Institute for Polymer Research, Germany) Stas Bevc (National Institute of Chemistry, Slovenia)

Former developers:

Thomas Brandes (Fraunhofer Institute SCAI, Germany) Dirk Reith (Fraunhofer Institute SCAI, Germany) Jonathan Halverson (Brookhaven National Laboratory, USA) Axel Arnold (Institute for Computational Physics, Uni-Stuttgart, Germany) Olaf Lenz (Institute for Computational Physics, Uni-Stuttgart, Germany) Christoph Junghans (Los Alamos National Laboratory, USA) Victor Ruehle (University of Cambridge, UK)

CHAPTER

THREE

FREQUENTLY ASKED QUESTIONS

Do I need to learn Python when using ESPResSo++?

The short answer is "no". Most of the example scripts are self-explanatory and can be adapted for your purposes by simple changes. You can also use ESPResSo++ like other MD simulation software, that is driven by some kind of configuration file.

The long answer is "yes". If you want to take advantage of all features of ESPResSo++ you need some knowledge of how the Python interpreter works.

But don't be afraid of learning Python:

- Python is easy to learn
- The ESPResSo++ example simulation scripts gives you a very fast insight of how Python works.
- Writing programs in Python is much easier than writing programs in C++
- Python programs are easier to read than Tcl or Perl programs.

And here are some arguments why it is worth while:

- There are many Python programs you can use in your applications
- Python gives you a flexible way of running MD simulations with ESPResSo++

Do you support other script languages, e.g. Tcl/Tk?

No. We choose the support only Python as ESPResSo++ scripting language. This enables ESPResSo++ users to read and adapt scripts written by other ESPResSo++ users.

Can Tcl scripts converted to Python automatically?

The recommendation is - don't do it! Instead, a Tcl interpreter can be loaded via Python and given the job to do. That is similar to what Tkinter does; Tkinter is a wrapper to use the Tk toolkit from Python.

Why should I use Python if C++ programs are much faster?

Python is the driver of your simulation which will still run in the ESPResSo++ C++ engine.

Python programs are about 30 to 50 times slower than the same programs written in C++.

That is why we use Python to set up and control simulations while the simulation system itself is written in efficient C++ code.

Can I run ESPResSo++ on parallel machines?

Yes. The parallel version uses MPI and is therefore as portable as MPI is. Typical MD simulations scale rather well.

Do I need to write parallel scripts for parallel machines?

No. The Python scripts are executed only by the first processor which will broadcast the ESPResSo++ commands to the other processors automatically using the PMI interface (Parallel Method Invocation). For you, it will look a serial script. But the particles of the simulation are distributed among the available processors and the commands issued for ESPResSo++ will be executed by each processor.

How efficient is ESPResSo++?

Efficiency is a high priority though less than the extendability of the system. You should expect a good performance but might be that ESPResSo++ is less efficient than other simulation programs that are around.

If you experience that ESPResSo++ is more than 2 times slower than other simulation systems you have found a performance bug.

Do I need the source code distribution or can I use binary version?

Currently, we only provide a source code distribution. This might change in the future. Our major problem to provide binary versions is that there are many different Python versions and the binary versions of the ESPResSo++ and python libraries must not be of mixed versions.

Which build systems are used for ESPResSo++?

Compilation and installation of ESPResSo++ is due to the many shared libraries and loadable modules rather complex and so we use a build system to make our job and maintainability easier.

Currently we support building the system with cmake.

What means extendability?

Each software is in a certain sense extendable by adding some functionality somewhere in the code. But we understand extendability in the following sense:

- You can add functionality without changing existent interfaces. For the object-oriented approach this means in practice: take an available base class and define a new derived class with your needed functionality.

CHAPTER

FOUR

ESPRESSO++ TUTORIAL

4.1 Installation of ESPResSo++

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

https://www.espresso-pp.de/Download/espressopp_1_3_1.tgz

On the command line type:

tar -xzf espressopp_1_3_1.tgz

This will create a subdirectory espressopp-1.3.1

Enter this subdirectory

cd espressopp-1.3.1

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

make (This will take several minutes)

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.

4.2 Basic System Setup

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

```
>>> import espresso
```

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          = espresso.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             = espresso.esutil.RNG()
```

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

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

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

```
>>> ddstorage       = espresso.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     = espresso.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 = espresso.Real3D(2.0, 5.0, 6.0)
>>> b = espresso.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 espresso 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 espresso 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"
>>> espresso.tools.pdb.pdbwrite(filename, system)
```

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

```
>>> espresso.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
```

4.3 Simple Lennard Jones System

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

```
>>> import espresso
>>> from espresso import Real3D
>>>
>>> system         = espresso.System()
>>> box            = (10, 10, 10)
>>> rng            = espresso.esutil.RNG()
>>> bc             = espresso.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       = espresso.tools.decomp.nodeGrid(espresso.MPI.COMM_WORLD.size)
>>> cellGrid       = espresso.tools.decomp.cellGrid(box, nodeGrid, maxcutoff, skin)
>>> ddstorage      = espresso.storage.DomainDecomposition(system, nodeGrid, cellGrid)
>>> system.storage = ddstorage
>>>
>>> integrator     = espresso.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)
>>> system.storage.addParticles(particle_list, 'id', 'pos', 'type', 'v', 'mass')
>>> system.storage.decompose()
```

All particles should interact via a Lennard Jones potential:

```
>>> LJPot = espresso.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 = espresso.VerletList(system, cutoff=maxcutoff)
```

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

```
>>> NonBondedInteraction = espresso.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             = espresso.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:

```
>>> espresso.tools.analyse.info(system, integrator)
>>> for k in range(100):
>>>   integrator.run(10)
>>>   espresso.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   = espresso.integrator.CapForce(system, 1000000.0)
>>> integrator.addExtension(force_capping)
>>> # reduce the time step of the integrator to make the integration numerically more stable
>>> integrator.dt = 0.0001
>>> espresso.tools.analyse.info(system, integrator)
>>> for k in range(10):
>>>   integrator.run(1000)
>>>   espresso.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."
```

4.3.1 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

```
>>> espresso.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 ?

4.4 Advanced Lennard Jones System

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

```
>>> import espresso
```

After importing espresso 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 = espresso.standard_system.LennardJones(N,(L, L, L), dt=0.0001)
```

We also add a Langevin thermostat:

```
>>> langevin = espresso.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   = espresso.integrator.CapForce(system, 1000000.0)
>>> integrator.addExtension(force_capping)
>>> espresso.tools.analyse.info(system, integrator)
>>> for k in range(10):
>>>   integrator.run(100)
>>>   espresso.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   = espresso.analysis.Temperature(system)
>>> E   = espresso.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 = espresso.tools.vmd.connect(system)
>>> for k in range(200):
>>>   integrator.run(1000)
>>>   espresso.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')
```

4.5 Polymer Melt

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

```
>>> import espresso
>>> 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         = espresso.System()
>>> system.rng     = espresso.esutil.RNG()
>>> system.bc      = espresso.bc.OrthorhombicBC(system.rng, box)
>>> system.skin    = skin
```

```
>>> nodeGrid       = espresso.tools.decomp.nodeGrid(espresso.MPI.COMM_WORLD.size)
>>> cellGrid       = espresso.tools.decomp.cellGrid(box, nodeGrid, rc, skin)
>>> system.storage = espresso.storage.DomainDecomposition(system, nodeGrid, cellGrid)
>>> interaction    = espresso.interaction.VerletListLennardJones(espresso.VerletList(system, cutoff=r
>>> potLJ          = espresso.interaction.LennardJones(epsilon, sigma, rc)
>>> interaction.setPotential(type1=0, type2=0, potential=potLJ)
>>> system.addInteraction(interaction)
```

Then the integrator with the Langevin extension

```
>>> integrator     = espresso.integrator.VelocityVerlet(system)
>>> integrator.dt  = dt
>>> thermostat     = espresso.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 = espresso.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 = espresso.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 = espresso.tools.topology.polymerRW(pid, startpos, monomers_per_chain, bondler
>>>   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

```
>>> espresso.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   = espresso.interaction.FENE(K=30.0, r0=0.0, rMax=1.5)
>>> interFENE = espresso.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.

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

4.5.1 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.

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

4.6 Adaptive Resolution Scheme (AdResS)

CHAPTER

FIVE

ESPRESSO++ USER INTERFACE - CLASS DESCRIPTION

5.1 Version - Object

Return version information of espresso module

Example:

```
>>> version     = espresso.Version()
>>> print "Name                    = ", version.name
>>> print "Major version number    = ", version.major
>>> print "Minor version number    = ", version.minor
>>> print "Mercurial(hg) revision = ", version.hgrevision
>>> print "boost version           = ", version.boostversion
>>> print "Patchlevel              = ", version.patchlevel
>>> print "Compilation date        = ", version.date
>>> print "Compilation time        = ", version.time
```

to print a full version info string:

```
>>> print version.info()
```

5.2 PMI - Parallel Method Invocation

PMI allows users to write serial Python scripts that use functions and classes that are executed in parallel.

PMI is intended to be used in data-parallel environments, where several threads run in parallel and can communicate via MPI.

In PMI mode, a single thread of control (a python script that runs on the *controller*, i.e. the MPI root task) can invoke arbitrary functions on all other threads (the *workers*) in parallel via *call()*, *invoke()* and *reduce()*. When the function on the workers return, the control is returned to the controller.

This model is equivalent to the "Fork-Join execution model" used e.g. in OpenMP.

PMI also allows to create parallel instances of object classes via *create()*, i.e. instances that have a corresponding object instance on all workers. *call()*, *invoke()* and *reduce()* can be used to call arbitrary methods of these instances.

to execute arbitrary code on all workers, *exec_()* can be used, and to import python modules to all workers, use 'import_()'.

5.2.1 Main program

On the workers, the main program of a PMI script usually consists of a single call to the function *startWorkerLoop()*. On the workers, this will start an infinite loop on the workers that waits to receive the next PMI call, while it will immediately return on the controller. On the workers, the loop ends only, when one of the commands *finalizeWorkers()* or *stopWorkerLoop()* is issued on the controller. A typical PMI main program looks like this:

```
>>> # compute 2*factorial(42) in parallel
>>> import pmi
>>>
>>> # start the worker loop
>>> # on the controller, this function returns immediately
>>> pmi.startWorkerLoop()
>>>
>>> # Do the parallel computation
>>> pmi.import_('math')
>>> pmi.reduce('lambda a,b: a+b', 'math.factorial', 42)
>>>
>>> # exit all workers
>>> pmi.finalizeWorkers()
```

Instead of using *finalizeWorkers()* at the end of the script, you can call *registerAtExit()* anywhere else, which will cause *finalizeWorkers()* to be called when the python interpreter exits.

Alternatively, it is possible to use PMI in an SPMD-like fashion, where each call to a PMI command on the controller must be accompanied by a corresponding call on the worker. This can be either a simple call to *receive()* that accepts any PMI command, or a call to the identical PMI command. In that case, the arguments of the call to the PMI command on the workers are ignored. In this way, it is possible to write SPMD scripts that profit from the PMI communication patterns.

```
>>> # compute 2*factorial(42) in parallel
>>> import pmi
>>>
>>> pmi.exec_('import math')
>>> pmi.reduce('lambda a,b: a+b', 'math.factorial', 42)
```

To start the worker loop, the command *startWorkerLoop()* can be issued on the workers. To stop the worker loop, *stopWorkerLoop()* can be issued on the controller, which will end the worker loop without exiting the workers.

5.2.2 Controller commands

These commands can be called in the controller script. When any of these commands is issued on a worker during the worker loop, a *UserError* is raised.

- *call()*, *invoke()*, *reduce()* to call functions and methods in parallel
- *create()* to create parallel object instances
- *exec_()* and *import_()* to execute arbitrary python code in parallel and to import classes and functions into the global namespace of pmi.
- *sync()* to make sure that all deleted PMI objects have been deleted.
- *finalizeWorkers()* to stop and exit all workers
- *registerAtExit()* to make sure that finalizeWorkers() is called when python exits on the controller
- *stopWorkerLoop()* to interrupt the worker loop an all workers and to return control to the single workers

5.2.3 Worker commands

These commands can be called on a worker.

- *startWorkerLoop()* to start the worker loop
- *receive()* to receive a single PMI command
- *call()*, *invoke()*, *reduce()*, *create()* and *exec_()* to receive a single corresponding PMI command. Note that these commands will ignore any arguments when called on a worker.

5.2.4 PMI Proxy metaclass

The *Proxy* metaclass can be used to easily generate front-end classes to distributed PMI classes. . . .

5.2.5 Useful constants and variables

The pmi module defines the following useful constants and variables:

- *isController* is True when used on the controller, False otherwise
- *isWorker* = not isController
- *ID* is the rank of the MPI task
- *CONTROLLER* is the rank of the Controller (normally the MPI root)
- *workerStr* is a string describing the thread ('Worker #' or 'Controller')
- *inWorkerLoop* is True, if PMI currently executes the worker loop on the workers.

espresso.pmi.**exec_**(**args*)
Controller command that executes arbitrary python code on all (active) workers.

exec_() allows to execute arbitrary Python code on all workers. It can be used to define classes and functions on all workers. Modules should not be imported via exec_(), instead import_() should be used.

Each element of args should be string that is executed on all workers.

Example:

```
>>> pmi.exec_('import hello')
>>> hw = pmi.create('hello.HelloWorld')
```

espresso.pmi.**import_**(**args*)
Controller command that imports python modules on all (active) workers.

Each element of args should be a module name that is imported to all workers.

Example:

```
>>> pmi.import_('hello')
>>> hw = pmi.create('hello.HelloWorld')
```

espresso.pmi.**create**(*cls=None*, **args*, ***kwds*)
Controller command that creates an object on all workers.

cls describes the (new-style) class that should be instantiated. args are the arguments to the constructor of the class. Only classes that are known to PMI can be used, that is, classes that have been imported to pmi via *exec_()* or *import_()*.

Example:

```
>>> pmi.exec_('import hello')
>>> hw = pmi.create('hello.HelloWorld')
>>> print(hw)
MPI process #0: Hello World!
MPI process #1: Hello World!
...
```

Alternative: Note that in this case the class has to be imported to the calling module *and* via PMI.

```
>>> import hello
>>> pmi.exec_('import hello')
>>> hw = pmi.create(hello.HelloWorld)
>>> print(hw)
MPI process #0: Hello World!
MPI process #1: Hello World!
...
```

espresso.pmi.**call**(**args*, ***kwds*)

Call a function on all workers, returning only the return value on the controller.

function denotes the function that is to be called, args and kwds are the arguments to the function. If kwds contains keys that start with with the prefix '__pmictr_', they are stripped of the prefix and are passed only to the controller. If the function should return any results, it will be locally returned. Only functions that are known to PMI can be used, that is functions that have been imported to pmi via *exec_()* or *import_()*.

Example:

```
>>> pmi.exec_('import hello')
>>> hw = pmi.create('hello.HelloWorld')
>>> pmi.call(hw.hello)
>>> # equivalent:
>>> pmi.call('hello.HelloWorld', hw)
```

Note, that you can use only functions that are know to PMI when *call()* is called, i.e. functions in modules that have been imported via *exec_()*.

espresso.pmi.**invoke**(**args*, ***kwds*)

Invoke a function on all workers, gathering the return values into a list.

function denotes the function that is to be called, args and kwds are the arguments to the function. If kwds contains keys that start with with the prefix '__pmictr_', they are stripped of the prefix and are passed only to the controller.

On the controller, invoke() returns the results of the different workers as a list. On the workers, invoke returns None. Only functions that are known to PMI can be used, that is functions that have been imported to pmi via *exec_()* or *import_()*.

Example:

```
>>> pmi.exec_('import hello')
>>> hw = pmi.create('hello.HelloWorld')
>>> messages = pmi.invoke(hw.hello())
>>> # alternative:
>>> messages = pmi.invoke('hello.HelloWorld.hello', hw)
```

espresso.pmi.**reduce**(**args*, ***kwds*)

Invoke a function on all workers, reducing the return values to a single value.

reduceOp is the (associative) operator that is used to process the return values, function denotes the function that is to be called, args and kwds are the arguments to the function. If kwds contains keys that start with with the prefix '__pmictr_', they are stripped of the prefix and are passed only to the controller.

reduce() reduces the results of the different workers into a single value via the operation reduceOp. reduceOp is assumed to be associative. Both reduceOp and function have to be known to PMI, that is they must have been imported to pmi via *exec_()* or *import_()*.

Example:

```
>>> pmi.exec_('import hello')
>>> pmi.exec_('joinstr=lambda a,b: "\n".join(a,b)')
>>> hw = pmi.create('hello.HelloWorld')
>>> print(pmi.reduce('joinstr', hw.hello()))
>>> # equivalent:
>>> print(
...   pmi.reduce('lambda a,b: "\n".join(a,b)',
...              'hello.HelloWorld.hello', hw)
...              )
```

espresso.pmi.**sync**()
Controller command that deletes the PMI objects on the workers that have already been deleted on the controller.

espresso.pmi.**receive**(*expected=None*)
Worker command that receives and handles the next PMI command.

This function waits to receive and handle a single PMI command. If expected is not None and the received command does not equal expected, raise a *UserError*.

espresso.pmi.**startWorkerLoop**()
Worker command that starts the main worker loop.

This function starts a loop that expects to receive PMI commands until *stopWorkerLoop()* or *finalizeWorkers()* is called on the controller.

espresso.pmi.**finalizeWorkers**()
Controller command that stops and exits all workers.

espresso.pmi.**stopWorkerLoop**(*doExit=False*)
Controller command that stops all workers.

If doExit is set, the workers exit afterwards.

espresso.pmi.**registerAtExit**()
Controller command that registers the function *finalizeWorkers()* via atexit.

class espresso.pmi.**Proxy**(*name*, *bases*, *dict*)
A metaclass to be used to create frontend serial objects.

exception espresso.pmi.**UserError**(*msg*)
Raised when PMI has encountered a user error.

5.3 System - Object

The main purpose of this class is to store pointers to some important other classes and thus make them available to C++. In a way the System class can be viewed as a container for system wide global variables. If you need to run more than one system at the same time you can combine several systems with the help of the Multisystem class.

5.3.1 In detail the System class holds pointers to:

- the *storage* (e.g. DomainDecomposition)
- the boundary conditions *bc* for the system (e.g. OrthorhombicBC)

- a random number generator *rng* which is for example used by a thermostat
- the *skin* which is needed for the Verlet lists and the cell grid
- a list of short range interactions that apply to the system these interactions are added with the *addInteraction()* method of the System

Example (not complete):

```
>>> LJSystem      = espresso.System()
>>> LJSystem.bc   = espresso.bc.OrthorhombicBC(rng, boxsize)
>>> LJSystem.rng
>>> LJSystem.skin = 0.4
>>> LJSystem.addInteraction(interLJ)
```

5.4 BC - Boundary Condition Object

This is the abstract base class for all boundary condition objects. It cannot be used directly. All derived classes implement at least the following methods:

- *getMinimumImageVector(pos1, pos2)*
- *getFoldedPosition(pos, imageBox)*
- *getUnfoldedPosition(pos, imageBox)*
- *getRandomPos()*

pos, *pos1* and *pos2* are particle coordinates (type: (*float*, *float*, *float*)). *imageBox* (type: (*int*, *int*, *int*)) specifies the

5.5 OrthorhombicBC - Object

Like all boundary condition objects, this class implements all the methods of the base class **BC** , which are described in detail in the documentation of the abstract class **BC**.

The OrthorhombicBC class is responsible for the orthorhombic boundary condition. Currently only periodic boundary conditions are supported.

Example:

```
>>> boxsize = (Lx, Ly, Lz)
>>> bc = espresso.bc.OrthorhombicBC(rng, boxsize)
```

5.6 Storage - Storage Object

This is the base class for all storage objects. All derived classes implement at least the following methods:

- *decompose()*

  Send all particles to their corresponding cell/cpu

- *addParticle(pid, pos)*:

  Add a particle to the storage

- *removeParticle(pid)*:

Remove a particle with id number *pid* from the storage.

```
>>> system.storage.removeParticle(4)
```

There is an example in *examples* folder

- *getParticle(pid)*:

  Get a particle object. This can be used to get specific particle information:

  ```
  >>> particle = system.storage.getParticle(15)
  >>> print "Particle ID is       : ", particle.id
  >>> print "Particle position is : ", particle.pos
  ```

  you cannot use this particle object to modify particle data. You have to use the modifyParticle command for that (see below).

- *addAdrParticle(pid, pos, last_pos)*:

  Add an AdResS Particle to the storage

- *setFixedTuplesAdress(fixed_tuple_list)*:

- *addParticles(particle_list, *properties)*:

  This routine adds particles with certain properties to the storage.

  param particleList list of particles (and properties) to be added

  param properties property strings

  Each particle in the list must be itself a list where each entry corresponds to the property specified in properties.

  Example:

  ```
  >>> addParticles([[id, pos, type, ... ], ...], 'id', 'pos', 'type', ...)
  ```

- *modifyParticle(pid, property, value, decompose='yes')*

  This routine allows to modify any properties of an already existing particle.

  Example:

  ```
  >>> modifyParticle(pid, 'pos', Real3D(new_x, new_y, new_z))
  ```

- *removeAllParticles()*:

  This routine removes all particles from the storage.

- 'system':

  The property 'system' returns the System object of the storage.

Examples:

```
>>> s.storage.addParticles([[1, espresso.Real3D(3,3,3)], [2, espresso.Real3D(4,4,4)]],'id','pos')
>>> s.storage.decompose()
>>> s.storage.modifyParticle(15, 'pos', Real3D(new_x, new_y, new_z))
```

5.7 BerendsenBarostat - Berendsen barostat Object

This is the Berendsen barostat implementation according to the original paper [Berendsen84]. If Berendsen barostat is defined (as a property of integrator) then at the each run the system size and the particle coordinates will be scaled

by scaling parameter μ according to the formula:

$$\mu = [1 - \Delta t/\tau(P_0 - P)]^{1/3}$$

where Δt - integration timestep, τ - time parameter (coupling parameter), P_0 - external pressure and P - instantaneous pressure.

Example:

```
>>> berendsenP = espresso.integrator.BerendsenBarostat(system)
>>> berendsenP.tau = 0.1
>>> berendsenP.pressure = 1.0
>>> integrator.addExtension(berendsenP)
```

!IMPORTANT In order to run *npt* simulation one should separately define thermostat as well (e.g. BerendsenThermostat).

Definition:

In order to define the Berendsen barostat

```
>>> berendsenP = espresso.integrator.BerendsenBarostat(system)
```

one should have the System defined.

Properties:

- *berendsenP.tau*

  The property 'tau' defines the time parameter τ.

- *berendsenP.pressure*

  The property 'pressure' defines the external pressure P_0.

Setting the integration property:

```
>>> integrator.addExtension(berendsenP)
```

It will define Berendsen barostat as a property of integrator.

One more example:

```
>>> berendsen_barostat = espresso.integrator.BerendsenBarostat(system)
>>> berendsen_barostat.tau = 10.0
>>> berendsen_barostat.pressure = 3.5
>>> integrator.addExtension(berendsen_barostat)
```

Canceling the barostat:

If one do not need the pressure regulation in system anymore or need to switch the ensamble or whatever :)

```
>>> # define barostat with parameters
>>> berendsen = espresso.integrator.BerendsenBarostat(system)
>>> berendsen.tau = 0.8
>>> berendsen.pressure = 15.0
>>> integrator.addExtension(berendsen)
>>> ...
>>> # some runs
>>> ...
>>> # disconnect Berendsen barostat
```

```
>>> berendsen.disconnect()
>>> # the next runs will not include the system size and particle coordinates scaling
```

Connecting the barostat back after the disconnection

```
>>> berendsen.connect()
```

References:

5.8 BerendsenThermostat - Berendsen thermostat Object

This is the Berendsen thermostat implementation according to the original paper [Berendsen84]. If Berendsen thermostat is defined (as a property of integrator) then at the each run the system size and the particle coordinates will be scaled by scaling parameter λ according to the formula:

$$\lambda = [1 + \Delta t/\tau_T(T_0/T - 1)]^{1/2}$$

where Δt - integration timestep, τ_T - time parameter (coupling parameter), T_0 - external temperature and T - instantaneous temperature.

Example:

```
>>> berendsenT = espresso.integrator.BerendsenThermostat(system)
>>> berendsenT.tau = 1.0
>>> berendsenT.temperature = 1.0
>>> integrator.addExtension(berendsenT)
```

Definition:

In order to define the Berendsen thermostat

```
>>> berendsenT = espresso.integrator.BerendsenThermostat(system)
```

one should have the System defined.

Properties:

- *berendsenT.tau*

  The property 'tau' defines the time parameter τ_T.

- *berendsenT.temperature*

  The property 'temperature' defines the external temperature T_0.

Setting the integration property:

```
>>> integrator.addExtension(berendsenT)
```

It will define Berendsen thermostat as a property of integrator.

One more example:

```
>>> berendsen_thermostat = espresso.integrator.BerendsenThermostat(system)
>>> berendsen_thermostat.tau = 0.1
>>> berendsen_thermostat.temperature = 3.2
>>> integrator.addExtension(berendsen_thermostat)
```

Canceling the thermostat:

```
>>> # define thermostat with parameters
>>> berendsen = espresso.integrator.BerendsenThermostat(system)
>>> berendsen.tau = 2.0
>>> berendsen.temperature = 5.0
>>> integrator.addExtension(berendsen)
>>> ...
>>> # some runs
>>> ...
>>> # disconnect Berendsen thermostat
>>> berendsen.disconnect()
```

Connecting the thermostat back after the disconnection

```
>>> berendsen.connect()
```

5.9 LangevinBarostat - Langevin-Hoover barostat Object

This is the barostat implementation to perform Langevin dynamics in a Hoover style extended system according to the paper [Quigley04]. It includes corrections of Hoover approach which were introduced by Martyna et al [Martyna94]. If LangevinBarostat is defined (as a property of integrator) the integration equations will be modified. The volume of system V is introduced as a dynamical variable:

$$\dot{\boldsymbol{r}}_i = \frac{\boldsymbol{p}_i}{m_i} + \frac{p_\epsilon}{W}\boldsymbol{r}_i$$

$$\dot{\boldsymbol{p}}_i = - \bigtriangledown_{\boldsymbol{r}_i} \Phi - (1 + \frac{n}{N_f})\frac{p_\epsilon}{W}\boldsymbol{p}_i - \gamma \boldsymbol{p}_i + \boldsymbol{R}_i$$

$$\dot{V} = dV p_\epsilon / W$$

$$\dot{p}_\epsilon = nV(X - P_{ext}) + \frac{n}{N_f}\sum_{i=1}^{N}\frac{\boldsymbol{p}_i^2}{m_i} - \gamma_p p_\epsilon + R_p$$

where volume has a fictitious mass W and associated momentum p_ϵ, γ_p - friction coefficient, P_{ext} - external pressure and X - instantaneous pressure without white noise contribution from thermostat, n - dimension, N_f - degrees of freedom (if there are no constrains and N is the number of particles in system $N_f = nN$). R_p - values which are drawn from Gaussian distribution of zero mean and unit variance scaled by

$$\sqrt{\frac{2k_B T W \gamma_p}{\Delta t}}$$

!IMPORTANT Terms $-\gamma \boldsymbol{p}_i + \boldsymbol{R}_i$ correspond to the termostat. They are not included here and will not be calculated if the Langevin Thermostat is not defined.

Example:

```
>>> rng = espresso.esutil.RNG()
>>> langevinP = espresso.integrator.LangevinBarostat(system, rng, desiredTemperature)
>>> langevinP.gammaP = 0.05
>>> langevinP.pressure = 1.0
>>> langevinP.mass = pow(10.0, 4)
>>> integrator.addExtension(langevinP)
```

!IMPORTANT This barostat is supposed to be run in a couple with thermostat in order to simulate the *npt* ensamble, because the term R_p needs the temperature as a parameter.

Definition:

In order to define the Langevin-Hoover barostat

```
>>> langevinP = espresso.integrator.LangevinBarostat(system, rng, desiredTemperature)
```

one should have the System and RNG defined and know the desired temperature.

Properties:

- *langevinP.gammaP*

  The property 'gammaP' defines the friction coefficient γ_p.

- *langevinP.pressure*

  The property 'pressure' defines the external pressure P_{ext}.

- *langevinP.mass*

  The property 'mass' defines the fictitious mass W.

Methods:

- *setMassByFrequency(frequency)*

  Set the proper *langevinP.mass* using expression $W = dNk_bT/\omega_b^2$, where frequency, ω_b, is the frequency of required volume fluctuations. The value of ω_b should be less then the lowest frequency which appears in the NVT temperature spectrum [Quigley04] in order to match the canonical distribution. d - dimensions, N - number of particles, k_b - Boltzmann constant, T - desired temperature.

NOTE The *langevinP.mass* can be set both directly and using the (*setMassByFrequency(frequency)*)

Adding to the integration:

```
>>> integrator.addExtension(langevinP)
```

It will define Langevin-Hoover barostat as a property of integrator.

One more example:

```
>>> rngBaro = espresso.esutil.RNG()
>>> lP = espresso.integrator.LangevinBarostat(system, rngBaro, desiredTemperature)
>>> lP.gammaP = .5
>>> lP.pressure = 1.0
>>> lP.mass = pow(10.0, 5)
>>> integrator.addExtension(lP)
```

Canceling the barostat:

If one do not need the pressure regulation in system anymore or need to switch the ensamble or whatever :)

```
>>> # define barostat with parameters
>>> rngBaro = espresso.esutil.RNG()
>>> lP = espresso.integrator.LangevinBarostat(system, rngBaro, desiredTemperature)
>>> lP.gammaP = .5
>>> lP.pressure = 1.0
>>> lP.mass = pow(10.0, 5)
>>> integrator.langevinBarostat = lP
>>> ...
>>> # some runs
>>> ...
>>> # disconnect barostat
>>> langevinBarostat.disconnect()
>>> # the next runs will not include the modification of integration equations
```

Connecting the barostat back after the disconnection

```
>>> langevinBarostat.connect()
```

References:

5.10 CoulombRSpace - Coulomb potential and interaction Objects (*R* space part)

This is the *R* space part of potential of Coulomb long range interaction according to the Ewald summation technique. Good explanation of Ewald summation could be found here [Allen89], [Deserno98].

Example:

```
>>> vl = espresso.VerletList(system, rspacecutoff+skin)
>>> coulombR_pot = espresso.interaction.CoulombRSpace(coulomb_prefactor, alpha, rspacecutoff)
>>> coulombR_int = espresso.interaction.VerletListCoulombRSpace(vl)
>>> coulombR_int.setPotential(type1=0, type2=0, potential = coulombR_pot)
>>> system.addInteraction(coulombR_int)
```

!IMPORTANT Coulomb interaction needs k-space part as well EwaldKSpace.

Definition:

It provides potential object *CoulombRSpace* and interaction object *VerletListCoulombRSpace*

The *potential* is based on parameters: Coulomb prefactor (coulomb_prefactor), Ewald parameter (alpha), and the cutoff in R space (rspacecutoff).

```
>>> coulombR_pot = espresso.interaction.CoulombRSpace(coulomb_prefactor, alpha, rspacecutoff)
```

Potential Properties:

- *coulombR_pot.prefactor*

  The property 'prefactor' defines the Coulomb prefactor.

- *coulombR_pot.alpha*

  The property 'alpha' defines the Ewald parameter α.

- *coulombR_pot.cutoff*

  The property 'cutoff' defines the cutoff in R space.

The *interaction* is based on the Verlet list (VerletList)

```
>>> vl = espresso.VerletList(system, rspacecutoff+skin)
>>> coulombR_int = espresso.interaction.VerletListCoulombRSpace(vl)
```

It should include at least one potential

```
>>> coulombR_int.setPotential(type1=0, type2=0, potential = coulombR_pot)
```

Interaction Methods:

- *setPotential(type1, type2, potential)*

  This method sets the *potential* for the particles of *type1* and *type2*. It could be a bunch of potentials for the different particle types.

- *getVerletListLocal()*

  Access to the local Verlet list.

Adding the interaction to the system:

```
>>> system.addInteraction(coulombR_int)
```

5.11 CoulombKSpaceEwald - Coulomb potential and interaction Objects (*K* space part)

This is the *K* space part of potential of Coulomb long range interaction according to the Ewald summation technique. Good explanation of Ewald summation could be found here [Allen89], [Deserno98].

Example:

```
>>> ewaldK_pot = espresso.interaction.CoulombKSpaceEwald(system, coulomb_prefactor, alpha, kspacecut
>>> ewaldK_int = espresso.interaction.CellListCoulombKSpaceEwald(system.storage, ewaldK_pot)
>>> system.addInteraction(ewaldK_int)
```

!IMPORTANT Coulomb interaction needs *R* space part as well CoulombRSpace.

Definition:

It provides potential object *CoulombKSpaceEwald* and interaction object *CellListCoulombKSpaceEwald* based on all particles list.

The *potential* is based on the system information (System) and parameters: Coulomb prefactor (coulomb_prefactor), Ewald parameter (alpha), and the cutoff in K space (kspacecutoff).

```
>>> ewaldK_pot = espresso.interaction.CoulombKSpaceEwald(system, coulomb_prefactor, alpha, kspac
```

Potential Properties:

- *ewaldK_pot.prefactor*

  The property 'prefactor' defines the Coulomb prefactor.

- *ewaldK_pot.alpha*

  The property 'alpha' defines the Ewald parameter α.

- *ewaldK_pot.kmax*

  The property 'kmax' defines the cutoff in *K* space.

The *interaction* is based on the all particles list. It needs the information from Storage and *K* space part of potential.

```
>>> ewaldK_int = espresso.interaction.CellListCoulombKSpaceEwald(system.storage, ewaldK_pot)
```

Interaction Methods:

- *getPotential()*

  Access to the local potential.

Adding the interaction to the system:

```
>>> system.addInteraction(ewaldK_int)
```

References:

5.12 decomp.py - Auxiliary python functions

- *nodeGrid(n)*:

  It determines how the processors are distributed and how the cells are arranged. *n* - number of processes

- *cellGrid(box_size, node_grid, rc, skin)*:

  It returns an appropriate grid of cells.

- *tuneSkin(system, integrator, minSkin=0.01, maxSkin=1.2, precision=0.001)*:

  It tunes the skin size for the current system

- *printTimeVsSkin(system, integrator, minSkin=0.01, maxSkin=1.5, skinStep = 0.01)*:

  It prints time of running versus skin size in the range [minSkin, maxSkin] with the step skinStep

5.13 espresso

5.13.1 PMI - Parallel Method Invocation

PMI allows users to write serial Python scripts that use functions and classes that are executed in parallel.

PMI is intended to be used in data-parallel environments, where several threads run in parallel and can communicate via MPI.

In PMI mode, a single thread of control (a python script that runs on the *controller*, i.e. the MPI root task) can invoke arbitrary functions on all other threads (the *workers*) in parallel via *call()*, *invoke()* and *reduce()*. When the function on the workers return, the control is returned to the controller.

This model is equivalent to the "Fork-Join execution model" used e.g. in OpenMP.

PMI also allows to create parallel instances of object classes via *create()*, i.e. instances that have a corresponding object instance on all workers. *call()*, *invoke()* and *reduce()* can be used to call arbitrary methods of these instances.

to execute arbitrary code on all workers, *exec_()* can be used, and to import python modules to all workers, use 'import_()'.

Main program

On the workers, the main program of a PMI script usually consists of a single call to the function *startWorkerLoop()*. On the workers, this will start an infinite loop on the workers that waits to receive the next PMI call, while it will immediately return on the controller. On the workers, the loop ends only, when one of the commands *finalizeWorkers()* or *stopWorkerLoop()* is issued on the controller. A typical PMI main program looks like this:

```
>>> # compute 2*factorial(42) in parallel
>>> import pmi
>>>
>>> # start the worker loop
>>> # on the controller, this function returns immediately
>>> pmi.startWorkerLoop()
>>>
>>> # Do the parallel computation
>>> pmi.import_('math')
>>> pmi.reduce('lambda a,b: a+b', 'math.factorial', 42)
>>>
>>> # exit all workers
>>> pmi.finalizeWorkers()
```

Instead of using *finalizeWorkers()* at the end of the script, you can call *registerAtExit()* anywhere else, which will cause *finalizeWorkers()* to be called when the python interpreter exits.

Alternatively, it is possible to use PMI in an SPMD-like fashion, where each call to a PMI command on the controller must be accompanied by a corresponding call on the worker. This can be either a simple call to *receive()* that accepts any PMI command, or a call to the identical PMI command. In that case, the arguments of the call to the PMI command on the workers are ignored. In this way, it is possible to write SPMD scripts that profit from the PMI communication patterns.

```
>>> # compute 2*factorial(42) in parallel
>>> import pmi
>>>
>>> pmi.exec_('import math')
>>> pmi.reduce('lambda a,b: a+b', 'math.factorial', 42)
```

To start the worker loop, the command *startWorkerLoop()* can be issued on the workers. To stop the worker loop, *stopWorkerLoop()* can be issued on the controller, which will end the worker loop without exiting the workers.

Controller commands

These commands can be called in the controller script. When any of these commands is issued on a worker during the worker loop, a *UserError* is raised.

- *call()*, *invoke()*, *reduce()* to call functions and methods in parallel
- *create()* to create parallel object instances
- *exec_()* and *import_()* to execute arbitrary python code in parallel and to import classes and functions into the global namespace of pmi.
- *sync()* to make sure that all deleted PMI objects have been deleted.
- *finalizeWorkers()* to stop and exit all workers
- *registerAtExit()* to make sure that finalizeWorkers() is called when python exits on the controller
- *stopWorkerLoop()* to interrupt the worker loop an all workers and to return control to the single workers

Worker commands

These commands can be called on a worker.

- *startWorkerLoop()* to start the worker loop
- *receive()* to receive a single PMI command
- *call()*, *invoke()*, *reduce()*, *create()* and *exec_()* to receive a single corresponding PMI command. Note that these commands will ignore any arguments when called on a worker.

PMI Proxy metaclass

The *Proxy* metaclass can be used to easily generate front-end classes to distributed PMI classes. . . .

Useful constants and variables

The pmi module defines the following useful constants and variables:

- *isController* is True when used on the controller, False otherwise
- *isWorker* = not isController
- *ID* is the rank of the MPI task
- *CONTROLLER* is the rank of the Controller (normally the MPI root)
- *workerStr* is a string describing the thread ('Worker #' or 'Controller')
- *inWorkerLoop* is True, if PMI currently executes the worker loop on the workers.

espresso.pmi.**exec_**(**args*)
Controller command that executes arbitrary python code on all (active) workers.

exec_() allows to execute arbitrary Python code on all workers. It can be used to define classes and functions on all workers. Modules should not be imported via exec_(), instead import_() should be used.

Each element of args should be string that is executed on all workers.

Example:

```
>>> pmi.exec_('import hello')
>>> hw = pmi.create('hello.HelloWorld')
```

espresso.pmi.**import_**(**args*)
Controller command that imports python modules on all (active) workers.

Each element of args should be a module name that is imported to all workers.

Example:

```
>>> pmi.import_('hello')
>>> hw = pmi.create('hello.HelloWorld')
```

espresso.pmi.**create**(*cls=None*, **args*, ***kwds*)
Controller command that creates an object on all workers.

cls describes the (new-style) class that should be instantiated. args are the arguments to the constructor of the class. Only classes that are known to PMI can be used, that is, classes that have been imported to pmi via *exec_()* or *import_()*.

Example:

```
>>> pmi.exec_('import hello')
>>> hw = pmi.create('hello.HelloWorld')
>>> print(hw)
MPI process #0: Hello World!
MPI process #1: Hello World!
...
```

Alternative: Note that in this case the class has to be imported to the calling module *and* via PMI.

```
>>> import hello
>>> pmi.exec_('import hello')
>>> hw = pmi.create(hello.HelloWorld)
>>> print(hw)
MPI process #0: Hello World!
MPI process #1: Hello World!
...
```

espresso.pmi.**call**(**args*, ***kwds*)

Call a function on all workers, returning only the return value on the controller.

function denotes the function that is to be called, args and kwds are the arguments to the function. If kwds contains keys that start with with the prefix '__pmictr_', they are stripped of the prefix and are passed only to the controller. If the function should return any results, it will be locally returned. Only functions that are known to PMI can be used, that is functions that have been imported to pmi via *exec_()* or *import_()*.

Example:

```
>>> pmi.exec_('import hello')
>>> hw = pmi.create('hello.HelloWorld')
>>> pmi.call(hw.hello)
>>> # equivalent:
>>> pmi.call('hello.HelloWorld', hw)
```

Note, that you can use only functions that are know to PMI when *call()* is called, i.e. functions in modules that have been imported via *exec_()*.

espresso.pmi.**invoke**(**args*, ***kwds*)

Invoke a function on all workers, gathering the return values into a list.

function denotes the function that is to be called, args and kwds are the arguments to the function. If kwds contains keys that start with with the prefix '__pmictr_', they are stripped of the prefix and are passed only to the controller.

On the controller, invoke() returns the results of the different workers as a list. On the workers, invoke returns None. Only functions that are known to PMI can be used, that is functions that have been imported to pmi via *exec_()* or *import_()*.

Example:

```
>>> pmi.exec_('import hello')
>>> hw = pmi.create('hello.HelloWorld')
>>> messages = pmi.invoke(hw.hello())
>>> # alternative:
>>> messages = pmi.invoke('hello.HelloWorld.hello', hw)
```

espresso.pmi.**reduce**(**args*, ***kwds*)

Invoke a function on all workers, reducing the return values to a single value.

reduceOp is the (associative) operator that is used to process the return values, function denotes the function that is to be called, args and kwds are the arguments to the function. If kwds contains keys that start with with the prefix '__pmictr_', they are stripped of the prefix and are passed only to the controller.

reduce() reduces the results of the different workers into a single value via the operation reduceOp. reduceOp is assumed to be associative. Both reduceOp and function have to be known to PMI, that is they must have been imported to pmi via *exec_()* or *import_()*.

Example:

```
>>> pmi.exec_('import hello')
>>> pmi.exec_('joinstr=lambda a,b: "\n".join(a,b)')
>>> hw = pmi.create('hello.HelloWorld')
>>> print(pmi.reduce('joinstr', hw.hello()))
>>> # equivalent:
>>> print(
...   pmi.reduce('lambda a,b: "\n".join(a,b)',
...              'hello.HelloWorld.hello', hw)
...              )
```

espresso.pmi.**sync**()
Controller command that deletes the PMI objects on the workers that have already been deleted on the controller.

espresso.pmi.**receive**(*expected=None*)
Worker command that receives and handles the next PMI command.

This function waits to receive and handle a single PMI command. If expected is not None and the received command does not equal expected, raise a *UserError*.

espresso.pmi.**startWorkerLoop**()
Worker command that starts the main worker loop.

This function starts a loop that expects to receive PMI commands until *stopWorkerLoop()* or *finalizeWorkers()* is called on the controller.

espresso.pmi.**finalizeWorkers**()
Controller command that stops and exits all workers.

espresso.pmi.**stopWorkerLoop**(*doExit=False*)
Controller command that stops all workers.

If doExit is set, the workers exit afterwards.

espresso.pmi.**registerAtExit**()
Controller command that registers the function *finalizeWorkers()* via atexit.

class espresso.pmi.**Proxy**(*name*, *bases*, *dict*)
A metaclass to be used to create frontend serial objects.

exception espresso.pmi.**UserError**(*msg*)
Raised when PMI has encountered a user error.

5.13.2 espresso.Exceptions

exception espresso.Exceptions.**Error**(*msg*)
Raised to show unrecoverable espresso errors.

exception espresso.Exceptions.**MissingFixedPairList**(*msg*)
Raised to indicate, that a FixedPairList object is missing

exception espresso.Exceptions.**ParticleDoesNotExistHere**(*msg*)
Raised to indicate, that a certain Particle does not exist on a CPU

exception espresso.Exceptions.**UnknownParticleProperty**(*msg*)
Raised to indicate, that a certain Particle property does not exists

5.13.3 espresso.FixedPairDistList

class espresso.FixedPairDistList.**FixedPairDistListLocal**(*storage*)
The (local) fixed pair list.

add(*pid1*, *pid2*)
add pair to fixed pair list

addPairs(*bondlist*)
Each processor takes the broadcasted bondlist and adds those pairs whose first particle is owned by this processor.

getPairs()
return the bonds of the GlobalPairList

getPairsDist()
return the bonds of the GlobalPairList

size()
count number of bonds in GlobalPairList, involves global reduction

5.13.4 espresso.FixedPairList

class espresso.FixedPairList.**FixedPairListLocal**(*storage*)
The (local) fixed pair list.

add(*pid1*, *pid2*)
add pair to fixed pair list

addBonds(*bondlist*)
Each processor takes the broadcasted bondlist and adds those pairs whose first particle is owned by this processor.

getBonds()
return the bonds of the GlobalPairList

getLongtimeMaxBondLocal()
return the maximum bond length this pairlist ever had (since reset or construction)

resetLongtimeMaxBond()
reset long time maximum bond to 0.0

size()
count number of bonds in GlobalPairList, involves global reduction

5.13.5 FixedPairListAdress - Object

The FixedPairListAdress is the Fixed Pair List to be used for AdResS or H-AdResS simulations. When creating the FixedPairListAdress one has to provide the storage and the tuples. Afterwards the bonds can be added. In the example "bonds" is a python list of the form ((pid1, pid2), (pid3, pid4), ...) where each inner pair defines a bond between the particles with the given particle ids.

Example - creating the FixedPairListAdress and adding bonds:

```
>>> ftpl = espresso.FixedTupleList(system.storage)
>>> fpl = espresso.FixedPairListAdress(system.storage, ftpl)
>>> fpl.addBonds(bonds)
```

class espresso.FixedPairListAdress.**FixedPairListAdressLocal**(*storage*, *fixedtupleList*)
The (local) fixed pair list.

add(*pid1*, *pid2*)
add pair to fixed pair list

addBonds(*bondlist*)
Each processor takes the broadcasted bondlist and adds those pairs whose first particle is owned by this processor.

getBonds()
return the bonds of the GlobalPairList

5.13.6 espresso.FixedQuadrupleAngleList

class espresso.FixedQuadrupleAngleList.**FixedQuadrupleAngleListLocal**(*storage*)
The (local) fixed quadruple list.

add(*pid1*, *pid2*, *pid3*, *pid4*)
add quadruple to fixed quadruple list

addQuadruples(*quadruplelist*)
Each processor takes the broadcasted quadruplelist and adds those quadruples whose first particle is owned by this processor.

getQuadruples()
return the quadruples of the GlobalQuadrupleList

getQuadruplesAngles()
return the quadruples with angle

size()
count number of Quadruples in GlobalQuadrupleList, involves global reduction

5.13.7 espresso.FixedQuadrupleList

class espresso.FixedQuadrupleList.**FixedQuadrupleListLocal**(*storage*)
The (local) fixed quadruple list.

add(*pid1*, *pid2*, *pid3*, *pid4*)
add quadruple to fixed quadruple list

addQuadruples(*quadruplelist*)
Each processor takes the broadcasted quadruplelist and adds those quadruples whose first particle is owned by this processor.

getQuadruples()
return the quadruples of the GlobalQuadrupleList

size()
count number of Quadruples in GlobalQuadrupleList, involves global reduction

5.13.8 espresso.FixedSingleList

class espresso.FixedSingleList.**FixedSingleListLocal**(*storage*)
The (local) fixed single list.

add(*pid1*)
add particle to fixed single list

addSingles(*singlelist*)
Each processor takes the broadcasted singlelist and adds those particles that are owned by this processor.

getSingles()
return the singles of the GlobalSingleList

size()
count number of particles in GlobalSingleList, involves global reduction

5.13.9 espresso.FixedTripleAngleList

class espresso.FixedTripleAngleList.**FixedTripleAngleListLocal**(*storage*)
The (local) fixed triple list.

add(*pid1*, *pid2*, *pid3*)
add triple to fixed triple list

addTriples(*triplelist*)
Each processor takes the broadcasted triplelist and adds those triples whose first particle is owned by this processor.

getTriples()
return the triples of the GlobalTripleList

getTriplesAngles()
return the triples of the GlobalTripleList

size()
count number of Triples in GlobalTripleList, involves global reduction

5.13.10 espresso.FixedTripleList

class espresso.FixedTripleList.**FixedTripleListLocal**(*storage*)
The (local) fixed triple list.

add(*pid1*, *pid2*, *pid3*)
add triple to fixed triple list

addTriples(*triplelist*)
Each processor takes the broadcasted triplelist and adds those triples whose first particle is owned by this processor.

getTriples()
return the triples of the GlobalTripleList

size()
count number of Triples in GlobalTripleList, involves global reduction

5.13.11 espresso.FixedTripleListAdress

class espresso.FixedTripleListAdress.**FixedTripleListAdressLocal**(*storage*, *fixedtupleList*)
The (local) fixed triple list.

add(*pid1*, *pid2*)
add pair to fixed triple list

addTriples(*triplelist*)
Each processor takes the broadcasted triplelist and adds those pairs whose first particle is owned by this processor.

5.13.12 espresso.FixedTupleList

class espresso.FixedTupleList.**FixedTupleListLocal**(*storage*)
The (local) fixed tuple list.

size()
count number of Tuple in GlobalTupleList, involves global reduction

5.13.13 FixedTupleListAdress - Object

The FixedTupleListAdress is important for AdResS and H-AdResS simulations. It is the connection between the atomistic and coarse-grained particles. It defines which atomistic particles belong to which coarse-grained particle. In the following example "tuples" is a python list of the form ((pid_CG1, pidAT11, pidAT12, pidAT13, ...), (pid_CG2, pidAT21, pidAT22, pidAT23, ...), ...). Each inner list (pid_CG1, pidAT11, pidAT12, pidAT13, ...) defines a tuple. The first number is the particle id of the coarse-grained particle while the following numbers are the particle ids of the corresponding atomistic particles.

Example - creating the FixedTupleListAdress:

```
>>> ftpl = espresso.FixedTupleListAdress(system.storage)
>>> ftpl.addTuples(tuples)
>>> system.storage.setFixedTuples(ftpl)
```

class espresso.FixedTupleListAdress.**FixedTupleListAdressLocal**(*storage*)
The (local) fixed touple list.

addTuples(*tuplelist*)
Each processor takes the broadcasted tuplelist and adds those tuples whose virtual particle is owned by this processor.

5.13.14 espresso.Int3D

espresso.Int3D.**toInt3D**(**args*)
Try to convert the arguments to a Int3D, returns the argument, if it is already a Int3D.

espresso.Int3D.**toInt3DFromVector**(**args*)
Try to convert the arguments to a Int3D.

This function will only convert to a Int3D if x, y and z are specified.

5.13.15 espresso.MultiSystem

class espresso.MultiSystem.**MultiSystem**
MultiSystemIntegrator to simulate and analyze several systems in parallel.

class espresso.MultiSystem.**MultiSystemLocal**
Local MultiSystem to simulate and analyze several systems in parallel.

5.13.16 espresso.ParallelTempering

5.13.17 espresso.Particle

class espresso.Particle.**ParticleLocal**(*pid*, *storage*)
The local particle.

Throws an exception: * when the particle does not exists locally

TODO: Should throw an exception: * when a ghost particle is to be written * when data is to be read from a ghost that is not available

5.13.18 ParticleAccess - abstract base class for analysis/measurement/io

class espresso.ParticleAccess.**ParticleAccess**
Abstract base class

class espresso.ParticleAccess.**ParticleAccessLocal**
Abstract local base class

5.13.19 espresso.ParticleGroup

class espresso.ParticleGroup.**ParticleGroupLocal**(*storage*)
The local particle group.

5.13.20 espresso.Real3D

espresso.Real3D.**toReal3D**(**args*)
Try to convert the arguments to a Real3D, returns the argument, if it is already a Real3D.

espresso.Real3D.**toReal3DFromVector**(**args*)
Try to convert the arguments to a Real3D.

This function will only convert to a Real3D if x, y and z are specified.

5.13.21 RealND -

This is the object which represents N-dimensional vector. It is an extended Real3D, basicly, it hase the same functionallity but in N-dimetions. First of all it is usefull for classes in 'espresso.analysis'.

Description

...

espresso.RealND.**toRealND**(**args*)
Try to convert the arguments to a RealND, returns the argument, if it is already a RealND.

espresso.RealND.**toRealNDFromVector**(**args*)
Try to convert the arguments to a RealND.

This function will only convert to a RealND if x, y and z are specified.

5.13.22 espresso.Settle

class espresso.Settle.**SettleLocal**(*storage*, *integrator*, *mO=16.0*, *mH=1.0*, *distHH=1.58*, *distOH=1.0*)

The (local) settle.

addMolecules(*moleculelist*)

Each processor takes the broadcasted list.

5.13.23 espresso.Tensor

espresso.Tensor.**toTensor**(**args*)

Try to convert the arguments to a Tensor, returns the argument, if it is already a Tensor.

espresso.Tensor.**toTensorFromVector**(**args*)

Try to convert the arguments to a Tensor.

This function will only convert to a Tensor if x, y and z are specified.

5.13.24 espresso.VerletList

class espresso.VerletList.**VerletListLocal**(*system*, *cutoff*, *exclusionlist=*[ ])

The (local) verlet list.

exclude(*exclusionlist*)

Each processor takes the broadcasted exclusion list and adds it to its list.

getAllPairs()

return the pairs of the local verlet list

localSize()

count number of pairs in local VerletList

totalSize()

count number of pairs in VerletList, involves global reduction

5.13.25 VerletListAdress - Object

The VerletListAdress is the Verlet List to be used for AdResS or H-AdResS simulations. When creating the VerletListAdress one has to provide the system and specify both cutoff for the CG interaction and adrcutoff for the atomistic interaction. Often, it is important to set the atomistic adrcutoff much bigger than the actual interaction's cutoff would be, since also the atomistic part of the VerletListAdress (adrPairs) is built based on the coarse-grained particle positions. For a much larger coarse-grained cutoff it is for example possible to also set the atomistic cutoff on the same value as the coarse-grained one.

Furthermore, the sizes of the explicit and hybrid region have to be provided (dEx and dHy in the example below) and the center of the atomistic region has to be set (adrCenter). In the current implementation this results in a resolution change along the x-direction of the box. A spherical symmetry can be obtained by only minor code changes.

Bascially the VerListAdress provides 4 lists:

- adrZone: A list which holds all particles in the atomistic and hybrid region
- cgZone: A list which holds all particles in the coarse-grained region

- adrPairs: A list which holds all pairs which have at least one particle in the adrZone, i.e. in the atomistic or hybrid region
- vlPairs: A list which holds all pairs which have both particles in the cgZone, i.e. in the coarse-grained region

Example - creating the VerletListAdress:

```
>>> vl      = espresso.VerletListAdress(system, cutoff=rc, adrcut=rc, dEx=ex_size, dHy=hy_size, adrCe
```

class espresso.VerletListAdress.**VerletListAdressLocal**(*system*, *cutoff*, *adrcut*, *dEx*, *dHy*, *adrCenter*=[], *pids*=[], *exclusionlist*=[])

The (local) verlet list AdResS

addAdrParticles(*pids*, *rebuild=True*)
Each processor takes the broadcasted atomistic particles and adds it to its list.

exclude(*exclusionlist*)
Each processor takes the broadcasted exclusion list and adds it to its list.

totalSize()
count number of pairs in VerletList, involves global reduction

5.13.26 espresso.VerletListTriple

class espresso.VerletListTriple.**VerletListTripleLocal**(*system*, *cutoff*, *exclusionlist*=[])

The (local) verlet triple list

exclude(*exclusionlist*)
Each processor takes the broadcasted exclusion list and adds it to its list.

getAllTriples()
return the triples of the local verlet list

localSize()
count number of triples in local VerletListTriple

totalSize()
count number of triples in VerletListTriple, involves global reduction

5.14 analysis

5.14.1 espresso.analysis.AllParticlePos

class espresso.analysis.AllParticlePos.**AllParticlePosLocal**

Abstract local base class for observables.

5.14.2 AnalysisBase - abstract base class for analysis/measurement

This abstract base class provides the interface and some basic functionality for classes that do analysis or observable measurements

It provides the following methods:

- **performMeasurement()** computes the observable and updates average and standard deviation
- **reset()** resets average and standard deviation

- **compute()** computes the instant value of the observable, return value is a python list or a scalar
- **getAverageValue()** returns the average value for the observable and the standard deviation, return value is a python list
- **getNumberOfMeasurements()** counts the number of measurements that have been performed (standalone or in integrator) does _not_ include measurements that have been done using "compute()"

class espresso.analysis.AnalysisBase.**AnalysisBase**
Abstract base class for observable.

class espresso.analysis.AnalysisBase.**AnalysisBaseLocal**
Abstract local base class for observables.

5.14.3 espresso.analysis.Autocorrelation

class espresso.analysis.Autocorrelation.**Autocorrelation**
Class for parallel analysis

class espresso.analysis.Autocorrelation.**AutocorrelationLocal**(*system*)
The (local) storage of configurations.

5.14.4 espresso.analysis.CenterOfMass

class espresso.analysis.CenterOfMass.**CenterOfMassLocal**(*system*)
The (local) compute of center-of-mass.

5.14.5 espresso.analysis.ConfigsParticleDecomp

class espresso.analysis.ConfigsParticleDecomp.**ConfigsParticleDecomp**
Abstract base class for parallel analysis based on particle decomposition.

class espresso.analysis.ConfigsParticleDecomp.**ConfigsParticleDecompLocal**(*system*)
The (local) storage of configurations.

5.14.6 Configurations - Configurations Object

- *gather()* add configuration to trajectory
- *clear()* clear trajectory
- *back()* get last configuration of trajectory
- *capacity* maximum number of configurations in trajectory further adding (*gather()*) configurations results in erasing oldest configuration before adding new one capacity=0 means: infinite capacity (until memory is full)
- *size* number of stored configurations

usage:

storing trajectory

```
>>> configurations = espresso.Configurations(system)
>>> configurations.gather()
>>> for k in range(100):
>>>   integrator.run(100)
>>>   configurations.gather()
```

accessing trajectory data:

iterate over all stored configurations:

```
>>> for conf in configurations:
```

iterate over all particles stored in configuration:

```
>>>   for pid in conf
>>>     particle_coords = conf[pid]
>>>     print pid, particle_coords
```

access particle with id <pid> of stored configuration <n>:

```
>>> print "particle coord: ",configurations[n][pid]
```

class espresso.analysis.Configurations.**ConfigurationsLocal**(*system*)
The (local) storage of configurations.

5.14.7 ConfigurationsExt - ConfigurationsExt Object

- *gather()* add configuration to trajectory
- *clear()* clear trajectory
- *back()* get last configuration of trajectory
- *capacity* maximum number of configurations in trajectory further adding (*gather()*) configurations results in erasing oldest configuration before adding new one capacity=0 means: infinite capacity (until memory is full)
- *size* number of stored configurations

usage:

storing trajectory

```
>>> configurations = espresso.ConfigurationsExt(system)
>>> configurations.gather()
>>> for k in range(100):
>>>   integrator.run(100)
>>>   configurations.gather()
```

accessing trajectory data:

iterate over all stored configurations:

```
>>> for conf in configurations:
```

iterate over all particles stored in configuration:

```
>>>   for pid in conf
>>>     particle_coords = conf[pid]
>>>     print pid, particle_coords
```

access particle with id <pid> of stored configuration <n>:

```
>>> print "particle coord: ",configurations[n][pid]
```

class espresso.analysis.ConfigurationsExt.**ConfigurationsExtLocal**(*system*)
The (local) storage of configurations.

5.14.8 espresso.analysis.Energy

class espresso.analysis.Energy.**EnergyKin**(*system*, *per_atom=False*)
Kinetic energy of the system.

class espresso.analysis.Energy.**EnergyPot**(*system*, *per_atom=False*)
Potential energy of the system.

class espresso.analysis.Energy.**EnergyTot**(*system*, *per_atom=False*)
Total energy (EKin + EPot) of the system.

5.14.9 espresso.analysis.IntraChainDistSq

class espresso.analysis.IntraChainDistSq.**IntraChainDistSqLocal**(*system*, *fpl*)
The (local) IntraChainDistSq object

5.14.10 espresso.analysis.LBOutput

class espresso.analysis.LBOutput.**LBOutputLocal**
The (local) compute of LBOutput.

5.14.11 espresso.analysis.LBOutputProfileVzOfX

class espresso.analysis.LBOutputProfileVzOfX.**LBOutputProfileVzOfXLocal**(*system*, *latticeboltzmann*)
The (local) compute of LBOutputProfileVzOfX.

5.14.12 espresso.analysis.LBOutputScreen

class espresso.analysis.LBOutputScreen.**LBOutputScreenLocal**(*system*, *latticeboltzmann*)
The (local) compute of LBOutputScreen.

5.14.13 espresso.analysis.LBOutputVzInTime

class espresso.analysis.LBOutputVzInTime.**LBOutputVzInTimeLocal**(*system*, *latticeboltzmann*)
The (local) compute of LBOutputVzInTime.

5.14.14 espresso.analysis.MaxPID

class espresso.analysis.MaxPID.**MaxPIDLocal**(*system*)
The (local) compute of the maximum pid number of the system.

5.14.15 espresso.analysis.MeanSquareDispl

class espresso.analysis.MeanSquareDispl.**MeanSquareDisplLocal**(*system*)
The (local) compute autocorrelation f.

5.14.16 espresso.analysis.NPart

class espresso.analysis.NPart.**NPartLocal**(*system*)
The (local) compute of the number of particles of the system.

5.14.17 espresso.analysis.NeighborFluctuation

class espresso.analysis.NeighborFluctuation.**NeighborFluctuationLocal**(*system*, *radius*)
The (local) compute of the neighbor fluctuations (<n^2>-<n>^2) in the number of particles found in a sphere of radius d around particle i.

5.14.18 espresso.analysis.Observable

class espresso.analysis.Observable.**Observable**
Abstract base class for observable.

class espresso.analysis.Observable.**ObservableLocal**
Abstract local base class for observables.

5.14.19 espresso.analysis.OrderParameter

class espresso.analysis.OrderParameter.**OrderParameterLocal**(*system*, *cutoff*, *angular_momentum=6*, *do_cluster_analysis=False*, *include_surface_particles=False*, *ql_low=-1.0*, *ql_high=1.0*)
The (local) compute of temperature.

5.14.20 espresso.analysis.ParticleRadiusDistribution

class espresso.analysis.ParticleRadiusDistribution.**ParticleRadiusDistributionLocal**(*system*)
The (local) compute of the particle radius distribution.

5.14.21 espresso.analysis.Pressure

class espresso.analysis.Pressure.**PressureLocal**(*system*)
The (local) compute of pressure.

5.14.22 PressureTensor - Analysis

This class computes the pressure tensor of the system. It can be used as standalone class in python as well as in combination with the integrator extension ExtAnalyze.

Standalone Usage:

```
>>> pt = espresso.analysis.PressureTensor(system)
>>> print "pressure tensor of current configuration = ", pt.compute()
```

or

```
>>> pt = espresso.analysis.PressureTensor(system)
>>> for k in range(100):
>>>     integrator.run(100)
>>>     pt.performMeasurement()
>>> print "average pressure tensor = ", pt.getAverageValue()
```

Usage in integrator with ExtAnalyze:

```
>>> pt           = espresso.analysis.PressureTensor(system)
>>> extension_pt = espresso.integrator.ExtAnalyze(pt , interval=100)
>>> integrator.addExtension(extension_pt)
>>> integrator.run(10000)
>>> pt_ave = pt.getAverageValue()
>>> print "average Pressure Tensor = ", pt_ave[:6]
>>> print "          std deviation = ", pt_ave[6:]
>>> print "number of measurements  = ", pt.getNumberOfMeasurements()
```

The following methods are supported:

- **performMeasurement()** computes the pressure tensor and updates average and standard deviation
- **reset()** resets average and standard deviation to 0
- **compute()** computes the instant pressure tensor, return value: [xx, yy, zz, xy, xz, yz]
- **getAverageValue()** returns the average pressure tensor and the standard deviation, return value: [xx, yy, zz, xy, xz, yz, +-xx, +-yy, +-zz, +-xy, +-xz, +-yz]
- **getNumberOfMeasurements()** counts the number of measurements that have been computed (standalone or in integrator) does _not_ include measurements that have been done using "compute()"

class espresso.analysis.PressureTensor.**PressureTensorLocal** (*system*)
The (local) compute of pressure tensor.

5.14.23 PressureTensorLayer - Analysis

This class computes the pressure tensor of the system in layer h0. It can be used as standalone class in python as well as in combination with the integrator extension ExtAnalyze.

Standalone Usage:

```
>>> pt = espresso.analysis.PressureTensorLayer(system, h0, dh)
>>> print "pressure tensor of current configuration = ", pt.compute()
```

or

```
>>> pt = espresso.analysis.PressureTensorLayer(system)
>>> for k in range(100):
>>>     integrator.run(100)
```

```
>>>     pt.performMeasurement()
>>> print "average pressure tensor = ", pt.getAverageValue()
```

Usage in integrator with ExtAnalyze:

```
>>> pt           = espresso.analysis.PressureTensorLayer(system)
>>> extension_pt = espresso.integrator.ExtAnalyze(pt , interval=100)
>>> integrator.addExtension(extension_pt)
>>> integrator.run(10000)
>>> pt_ave = pt.getAverageValue()
>>> print "average Pressure Tensor = ", pt_ave[:6]
>>> print "          std deviation = ", pt_ave[6:]
>>> print "number of measurements  = ", pt.getNumberOfMeasurements()
```

The following methods are supported:

- **performMeasurement()** computes the pressure tensor and updates average and standard deviation
- **reset()** resets average and standard deviation to 0
- **compute()** computes the instant pressure tensor in layer h0, return value: [xx, yy, zz, xy, xz, yz]
- **getAverageValue()** returns the average pressure tensor and the standard deviation, return value: [xx, yy, zz, xy, xz, yz, +-xx, +-yy, +-zz, +-xy, +-xz, +-yz]
- **getNumberOfMeasurements()** counts the number of measurements that have been computed (standalone or in integrator) does _not_ include measurements that have been done using "compute()"

class espresso.analysis.PressureTensorLayer.**PressureTensorLayerLocal**(*system*, *h0*, *dh*)

The (local) compute of pressure tensor.

5.14.24 PressureTensorMultiLayer - Analysis

This class computes the pressure tensor of the system in *n* layers. Layers are perpendicular to Z direction and are equidistant(distance is Lz/n). It can be used as standalone class in python as well as in combination with the integrator extension ExtAnalyze.

Standalone Usage:

```
>>> pt = espresso.analysis.PressureTensorMultiLayer(system, n, dh)
>>> for i in range(n):
>>>     print "pressure tensor in layer %d: %s" % ( i, pt.compute())
```

or

```
>>> pt = espresso.analysis.PressureTensorMultiLayer(system, n, dh)
>>> for k in range(100):
>>>     integrator.run(100)
>>>     pt.performMeasurement()
>>> for i in range(n):
>>>     print "average pressure tensor in layer %d: %s" % ( i, pt.compute())
```

Usage in integrator with ExtAnalyze:

```
>>> pt           = espresso.analysis.PressureTensorMultiLayer(system, n, dh)
>>> extension_pt = espresso.integrator.ExtAnalyze(pt , interval=100)
>>> integrator.addExtension(extension_pt)
>>> integrator.run(10000)
>>> pt_ave = pt.getAverageValue()
>>> for i in range(n):
>>>   print "average Pressure Tensor = ", pt_ave[i][:6]
>>>   print "          std deviation = ", pt_ave[i][6:]
>>> print "number of measurements  = ", pt.getNumberOfMeasurements()
```

The following methods are supported:

- **performMeasurement()** computes the pressure tensor and updates average and standard deviation
- **reset()** resets average and standard deviation to 0
- **compute()** computes the instant pressure tensor in *n* layers, return value: [xx, yy, zz, xy, xz, yz]
- **getAverageValue()** returns the average pressure tensor and the standard deviation, return value: [xx, yy, zz, xy, xz, yz, +-xx, +-yy, +-zz, +-xy, +-xz, +-yz]
- **getNumberOfMeasurements()** counts the number of measurements that have been computed (standalone or in integrator) does _not_ include measurements that have been done using "compute()"

class espresso.analysis.PressureTensorMultiLayer.**PressureTensorMultiLayerLocal**(*system*, *n*, *dh*)

The (local) compute of pressure tensor.

5.14.25 espresso.analysis.RDFatomistic

class espresso.analysis.RDFatomistic.**RDFatomisticLocal**(*system*)

The (local) compute the radial distr function.

5.14.26 espresso.analysis.RadialDistrF

class espresso.analysis.RadialDistrF.**RadialDistrFLocal**(*system*)

The (local) compute the radial distr function.

5.14.27 espresso.analysis.StaticStructF

class espresso.analysis.StaticStructF.**StaticStructFLocal**(*system*)

The (local) compute the static structure function.

5.14.28 espresso.analysis.Temperature

class espresso.analysis.Temperature.**TemperatureLocal**(*system*)

The (local) compute of temperature.

5.14.29 espresso.analysis.Test

class espresso.analysis.Test.**TestLocal**(*system*)
The (local) test of analysis.

5.14.30 espresso.analysis.Velocities

class espresso.analysis.Velocities.**VelocitiesLocal**(*system*)
The (local) storage of configurations.

5.14.31 espresso.analysis.VelocityAutocorrelation

class espresso.analysis.VelocityAutocorrelation.**VelocityAutocorrelationLocal**(*system*)
The (local) compute autocorrelation f.

5.14.32 espresso.analysis.Viscosity

class espresso.analysis.Viscosity.**Viscosity**(*system*)
Class for parallel analysis

class espresso.analysis.Viscosity.**ViscosityLocal**(*system*)
The (local) storage of configurations.

5.14.33 espresso.analysis.XDensity

class espresso.analysis.XDensity.**XDensityLocal**(*system*)
The (local) compute the density profile in x direction.

5.14.34 espresso.analysis.XPressure

class espresso.analysis.XPressure.**XPressureLocal**(*system*)
The (local) compute the pressure profile in x direction.

5.15 bc

5.15.1 BC - Boundary Condition Object

This is the abstract base class for all boundary condition objects. It cannot be used directly. All derived classes implement at least the following methods:

- *getMinimumImageVector(pos1, pos2)*
- *getFoldedPosition(pos, imageBox)*
- *getUnfoldedPosition(pos, imageBox)*
- *getRandomPos()*

pos, *pos1* and *pos2* are particle coordinates (type: (*float*, *float*, *float*)). *imageBox* (type: (*int*, *int*, *int*)) specifies the

5.15.2 OrthorhombicBC - Object

Like all boundary condition objects, this class implements all the methods of the base class **BC** , which are described in detail in the documentation of the abstract class **BC**.

The OrthorhombicBC class is responsible for the orthorhombic boundary condition. Currently only periodic boundary conditions are supported.

Example:

```
>>> boxsize = (Lx, Ly, Lz)
>>> bc = espresso.bc.OrthorhombicBC(rng, boxsize)
```

5.16 check

5.16.1 espresso.check.System

5.17 esutil

5.17.1 espresso.esutil.GammaVariate

5.17.2 espresso.esutil.Grid

5.17.3 espresso.esutil.NormalVariate

5.17.4 espresso.esutil.RNG

5.17.5 espresso.esutil.UniformOnSphere

5.17.6 espresso.esutil.collectives

espresso.esutil.collectives.**locateItem**(*here*)

locate the node with here=True (e.g. indicating that data of a distributed storage is on the local node). This is a collective SPMD function.

here is a boolean value, which should be True on at most one node. Returns on the controller the number of the node with here=True, or an KeyError exception if no node had the item, i.e. had here=True.

5.18 external

Homogeneous Transformation Matrices and Quaternions.

A library for calculating 4x4 matrices for translating, rotating, reflecting, scaling, shearing, projecting, orthogonalizing, and superimposing arrays of 3D homogeneous coordinates as well as for converting between rotation matrices, Euler angles, and quaternions. Also includes an Arcball control object and functions to decompose transformation matrices.

Authors Christoph Gohlke, Laboratory for Fluorescence Dynamics, University of California, Irvine

Version 2011.01.25

5.18.1 Requirements

- Python 2.6 or 3.1
- Numpy 1.5
- transformations.c 2010.04.10 (optional implementation of some functions in C)

5.18.2 Notes

The API is not stable yet and is expected to change between revisions.

This Python code is not optimized for speed. Refer to the transformations.c module for a faster implementation of some functions.

Documentation in HTML format can be generated with epydoc.

Matrices (M) can be inverted using numpy.linalg.inv(M), concatenated using numpy.dot(M0, M1), or used to transform homogeneous coordinates (v) using numpy.dot(M, v) for shape (4, *) "point of arrays", respectively numpy.dot(v, M.T) for shape (*, 4) "array of points".

Use the transpose of transformation matrices for OpenGL glMultMatrixd().

Calculations are carried out with numpy.float64 precision.

Vector, point, quaternion, and matrix function arguments are expected to be "array like", i.e. tuple, list, or numpy arrays.

Return types are numpy arrays unless specified otherwise.

Angles are in radians unless specified otherwise.

Quaternions w+ix+jy+kz are represented as [w, x, y, z].

A triple of Euler angles can be applied/interpreted in 24 ways, which can be specified using a 4 character string or encoded 4-tuple:

Axes 4-string: e.g. 'sxyz' or 'ryxy'

- first character : rotations are applied to 's'tatic or 'r'otating frame
- remaining characters : successive rotation axis 'x', 'y', or 'z'

Axes 4-tuple: e.g. (0, 0, 0, 0) or (1, 1, 1, 1)

- inner axis: code of axis ('x':0, 'y':1, 'z':2) of rightmost matrix.
- parity : even (0) if inner axis 'x' is followed by 'y', 'y' is followed by 'z', or 'z' is followed by 'x'. Otherwise odd (1).
- repetition : first and last axis are same (1) or different (0).
- frame : rotations are applied to static (0) or rotating (1) frame.

5.18.3 References

1. Matrices and transformations. Ronald Goldman. In "Graphics Gems I", pp 472-475. Morgan Kaufmann, 1990.
2. More matrices and transformations: shear and pseudo-perspective. Ronald Goldman. In "Graphics Gems II", pp 320-323. Morgan Kaufmann, 1991.
3. Decomposing a matrix into simple transformations. Spencer Thomas. In "Graphics Gems II", pp 320-323. Morgan Kaufmann, 1991.

4. Recovering the data from the transformation matrix. Ronald Goldman. In "Graphics Gems II", pp 324-331. Morgan Kaufmann, 1991.
5. Euler angle conversion. Ken Shoemake. In "Graphics Gems IV", pp 222-229. Morgan Kaufmann, 1994.
6. Arcball rotation control. Ken Shoemake. In "Graphics Gems IV", pp 175-192. Morgan Kaufmann, 1994.
7. Representing attitude: Euler angles, unit quaternions, and rotation vectors. James Diebel. 2006.
8. A discussion of the solution for the best rotation to relate two sets of vectors. W Kabsch. Acta Cryst. 1978. A34, 827-828.
9. Closed-form solution of absolute orientation using unit quaternions. BKP Horn. J Opt Soc Am A. 1987. 4(4):629-642.
10. Quaternions. Ken Shoemake. http://www.sfu.ca/~jwa3/cmpt461/files/quatut.pdf
11. From quaternion to matrix and back. JMP van Waveren. 2005. http://www.intel.com/cd/ids/developer/asmo-na/eng/293748.htm
12. Uniform random rotations. Ken Shoemake. In "Graphics Gems III", pp 124-132. Morgan Kaufmann, 1992.
13. Quaternion in molecular modeling. CFF Karney. J Mol Graph Mod, 25(5):595-604
14. New method for extracting the quaternion from a rotation matrix. Itzhack Y Bar-Itzhack, J Guid Contr Dynam. 2000. 23(6): 1085-1087.

5.18.4 Examples

```
>>> alpha, beta, gamma = 0.123, -1.234, 2.345
>>> origin, xaxis, yaxis, zaxis = (0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1)
>>> I = identity_matrix()
>>> Rx = rotation_matrix(alpha, xaxis)
>>> Ry = rotation_matrix(beta, yaxis)
>>> Rz = rotation_matrix(gamma, zaxis)
>>> R = concatenate_matrices(Rx, Ry, Rz)
>>> euler = euler_from_matrix(R, 'rxyz')
>>> numpy.allclose([alpha, beta, gamma], euler)
True
>>> Re = euler_matrix(alpha, beta, gamma, 'rxyz')
>>> is_same_transform(R, Re)
True
>>> al, be, ga = euler_from_matrix(Re, 'rxyz')
>>> is_same_transform(Re, euler_matrix(al, be, ga, 'rxyz'))
True
>>> qx = quaternion_about_axis(alpha, xaxis)
>>> qy = quaternion_about_axis(beta, yaxis)
>>> qz = quaternion_about_axis(gamma, zaxis)
>>> q = quaternion_multiply(qx, qy)
>>> q = quaternion_multiply(q, qz)
>>> Rq = quaternion_matrix(q)
>>> is_same_transform(R, Rq)
True
>>> S = scale_matrix(1.23, origin)
>>> T = translation_matrix((1, 2, 3))
>>> Z = shear_matrix(beta, xaxis, origin, zaxis)
>>> R = random_rotation_matrix(numpy.random.rand(3))
>>> M = concatenate_matrices(T, R, Z, S)
>>> scale, shear, angles, trans, persp = decompose_matrix(M)
>>> numpy.allclose(scale, 1.23)
```

```
True
>>> numpy.allclose(trans, (1, 2, 3))
True
>>> numpy.allclose(shear, (0, math.tan(beta), 0))
True
>>> is_same_transform(R, euler_matrix(axes='sxyz', *angles))
True
>>> M1 = compose_matrix(scale, shear, angles, trans, persp)
>>> is_same_transform(M, M1)
True
>>> v0, v1 = random_vector(3), random_vector(3)
>>> M = rotation_matrix(angle_between_vectors(v0, v1), vector_product(v0, v1))
>>> v2 = numpy.dot(v0, M[:3,:3].T)
>>> numpy.allclose(unit_vector(v1), unit_vector(v2))
True
```

class espresso.external.transformations.**Arcball**(*initial=None*)

Virtual Trackball Control.

```
>>> ball = Arcball()
>>> ball = Arcball(initial=numpy.identity(4))
>>> ball.place([320, 320], 320)
>>> ball.down([500, 250])
>>> ball.drag([475, 275])
>>> R = ball.matrix()
>>> numpy.allclose(numpy.sum(R), 3.90583455)
True
>>> ball = Arcball(initial=[1, 0, 0, 0])
>>> ball.place([320, 320], 320)
>>> ball.setaxes([1,1,0], [-1, 1, 0])
>>> ball.setconstrain(True)
>>> ball.down([400, 200])
>>> ball.drag([200, 400])
>>> R = ball.matrix()
>>> numpy.allclose(numpy.sum(R), 0.2055924)
True
>>> ball.next()
```

down(*point*)

Set initial cursor window coordinates and pick constrain-axis.

drag(*point*)

Update current cursor window coordinates.

getconstrain()

Return state of constrain to axis mode.

matrix()

Return homogeneous rotation matrix.

next(*acceleration=0.0*)

Continue rotation in direction of last drag.

place(*center*, *radius*)

Place Arcball, e.g. when window size changes.

center [sequence[2]] Window coordinates of trackball center.

radius [float] Radius of trackball in window coordinates.

setaxes (**axes*)
Set axes to constrain rotations.

setconstrain (*constrain*)
Set state of constrain to axis mode.

espresso.external.transformations.**angle_between_vectors** (*v0*, *v1*, *directed=True*, *axis=0*)
Return angle between vectors.

If directed is False, the input vectors are interpreted as undirected axes, i.e. the maximum angle is pi/2.

```
>>> a = angle_between_vectors([1, -2, 3], [-1, 2, -3])
>>> numpy.allclose(a, math.pi)
True
>>> a = angle_between_vectors([1, -2, 3], [-1, 2, -3], directed=False)
>>> numpy.allclose(a, 0)
True
>>> v0 = [[2, 0, 0, 2], [0, 2, 0, 2], [0, 0, 2, 2]]
>>> v1 = [[3], [0], [0]]
>>> a = angle_between_vectors(v0, v1)
>>> numpy.allclose(a, [0., 1.5708, 1.5708, 0.95532])
True
>>> v0 = [[2, 0, 0], [2, 0, 0], [0, 2, 0], [2, 0, 0]]
>>> v1 = [[0, 3, 0], [0, 0, 3], [0, 0, 3], [3, 3, 3]]
>>> a = angle_between_vectors(v0, v1, axis=1)
>>> numpy.allclose(a, [1.5708, 1.5708, 1.5708, 0.95532])
True
```

espresso.external.transformations.**arcball_constrain_to_axis** (*point*, *axis*)
Return sphere point perpendicular to axis.

espresso.external.transformations.**arcball_map_to_sphere** (*point*, *center*, *radius*)
Return unit sphere coordinates from window coordinates.

espresso.external.transformations.**arcball_nearest_axis** (*point*, *axes*)
Return axis, which arc is nearest to point.

espresso.external.transformations.**clip_matrix** (*left*, *right*, *bottom*, *top*, *near*, *far*, *perspective=False*)
Return matrix to obtain normalized device coordinates from frustrum.

The frustrum bounds are axis-aligned along x (left, right), y (bottom, top) and z (near, far).

Normalized device coordinates are in range [-1, 1] if coordinates are inside the frustrum.

If perspective is True the frustrum is a truncated pyramid with the perspective point at origin and direction along z axis, otherwise an orthographic canonical view volume (a box).

Homogeneous coordinates transformed by the perspective clip matrix need to be dehomogenized (devided by w coordinate).

```
>>> frustrum = numpy.random.rand(6)
>>> frustrum[1] += frustrum[0]
>>> frustrum[3] += frustrum[2]
>>> frustrum[5] += frustrum[4]
>>> M = clip_matrix(perspective=False, *frustrum)
>>> numpy.dot(M, [frustrum[0], frustrum[2], frustrum[4], 1.0])
array([-1., -1., -1.,  1.])
>>> numpy.dot(M, [frustrum[1], frustrum[3], frustrum[5], 1.0])
array([ 1.,  1.,  1.,  1.])
>>> M = clip_matrix(perspective=True, *frustrum)
```

```
>>> v = numpy.dot(M, [frustrum[0], frustrum[2], frustrum[4], 1.0])
>>> v / v[3]
array([-1., -1., -1.,  1.])
>>> v = numpy.dot(M, [frustrum[1], frustrum[3], frustrum[4], 1.0])
>>> v / v[3]
array([ 1.,  1., -1.,  1.])
```

espresso.external.transformations.**compose_matrix**(*scale=None*, *shear=None*, *angles=None*, *translate=None*, *perspective=None*)

Return transformation matrix from sequence of transformations.

This is the inverse of the decompose_matrix function.

Sequence of transformations: scale : vector of 3 scaling factors shear : list of shear factors for x-y, x-z, y-z axes angles : list of Euler angles about static x, y, z axes translate : translation vector along x, y, z axes perspective : perspective partition of matrix

```
>>> scale = numpy.random.random(3) - 0.5
>>> shear = numpy.random.random(3) - 0.5
>>> angles = (numpy.random.random(3) - 0.5) * (2*math.pi)
>>> trans = numpy.random.random(3) - 0.5
>>> persp = numpy.random.random(4) - 0.5
>>> M0 = compose_matrix(scale, shear, angles, trans, persp)
>>> result = decompose_matrix(M0)
>>> M1 = compose_matrix(*result)
>>> is_same_transform(M0, M1)
True
```

espresso.external.transformations.**concatenate_matrices**(**matrices*)

Return concatenation of series of transformation matrices.

```
>>> M = numpy.random.rand(16).reshape((4, 4)) - 0.5
>>> numpy.allclose(M, concatenate_matrices(M))
True
>>> numpy.allclose(numpy.dot(M, M.T), concatenate_matrices(M, M.T))
True
```

espresso.external.transformations.**decompose_matrix**(*matrix*)

Return sequence of transformations from transformation matrix.

matrix [array_like] Non-degenerative homogeneous transformation matrix

Return tuple of: scale : vector of 3 scaling factors shear : list of shear factors for x-y, x-z, y-z axes angles : list of Euler angles about static x, y, z axes translate : translation vector along x, y, z axes perspective : perspective partition of matrix

Raise ValueError if matrix is of wrong type or degenerative.

```
>>> T0 = translation_matrix((1, 2, 3))
>>> scale, shear, angles, trans, persp = decompose_matrix(T0)
>>> T1 = translation_matrix(trans)
>>> numpy.allclose(T0, T1)
True
>>> S = scale_matrix(0.123)
>>> scale, shear, angles, trans, persp = decompose_matrix(S)
>>> scale[0]
0.123
>>> R0 = euler_matrix(1, 2, 3)
>>> scale, shear, angles, trans, persp = decompose_matrix(R0)
```

```
>>> R1 = euler_matrix(*angles)
>>> numpy.allclose(R0, R1)
True
```

espresso.external.transformations.**euler_from_matrix**(*matrix*, *axes='sxyz'*)

Return Euler angles from rotation matrix for specified axis sequence.

axes : One of 24 axis sequences as string or encoded tuple

Note that many Euler angle triplets can describe one matrix.

```
>>> R0 = euler_matrix(1, 2, 3, 'syxz')
>>> al, be, ga = euler_from_matrix(R0, 'syxz')
>>> R1 = euler_matrix(al, be, ga, 'syxz')
>>> numpy.allclose(R0, R1)
True
>>> angles = (4.0*math.pi) * (numpy.random.random(3) - 0.5)
>>> for axes in _AXES2TUPLE.keys():
...    R0 = euler_matrix(axes=axes, *angles)
...    R1 = euler_matrix(axes=axes, *euler_from_matrix(R0, axes))
...    if not numpy.allclose(R0, R1): print(axes, "failed")
```

espresso.external.transformations.**euler_from_quaternion**(*quaternion*, *axes='sxyz'*)

Return Euler angles from quaternion for specified axis sequence.

```
>>> angles = euler_from_quaternion([0.99810947, 0.06146124, 0, 0])
>>> numpy.allclose(angles, [0.123, 0, 0])
True
```

espresso.external.transformations.**euler_matrix**(*ai*, *aj*, *ak*, *axes='sxyz'*)

Return homogeneous rotation matrix from Euler angles and axis sequence.

ai, aj, ak : Euler's roll, pitch and yaw angles axes : One of 24 axis sequences as string or encoded tuple

```
>>> R = euler_matrix(1, 2, 3, 'syxz')
>>> numpy.allclose(numpy.sum(R[0]), -1.34786452)
True
>>> R = euler_matrix(1, 2, 3, (0, 1, 0, 1))
>>> numpy.allclose(numpy.sum(R[0]), -0.383436184)
True
>>> ai, aj, ak = (4.0*math.pi) * (numpy.random.random(3) - 0.5)
>>> for axes in _AXES2TUPLE.keys():
...    R = euler_matrix(ai, aj, ak, axes)
>>> for axes in _TUPLE2AXES.keys():
...    R = euler_matrix(ai, aj, ak, axes)
```

espresso.external.transformations.**identity_matrix**()

Return 4x4 identity/unit matrix.

```
>>> I = identity_matrix()
>>> numpy.allclose(I, numpy.dot(I, I))
True
>>> numpy.sum(I), numpy.trace(I)
(4.0, 4.0)
>>> numpy.allclose(I, numpy.identity(4, dtype=numpy.float64))
True
```

espresso.external.transformations.**inverse_matrix**(*matrix*)

Return inverse of square transformation matrix.

```
>>> M0 = random_rotation_matrix()
>>> M1 = inverse_matrix(M0.T)
>>> numpy.allclose(M1, numpy.linalg.inv(M0.T))
True
>>> for size in range(1, 7):
...     M0 = numpy.random.rand(size, size)
...     M1 = inverse_matrix(M0)
...     if not numpy.allclose(M1, numpy.linalg.inv(M0)): print(size)
```

espresso.external.transformations.**is_same_transform**(*matrix0*, *matrix1*)

Return True if two matrices perform same transformation.

```
>>> is_same_transform(numpy.identity(4), numpy.identity(4))
True
>>> is_same_transform(numpy.identity(4), random_rotation_matrix())
False
```

espresso.external.transformations.**orthogonalization_matrix**(*lengths*, *angles*)

Return orthogonalization matrix for crystallographic cell coordinates.

Angles are expected in degrees.

The de-orthogonalization matrix is the inverse.

```
>>> O = orthogonalization_matrix((10., 10., 10.), (90., 90., 90.))
>>> numpy.allclose(O[:3, :3], numpy.identity(3, float) * 10)
True
>>> O = orthogonalization_matrix([9.8, 12.0, 15.5], [87.2, 80.7, 69.7])
>>> numpy.allclose(numpy.sum(O), 43.063229)
True
```

espresso.external.transformations.**projection_from_matrix**(*matrix*, *pseudo=False*)

Return projection plane and perspective point from projection matrix.

Return values are same as arguments for projection_matrix function: point, normal, direction, perspective, and pseudo.

```
>>> point = numpy.random.random(3) - 0.5
>>> normal = numpy.random.random(3) - 0.5
>>> direct = numpy.random.random(3) - 0.5
>>> persp = numpy.random.random(3) - 0.5
>>> P0 = projection_matrix(point, normal)
>>> result = projection_from_matrix(P0)
>>> P1 = projection_matrix(*result)
>>> is_same_transform(P0, P1)
True
>>> P0 = projection_matrix(point, normal, direct)
>>> result = projection_from_matrix(P0)
>>> P1 = projection_matrix(*result)
>>> is_same_transform(P0, P1)
True
>>> P0 = projection_matrix(point, normal, perspective=persp, pseudo=False)
>>> result = projection_from_matrix(P0, pseudo=False)
>>> P1 = projection_matrix(*result)
>>> is_same_transform(P0, P1)
True
>>> P0 = projection_matrix(point, normal, perspective=persp, pseudo=True)
>>> result = projection_from_matrix(P0, pseudo=True)
>>> P1 = projection_matrix(*result)
```

```
>>> is_same_transform(P0, P1)
True
```

espresso.external.transformations.**projection_matrix**(*point*, *normal*, *direction=None*, *perspective=None*, *pseudo=False*)

Return matrix to project onto plane defined by point and normal.

Using either perspective point, projection direction, or none of both.

If pseudo is True, perspective projections will preserve relative depth such that Perspective = dot(Orthogonal, PseudoPerspective).

```
>>> P = projection_matrix((0, 0, 0), (1, 0, 0))
>>> numpy.allclose(P[1:, 1:], numpy.identity(4)[1:, 1:])
True
>>> point = numpy.random.random(3) - 0.5
>>> normal = numpy.random.random(3) - 0.5
>>> direct = numpy.random.random(3) - 0.5
>>> persp = numpy.random.random(3) - 0.5
>>> P0 = projection_matrix(point, normal)
>>> P1 = projection_matrix(point, normal, direction=direct)
>>> P2 = projection_matrix(point, normal, perspective=persp)
>>> P3 = projection_matrix(point, normal, perspective=persp, pseudo=True)
>>> is_same_transform(P2, numpy.dot(P0, P3))
True
>>> P = projection_matrix((3, 0, 0), (1, 1, 0), (1, 0, 0))
>>> v0 = (numpy.random.rand(4, 5) - 0.5) * 20.0
>>> v0[3] = 1.0
>>> v1 = numpy.dot(P, v0)
>>> numpy.allclose(v1[1], v0[1])
True
>>> numpy.allclose(v1[0], 3.0-v1[1])
True
```

espresso.external.transformations.**quaternion_about_axis**(*angle*, *axis*)

Return quaternion for rotation about axis.

```
>>> q = quaternion_about_axis(0.123, (1, 0, 0))
>>> numpy.allclose(q, [0.99810947, 0.06146124, 0, 0])
True
```

espresso.external.transformations.**quaternion_conjugate**(*quaternion*)

Return conjugate of quaternion.

```
>>> q0 = random_quaternion()
>>> q1 = quaternion_conjugate(q0)
>>> q1[0] == q0[0] and all(q1[1:] == -q0[1:])
True
```

espresso.external.transformations.**quaternion_from_euler**(*ai*, *aj*, *ak*, *axes='sxyz'*)

Return quaternion from Euler angles and axis sequence.

ai, aj, ak : Euler's roll, pitch and yaw angles axes : One of 24 axis sequences as string or encoded tuple

```
>>> q = quaternion_from_euler(1, 2, 3, 'ryxz')
>>> numpy.allclose(q, [0.435953, 0.310622, -0.718287, 0.444435])
True
```

espresso.external.transformations.**quaternion_from_matrix**(*matrix*, *isprecise=False*)

Return quaternion from rotation matrix.

If isprecise=True, the input matrix is assumed to be a precise rotation matrix and a faster algorithm is used.

```
>>> q = quaternion_from_matrix(identity_matrix(), True)
>>> numpy.allclose(q, [1., 0., 0., 0.])
True
>>> q = quaternion_from_matrix(numpy.diag([1., -1., -1., 1.]))
>>> numpy.allclose(q, [0, 1, 0, 0]) or numpy.allclose(q, [0, -1, 0, 0])
True
>>> R = rotation_matrix(0.123, (1, 2, 3))
>>> q = quaternion_from_matrix(R, True)
>>> numpy.allclose(q, [0.9981095, 0.0164262, 0.0328524, 0.0492786])
True
>>> R = [[-0.545, 0.797, 0.260, 0], [0.733, 0.603, -0.313, 0],
...      [-0.407, 0.021, -0.913, 0], [0, 0, 0, 1]]
>>> q = quaternion_from_matrix(R)
>>> numpy.allclose(q, [0.19069, 0.43736, 0.87485, -0.083611])
True
>>> R = [[0.395, 0.362, 0.843, 0], [-0.626, 0.796, -0.056, 0],
...      [-0.677, -0.498, 0.529, 0], [0, 0, 0, 1]]
>>> q = quaternion_from_matrix(R)
>>> numpy.allclose(q, [0.82336615, -0.13610694, 0.46344705, -0.29792603])
True
>>> R = random_rotation_matrix()
>>> q = quaternion_from_matrix(R)
>>> is_same_transform(R, quaternion_matrix(q))
True
```

espresso.external.transformations.**quaternion_imag**(*quaternion*)

Return imaginary part of quaternion.

```
>>> quaternion_imag([3.0, 0.0, 1.0, 2.0])
[0.0, 1.0, 2.0]
```

espresso.external.transformations.**quaternion_inverse**(*quaternion*)

Return inverse of quaternion.

```
>>> q0 = random_quaternion()
>>> q1 = quaternion_inverse(q0)
>>> numpy.allclose(quaternion_multiply(q0, q1), [1, 0, 0, 0])
True
```

espresso.external.transformations.**quaternion_matrix**(*quaternion*)

Return homogeneous rotation matrix from quaternion.

```
>>> M = quaternion_matrix([0.99810947, 0.06146124, 0, 0])
>>> numpy.allclose(M, rotation_matrix(0.123, (1, 0, 0)))
True
>>> M = quaternion_matrix([1, 0, 0, 0])
>>> numpy.allclose(M, identity_matrix())
True
>>> M = quaternion_matrix([0, 1, 0, 0])
>>> numpy.allclose(M, numpy.diag([1, -1, -1, 1]))
True
```

espresso.external.transformations.**quaternion_multiply**(*quaternion1*, *quaternion0*)

Return multiplication of two quaternions.

```
>>> q = quaternion_multiply([4, 1, -2, 3], [8, -5, 6, 7])
>>> numpy.allclose(q, [28, -44, -14, 48])
True
```

espresso.external.transformations.**quaternion_real**(*quaternion*)

Return real part of quaternion.

```
>>> quaternion_real([3.0, 0.0, 1.0, 2.0])
3.0
```

espresso.external.transformations.**quaternion_slerp**(*quat0*, *quat1*, *fraction*, *spin=0*, *shortestpath=True*)

Return spherical linear interpolation between two quaternions.

```
>>> q0 = random_quaternion()
>>> q1 = random_quaternion()
>>> q = quaternion_slerp(q0, q1, 0.0)
>>> numpy.allclose(q, q0)
True
>>> q = quaternion_slerp(q0, q1, 1.0, 1)
>>> numpy.allclose(q, q1)
True
>>> q = quaternion_slerp(q0, q1, 0.5)
>>> angle = math.acos(numpy.dot(q0, q))
>>> numpy.allclose(2.0, math.acos(numpy.dot(q0, q1)) / angle) or          numpy.allclose(2.0, mat
True
```

espresso.external.transformations.**random_quaternion**(*rand=None*)

Return uniform random unit quaternion.

rand: array like or None Three independent random variables that are uniformly distributed between 0 and 1.

```
>>> q = random_quaternion()
>>> numpy.allclose(1.0, vector_norm(q))
True
>>> q = random_quaternion(numpy.random.random(3))
>>> len(q.shape), q.shape[0]==4
(1, True)
```

espresso.external.transformations.**random_rotation_matrix**(*rand=None*)

Return uniform random rotation matrix.

rnd: array like Three independent random variables that are uniformly distributed between 0 and 1 for each returned quaternion.

```
>>> R = random_rotation_matrix()
>>> numpy.allclose(numpy.dot(R.T, R), numpy.identity(4))
True
```

espresso.external.transformations.**random_vector**(*size*)

Return array of random doubles in the half-open interval [0.0, 1.0).

```
>>> v = random_vector(10000)
>>> numpy.all(v >= 0.0) and numpy.all(v < 1.0)
True
>>> v0 = random_vector(10)
>>> v1 = random_vector(10)
>>> numpy.any(v0 == v1)
False
```

espresso.external.transformations.**reflection_from_matrix**(*matrix*)

Return mirror plane point and normal vector from reflection matrix.

```
>>> v0 = numpy.random.random(3) - 0.5
>>> v1 = numpy.random.random(3) - 0.5
```

```
>>> M0 = reflection_matrix(v0, v1)
>>> point, normal = reflection_from_matrix(M0)
>>> M1 = reflection_matrix(point, normal)
>>> is_same_transform(M0, M1)
True
```

espresso.external.transformations.**reflection_matrix**(*point*, *normal*)

Return matrix to mirror at plane defined by point and normal vector.

```
>>> v0 = numpy.random.random(4) - 0.5
>>> v0[3] = 1.0
>>> v1 = numpy.random.random(3) - 0.5
>>> R = reflection_matrix(v0, v1)
>>> numpy.allclose(2., numpy.trace(R))
True
>>> numpy.allclose(v0, numpy.dot(R, v0))
True
>>> v2 = v0.copy()
>>> v2[:3] += v1
>>> v3 = v0.copy()
>>> v2[:3] -= v1
>>> numpy.allclose(v2, numpy.dot(R, v3))
True
```

espresso.external.transformations.**rotation_from_matrix**(*matrix*)

Return rotation angle and axis from rotation matrix.

```
>>> angle = (random.random() - 0.5) * (2*math.pi)
>>> direc = numpy.random.random(3) - 0.5
>>> point = numpy.random.random(3) - 0.5
>>> R0 = rotation_matrix(angle, direc, point)
>>> angle, direc, point = rotation_from_matrix(R0)
>>> R1 = rotation_matrix(angle, direc, point)
>>> is_same_transform(R0, R1)
True
```

espresso.external.transformations.**rotation_matrix**(*angle*, *direction*, *point=None*)

Return matrix to rotate about axis defined by point and direction.

```
>>> R = rotation_matrix(math.pi/2.0, [0, 0, 1], [1, 0, 0])
>>> numpy.allclose(numpy.dot(R, [0, 0, 0, 1]), [ 1., -1.,  0.,  1.])
True
>>> angle = (random.random() - 0.5) * (2*math.pi)
>>> direc = numpy.random.random(3) - 0.5
>>> point = numpy.random.random(3) - 0.5
>>> R0 = rotation_matrix(angle, direc, point)
>>> R1 = rotation_matrix(angle-2*math.pi, direc, point)
>>> is_same_transform(R0, R1)
True
>>> R0 = rotation_matrix(angle, direc, point)
>>> R1 = rotation_matrix(-angle, -direc, point)
>>> is_same_transform(R0, R1)
True
>>> I = numpy.identity(4, numpy.float64)
>>> numpy.allclose(I, rotation_matrix(math.pi*2, direc))
True
>>> numpy.allclose(2., numpy.trace(rotation_matrix(math.pi/2,
...                                                direc, point)))
True
```

espresso.external.transformations.**scale_from_matrix**(*matrix*)

Return scaling factor, origin and direction from scaling matrix.

```
>>> factor = random.random() * 10 - 5
>>> origin = numpy.random.random(3) - 0.5
>>> direct = numpy.random.random(3) - 0.5
>>> S0 = scale_matrix(factor, origin)
>>> factor, origin, direction = scale_from_matrix(S0)
>>> S1 = scale_matrix(factor, origin, direction)
>>> is_same_transform(S0, S1)
True
>>> S0 = scale_matrix(factor, origin, direct)
>>> factor, origin, direction = scale_from_matrix(S0)
>>> S1 = scale_matrix(factor, origin, direction)
>>> is_same_transform(S0, S1)
True
```

espresso.external.transformations.**scale_matrix**(*factor*, *origin=None*, *direction=None*)

Return matrix to scale by factor around origin in direction.

Use factor -1 for point symmetry.

```
>>> v = (numpy.random.rand(4, 5) - 0.5) * 20.0
>>> v[3] = 1.0
>>> S = scale_matrix(-1.234)
>>> numpy.allclose(numpy.dot(S, v)[:3], -1.234*v[:3])
True
>>> factor = random.random() * 10 - 5
>>> origin = numpy.random.random(3) - 0.5
>>> direct = numpy.random.random(3) - 0.5
>>> S = scale_matrix(factor, origin)
>>> S = scale_matrix(factor, origin, direct)
```

espresso.external.transformations.**shear_from_matrix**(*matrix*)

Return shear angle, direction and plane from shear matrix.

```
>>> angle = (random.random() - 0.5) * 4*math.pi
>>> direct = numpy.random.random(3) - 0.5
>>> point = numpy.random.random(3) - 0.5
>>> normal = numpy.cross(direct, numpy.random.random(3))
>>> S0 = shear_matrix(angle, direct, point, normal)
>>> angle, direct, point, normal = shear_from_matrix(S0)
>>> S1 = shear_matrix(angle, direct, point, normal)
>>> is_same_transform(S0, S1)
True
```

espresso.external.transformations.**shear_matrix**(*angle*, *direction*, *point*, *normal*)

Return matrix to shear by angle along direction vector on shear plane.

The shear plane is defined by a point and normal vector. The direction vector must be orthogonal to the plane's normal vector.

A point P is transformed by the shear matrix into P" such that the vector P-P" is parallel to the direction vector and its extent is given by the angle of P-P'-P", where P' is the orthogonal projection of P onto the shear plane.

```
>>> angle = (random.random() - 0.5) * 4*math.pi
>>> direct = numpy.random.random(3) - 0.5
>>> point = numpy.random.random(3) - 0.5
>>> normal = numpy.cross(direct, numpy.random.random(3))
>>> S = shear_matrix(angle, direct, point, normal)
```

```
>>> numpy.allclose(1.0, numpy.linalg.det(S))
True
```

espresso.external.transformations.**superimposition_matrix**(*v0*, *v1*, *scaling=False*, *usesvd=True*)

Return matrix to transform given vector set into second vector set.

v0 and v1 are shape (3, *) or (4, *) arrays of at least 3 vectors.

If usesvd is True, the weighted sum of squared deviations (RMSD) is minimized according to the algorithm by W. Kabsch [8]. Otherwise the quaternion based algorithm by B. Horn [9] is used (slower when using this Python implementation).

The returned matrix performs rotation, translation and uniform scaling (if specified).

```
>>> v0 = numpy.random.rand(3, 10)
>>> M = superimposition_matrix(v0, v0)
>>> numpy.allclose(M, numpy.identity(4))
True
>>> R = random_rotation_matrix(numpy.random.random(3))
>>> v0 = ((1,0,0), (0,1,0), (0,0,1), (1,1,1))
>>> v1 = numpy.dot(R, v0)
>>> M = superimposition_matrix(v0, v1)
>>> numpy.allclose(v1, numpy.dot(M, v0))
True
>>> v0 = (numpy.random.rand(4, 100) - 0.5) * 20.0
>>> v0[3] = 1.0
>>> v1 = numpy.dot(R, v0)
>>> M = superimposition_matrix(v0, v1)
>>> numpy.allclose(v1, numpy.dot(M, v0))
True
>>> S = scale_matrix(random.random())
>>> T = translation_matrix(numpy.random.random(3)-0.5)
>>> M = concatenate_matrices(T, R, S)
>>> v1 = numpy.dot(M, v0)
>>> v0[:3] += numpy.random.normal(0.0, 1e-9, 300).reshape(3, -1)
>>> M = superimposition_matrix(v0, v1, scaling=True)
>>> numpy.allclose(v1, numpy.dot(M, v0))
True
>>> M = superimposition_matrix(v0, v1, scaling=True, usesvd=False)
>>> numpy.allclose(v1, numpy.dot(M, v0))
True
>>> v = numpy.empty((4, 100, 3), dtype=numpy.float64)
>>> v[:, :, 0] = v0
>>> M = superimposition_matrix(v0, v1, scaling=True, usesvd=False)
>>> numpy.allclose(v1, numpy.dot(M, v[:, :, 0]))
True
```

espresso.external.transformations.**translation_from_matrix**(*matrix*)

Return translation vector from translation matrix.

```
>>> v0 = numpy.random.random(3) - 0.5
>>> v1 = translation_from_matrix(translation_matrix(v0))
>>> numpy.allclose(v0, v1)
True
```

espresso.external.transformations.**translation_matrix**(*direction*)

Return matrix to translate by direction vector.

```
>>> v = numpy.random.random(3) - 0.5
>>> numpy.allclose(v, translation_matrix(v)[:3, 3])
True
```

espresso.external.transformations.**unit_vector**(*data*, *axis=None*, *out=None*)

Return ndarray normalized by length, i.e. eucledian norm, along axis.

```
>>> v0 = numpy.random.random(3)
>>> v1 = unit_vector(v0)
>>> numpy.allclose(v1, v0 / numpy.linalg.norm(v0))
True
>>> v0 = numpy.random.rand(5, 4, 3)
>>> v1 = unit_vector(v0, axis=-1)
>>> v2 = v0 / numpy.expand_dims(numpy.sqrt(numpy.sum(v0*v0, axis=2)), 2)
>>> numpy.allclose(v1, v2)
True
>>> v1 = unit_vector(v0, axis=1)
>>> v2 = v0 / numpy.expand_dims(numpy.sqrt(numpy.sum(v0*v0, axis=1)), 1)
>>> numpy.allclose(v1, v2)
True
>>> v1 = numpy.empty((5, 4, 3), dtype=numpy.float64)
>>> unit_vector(v0, axis=1, out=v1)
>>> numpy.allclose(v1, v2)
True
>>> list(unit_vector([]))
[]
>>> list(unit_vector([1.0]))
[1.0]
```

espresso.external.transformations.**vector_norm**(*data*, *axis=None*, *out=None*)

Return length, i.e. eucledian norm, of ndarray along axis.

```
>>> v = numpy.random.random(3)
>>> n = vector_norm(v)
>>> numpy.allclose(n, numpy.linalg.norm(v))
True
>>> v = numpy.random.rand(6, 5, 3)
>>> n = vector_norm(v, axis=-1)
>>> numpy.allclose(n, numpy.sqrt(numpy.sum(v*v, axis=2)))
True
>>> n = vector_norm(v, axis=1)
>>> numpy.allclose(n, numpy.sqrt(numpy.sum(v*v, axis=1)))
True
>>> v = numpy.random.rand(5, 4, 3)
>>> n = numpy.empty((5, 3), dtype=numpy.float64)
>>> vector_norm(v, axis=1, out=n)
>>> numpy.allclose(n, numpy.sqrt(numpy.sum(v*v, axis=1)))
True
>>> vector_norm([])
0.0
>>> vector_norm([1.0])
1.0
```

espresso.external.transformations.**vector_product**(*v0*, *v1*, *axis=0*)

Return vector perpendicular to vectors.

```
>>> v = vector_product([2, 0, 0], [0, 3, 0])
>>> numpy.allclose(v, [0, 0, 6])
True
```

```
>>> v0 = [[2, 0, 0, 2], [0, 2, 0, 2], [0, 0, 2, 2]]
>>> v1 = [[3], [0], [0]]
>>> v = vector_product(v0, v1)
>>> numpy.allclose(v, [[0, 0, 0, 0], [0, 0, 6, 6], [0, -6, 0, -6]])
True
>>> v0 = [[2, 0, 0], [2, 0, 0], [0, 2, 0], [2, 0, 0]]
>>> v1 = [[0, 3, 0], [0, 0, 3], [0, 0, 3], [3, 3, 3]]
>>> v = vector_product(v0, v1, axis=1)
>>> numpy.allclose(v, [[0, 0, 6], [0, -6, 0], [6, 0, 0], [0, -6, 6]])
True
```

5.19 integrator

5.19.1 AdResS - Object

The AdResS object is an extension to the integrator. It makes sure that the integrator also processes the atomistic particles and not only the CG particles. Hence, this object is of course only used when performing AdResS or H-AdResS simulations.

In detail the AdResS extension makes sure:

- that also the forces on the atomistic particles are initialized and set to by Adress::initForces
- that also the atomistic particles are integrated and propagated by Adress::integrate1 and Adress::integrate2

Example - how to turn on the AdResS integrator extension:

```
>>> adress      = espresso.integrator.Adress(system)
>>> integrator.addExtension(adress)
```

class espresso.integrator.Adress.**AdressLocal** (*system*)
The (local) AdResS

5.19.2 BerendsenBarostat - Berendsen barostat Object

This is the Berendsen barostat implementation according to the original paper [Berendsen84]. If Berendsen barostat is defined (as a property of integrator) then at the each run the system size and the particle coordinates will be scaled by scaling parameter μ according to the formula:

$$\mu = [1 - \Delta t/\tau(P_0 - P)]^{1/3}$$

where Δt - integration timestep, τ - time parameter (coupling parameter), P_0 - external pressure and P - instantaneous pressure.

Example:

```
>>> berendsenP = espresso.integrator.BerendsenBarostat(system)
>>> berendsenP.tau = 0.1
>>> berendsenP.pressure = 1.0
>>> integrator.addExtension(berendsenP)
```

!IMPORTANT In order to run *npt* simulation one should separately define thermostat as well (e.g. BerendsenThermostat).

Definition:

In order to define the Berendsen barostat

```
>>> berendsenP = espresso.integrator.BerendsenBarostat(system)
```

one should have the System defined.

Properties:

- *berendsenP.tau*

  The property 'tau' defines the time parameter τ.

- *berendsenP.pressure*

  The property 'pressure' defines the external pressure P_0.

Setting the integration property:

```
>>> integrator.addExtension(berendsenP)
```

It will define Berendsen barostat as a property of integrator.

One more example:

```
>>> berendsen_barostat = espresso.integrator.BerendsenBarostat(system)
>>> berendsen_barostat.tau = 10.0
>>> berendsen_barostat.pressure = 3.5
>>> integrator.addExtension(berendsen_barostat)
```

Canceling the barostat:

If one do not need the pressure regulation in system anymore or need to switch the ensamble or whatever :)

```
>>> # define barostat with parameters
>>> berendsen = espresso.integrator.BerendsenBarostat(system)
>>> berendsen.tau = 0.8
>>> berendsen.pressure = 15.0
>>> integrator.addExtension(berendsen)
>>> ...
>>> # some runs
>>> ...
>>> # disconnect Berendsen barostat
>>> berendsen.disconnect()
>>> # the next runs will not include the system size and particle coordinates scaling
```

Connecting the barostat back after the disconnection

```
>>> berendsen.connect()
```

References:

5.19.3 BerendsenBarostatAnisotropic - Berendsen barostat Object

#TODO fix these comments This is the Berendsen barostat implementation according to the original paper [Berendsen84]. If Berendsen barostat is defined (as a property of integrator) then at the each run the system size and the

particle coordinates will be scaled by scaling parameter μ according to the formula:

$$\mu = [1 - \Delta t/\tau(P_0 - P)]^{1/3}$$

where Δt - integration timestep, τ - time parameter (coupling parameter), P_0 - external pressure and P - instantaneous pressure.

Example:

```
>>> berendsenP = espresso.integrator.BerendsenBarostatAnisotropic(system)
>>> berendsenP.tau = 0.1
>>> berendsenP.pressure = 1.0
>>> integrator.addExtension(berendsenP)
```

!IMPORTANT In order to run *npt* simulation one should separately define thermostat as well (e.g. BerendsenThermostat).

Definition:

In order to define the Berendsen barostat

```
>>> berendsenP = espresso.integrator.BerendsenBarostatAnisotropic(system)
```

one should have the System defined.

Properties:

- *berendsenP.tau*

  The property 'tau' defines the time parameter τ.

- *berendsenP.pressure*

  The property 'pressure' defines the external pressure P_0.

Setting the integration property:

```
>>> integrator.addExtension(berendsenP)
```

It will define Berendsen barostat as a property of integrator.

One more example:

```
>>> berendsen_barostat = espresso.integrator.BerendsenBarostatAnisotropic(system)
>>> berendsen_barostat.tau = 10.0
>>> berendsen_barostat.pressure = 3.5
>>> integrator.addExtension(berendsen_barostat)
```

Canceling the barostat:

If one do not need the pressure regulation in system anymore or need to switch the ensamble or whatever :)

```
>>> # define barostat with parameters
>>> berendsen = espresso.integrator.BerendsenBarostatAnisotropic(system)
>>> berendsen.tau = 0.8
>>> berendsen.pressure = 15.0
>>> integrator.addExtension(berendsen)
>>> ...
>>> # some runs
>>> ...
>>> # disconnect Berendsen barostat
```

```
>>> berendsen.disconnect()
>>> # the next runs will not include the system size and particle coordinates scaling
```

Connecting the barostat back after the disconnection

```
>>> berendsen.connect()
```

5.19.4 BerendsenThermostat - Berendsen thermostat Object

This is the Berendsen thermostat implementation according to the original paper [Berendsen84]. If Berendsen thermostat is defined (as a property of integrator) then at the each run the system size and the particle coordinates will be scaled by scaling parameter λ according to the formula:

$$\lambda = [1 + \Delta t/\tau_T(T_0/T - 1)]^{1/2}$$

where Δt - integration timestep, τ_T - time parameter (coupling parameter), T_0 - external temperature and T - instantaneous temperature.

Example:

```
>>> berendsenT = espresso.integrator.BerendsenThermostat(system)
>>> berendsenT.tau = 1.0
>>> berendsenT.temperature = 1.0
>>> integrator.addExtension(berendsenT)
```

Definition:

In order to define the Berendsen thermostat

```
>>> berendsenT = espresso.integrator.BerendsenThermostat(system)
```

one should have the System defined.

Properties:

- *berendsenT.tau*

  The property 'tau' defines the time parameter τ_T.

- *berendsenT.temperature*

  The property 'temperature' defines the external temperature T_0.

Setting the integration property:

```
>>> integrator.addExtension(berendsenT)
```

It will define Berendsen thermostat as a property of integrator.

One more example:

```
>>> berendsen_thermostat = espresso.integrator.BerendsenThermostat(system)
>>> berendsen_thermostat.tau = 0.1
>>> berendsen_thermostat.temperature = 3.2
>>> integrator.addExtension(berendsen_thermostat)
```

Canceling the thermostat:

```
>>> # define thermostat with parameters
>>> berendsen = espresso.integrator.BerendsenThermostat(system)
>>> berendsen.tau = 2.0
>>> berendsen.temperature = 5.0
>>> integrator.addExtension(berendsen)
>>> ...
>>> # some runs
>>> ...
>>> # disconnect Berendsen thermostat
>>> berendsen.disconnect()
```

Connecting the thermostat back after the disconnection

```
>>> berendsen.connect()
```

5.19.5 CapForce - Integrator Extension

This class can be used to forcecap all particles or a group of particles. Force capping means that the force vector of a particle is rescaled so that the length of the force vector is <= capforce

Example Usage:

```
>>> capforce     = espresso.integrator.CapForce(system, 1000.0)
>>> integrator.addExtension(capForce)
```

CapForce can also be used to forcecap only a group of particles:

```
>>> particle_group = [45, 67, 89, 103]
>>> capforce       = espresso.integrator.CapForce(system, 1000.0, particle_group)
>>> integrator.addExtension(capForce)
```

class espresso.integrator.CapForce.**CapForceLocal**(*system*, *capForce*, *particleGroup=None*)
The (local) force capping part.

5.19.6 espresso.integrator.DPDThermostat

class espresso.integrator.DPDThermostat.**DPDThermostatLocal**(*system*, *vl*)
The (local) Velocity Verlet Integrator.

5.19.7 ExtAnalyze - Integrator Extension

This class can be used to execute nearly all analysis objects within the main integration loop which allows to automatically accumulate time averages (with standard deviation error bars).

Example Usage:

```
>>> pt           = espresso.analysis.PressureTensor(system)
>>> extension_pt = espresso.integrator.ExtAnalyze(pt , interval=100)
>>> integrator.addExtension(extension_pt)
>>> integrator.run(10000)
>>>
>>> pt_ave = pt.getAverageValue()
```

```
>>> print "average Pressure Tensor = ", pt_ave[:6]
>>> print "          std deviation = ", pt_ave[6:]
>>> print "number of measurements  = ", pt.getNumberOfMeasurements()
```

class espresso.integrator.ExtAnalyze.**ExtAnalyzeLocal**(*action_obj*, *interval=1*)
The (local) extension analyze.

5.19.8 espresso.integrator.ExtForce

class espresso.integrator.ExtForce.**ExtForceLocal**(*system*, *extForce*, *particleGroup=None*)
The (local) external force part.

5.19.9 espresso.integrator.Extension

class espresso.integrator.Extension.**ExtensionLocal**
The (local) Extension abstract base class.

5.19.10 espresso.integrator.FixPositions

class espresso.integrator.FixPositions.**FixPositionsLocal**(*system*, *particleGroup*, *fixMask*)
The (local) Fix Positions part.

5.19.11 espresso.integrator.FreeEnergyCompensation

class espresso.integrator.FreeEnergyCompensation.**FreeEnergyCompensationLocal**(*system*, *center=[]*)
The (local) Velocity Verlet Integrator.

addForce(*itype*, *filename*, *type*)
Each processor takes the broadcasted interpolation type, filename and particle type

5.19.12 espresso.integrator.Isokinetic

class espresso.integrator.Isokinetic.**IsokineticLocal**(*system*)
The (local) Isokinetic Thermostat.

5.19.13 LBInit - abstract base class for LatticeBoltzmann initialization

This abstract base class provides the interface and some basic functionality for classes that (re)initialize populations and handle external forces

It provides the following methods:

- **createDenVel(rho0,u0)** sets initial density and velocity
- **setForce()** sets external force to a specific values
- **setForce()** adds a specific value to the existing forces

class espresso.integrator.LBInit.**LBInitLocal**

Abstract local base class for LBInit.

addForce (*force*)

addForce adds an external force onto the system. All existing forces will be preserved! A user might use it for a superposition of forces desired in a specific application. A constant (gravity-like) force coded by LBInitConstForce and a sin-wave-like f_z force component as a function of x provided by LBInitPeriodicForce.

Example:

```
>>> lbforce = espresso.integrator.LBInitConstForce(system,lb)
>>> lbforce.addForce(Real3D(0.,0.,0.0005))
>>> # a vector adds the external body force directly in lb-units
```

Example:

```
>>> lbforce = espresso.integrator.LBInitPeriodicForce(system,lb)
>>> lbforce.addForce(Real3D(0.,0.,0.0005))
>>> # a vector adds the external body force with a Real3D amplitude
```

createDenVel (*rho0*, *u0*)

createDenVel helps to create initial populations with desired density and velocity. By default either a uniform conformation is created by function LBInitPopUniform or a conformation with a constant density and sin-wave-like v_z component as a function of x by function LBInitPopWave.

Example:

```
>>> initPop = espresso.integrator.LBInitPopUniform(system,lb)
>>> initPop.createDenVel(1.0, Real3D(0.,0.,0.0))
>>> # first number is the density, second number is a vector of velocity
```

Example:

```
>>> initPop = espresso.integrator.LBInitPopWave(system,lb)
>>> initPop.createDenVel(1.0, Real3D(0.,0.,0.0005))
>>> # the Real3D vector in this case includes amplitudes of the velocities
```

setForce (*force*)

setForce sets an external force onto the system. It is either a constant body force (gravity-like) coded by LBInitConstForce or a sin-wave-like f_z force component as a function of x provided by LBInitPeriodicForce.

Example:

```
>>> lbforce = espresso.integrator.LBInitConstForce(system,lb)
>>> lbforce.setForce(Real3D(0.,0.,0.0005))
>>> # a vector sets the external body force directly in lb-units
```

Example:

```
>>> lbforce = espresso.integrator.LBInitPeriodicForce(system,lb)
>>> lbforce.setForce(Real3D(0.,0.,0.0005))
>>> # a vector sets the external body force amplitude
```

5.19.14 LBInitConstForce - handles external constant (gravity-like) forces

This class sets and adds an external constant (gravity-like) forces to a liquid

class espresso.integrator.LBInitConstForce.**LBInitConstForceLocal**(*system*, *latticeboltzmann*)

The (local) compute of LBInitConstForce.

5.19.15 LBInitPeriodicForce - handles external periodic forces

This class sets and adds an external periodic forces to a liquid

class espresso.integrator.LBInitPeriodicForce.**LBInitPeriodicForceLocal**(*system*, *latticeboltzmann*)

The (local) compute of LBInitPeriodicForce.

5.19.16 LBInitPopUniform - creates initial populations with uniform density and velocity

This class creates initial populations with uniform density and velocity

class espresso.integrator.LBInitPopUniform.**LBInitPopUniformLocal**(*system*, *latticeboltzmann*)

The (local) compute of LBInitPopUniform.

5.19.17 LBInitPopWave - creates initial populations with uniform density and harmonic velocity

This class creates initial populations with uniform density and harmonic velocity: v_x = 0, v_y = 0, v_z = Amp * sin (2 * pi * i / N_x)

This may be used to test the system: total moment is zero and the liquid tends to equilibrium, i.e. relaxes to uniform zero velocity.

class espresso.integrator.LBInitPopWave.**LBInitPopWaveLocal**(*system*, *latticeboltzmann*)

The (local) compute of LBInitPopWave.

5.19.18 LangevinBarostat - Langevin-Hoover barostat Object

This is the barostat implementation to perform Langevin dynamics in a Hoover style extended system according to the paper [Quigley04]. It includes corrections of Hoover approach which were introduced by Martyna et al [Martyna94]. If LangevinBarostat is defined (as a property of integrator) the integration equations will be modified. The volume of system V is introduced as a dynamical variable:

$$\dot{\boldsymbol{r}}_i = \frac{\boldsymbol{p}_i}{m_i} + \frac{p_\epsilon}{W}\boldsymbol{r}_i$$

$$\dot{\boldsymbol{p}}_i = -\bigtriangledown_{\boldsymbol{r}_i}\Phi - (1+\frac{n}{N_f})\frac{p_\epsilon}{W}\boldsymbol{p}_i - \gamma\boldsymbol{p}_i + \boldsymbol{R}_i$$

$$\dot{V} = dVp_\epsilon/W$$

$$\dot{p}_\epsilon = nV(X - P_{ext}) + \frac{n}{N_f}\sum_{i=1}^{N}\frac{\boldsymbol{p}_i^2}{m_i} - \gamma_p p_\epsilon + R_p$$

where volume has a fictitious mass W and associated momentum p_ϵ, γ_p - friction coefficient, P_{ext} - external pressure and X - instantaneous pressure without white noise contribution from thermostat, n - dimension, N_f - degrees of freedom (if there are no constrains and N is the number of particles in system $N_f = nN$). R_p - values which are drawn from Gaussian distribution of zero mean and unit variance scaled by

$$\sqrt{\frac{2k_BTW\gamma_p}{\Delta t}}$$

!IMPORTANT Terms $-\gamma\boldsymbol{p}_i + \boldsymbol{R}_i$ correspond to the termostat. They are not included here and will not be calculated if the Langevin Thermostat is not defined.

Example:

```
>>> rng = espresso.esutil.RNG()
>>> langevinP = espresso.integrator.LangevinBarostat(system, rng, desiredTemperature)
>>> langevinP.gammaP = 0.05
>>> langevinP.pressure = 1.0
>>> langevinP.mass = pow(10.0, 4)
>>> integrator.addExtension(langevinP)
```

!IMPORTANT This barostat is supposed to be run in a couple with thermostat in order to simulate the *npt* ensamble, because the term R_p needs the temperature as a parameter.

Definition:

In order to define the Langevin-Hoover barostat

```
>>> langevinP = espresso.integrator.LangevinBarostat(system, rng, desiredTemperature)
```

one should have the System and RNG defined and know the desired temperature.

Properties:

- *langevinP.gammaP*

  The property 'gammaP' defines the friction coefficient γ_p.

- *langevinP.pressure*

  The property 'pressure' defines the external pressure P_{ext}.

- *langevinP.mass*

  The property 'mass' defines the fictitious mass W.

Methods:

- *setMassByFrequency(frequency)*

  Set the proper *langevinP.mass* using expression $W = dNk_bT/\omega_b^2$, where frequency, ω_b, is the frequency of required volume fluctuations. The value of ω_b should be less then the lowest frequency which appears in the NVT temperature spectrum [Quigley04] in order to match the canonical distribution. d - dimensions, N - number of particles, k_b - Boltzmann constant, T - desired temperature.

NOTE The *langevinP.mass* can be set both directly and using the (*setMassByFrequency(frequency)*)

Adding to the integration:

```
>>> integrator.addExtension(langevinP)
```

It will define Langevin-Hoover barostat as a property of integrator.

One more example:

```
>>> rngBaro = espresso.esutil.RNG()
>>> lP = espresso.integrator.LangevinBarostat(system, rngBaro, desiredTemperature)
>>> lP.gammaP = .5
>>> lP.pressure = 1.0
>>> lP.mass = pow(10.0, 5)
>>> integrator.addExtension(lP)
```

Canceling the barostat:

If one do not need the pressure regulation in system anymore or need to switch the ensamble or whatever :)

```
>>> # define barostat with parameters
>>> rngBaro = espresso.esutil.RNG()
>>> lP = espresso.integrator.LangevinBarostat(system, rngBaro, desiredTemperature)
>>> lP.gammaP = .5
>>> lP.pressure = 1.0
>>> lP.mass = pow(10.0, 5)
>>> integrator.langevinBarostat = lP
>>> ...
>>> # some runs
>>> ...
>>> # disconnect barostat
>>> langevinBarostat.disconnect()
>>> # the next runs will not include the modification of integration equations
```

Connecting the barostat back after the disconnection

```
>>> langevinBarostat.connect()
```

References:

5.19.19 espresso.integrator.LangevinThermostat

class espresso.integrator.LangevinThermostat.**LangevinThermostatLocal**(*system*)
The (local) Velocity Verlet Integrator.

5.19.20 espresso.integrator.LangevinThermostat1D

class espresso.integrator.LangevinThermostat1D.**LangevinThermostat1DLocal**(*system*)
The (local) Langevin Thermostat (1D).

5.19.21 espresso.integrator.LatticeBoltzmann

class espresso.integrator.LatticeBoltzmann.**LatticeBoltzmannLocal**(*system*, *Ni*, *a=1.0*, *tau=1.0*, *numDims=3*, *numVels=19*)
The (local) Lattice Boltzmann part.

Creates a simulation box with a specified dimensions and allocates the necessary memory for a lattice Boltzmann simulation. Default values of the parameters include: spacing of the latticea = 1., time spacing = 1., number of dimensions = 3, number of velocities at the lattice site = 19.

Example

```
>>> lb = espresso.integrator.LatticeBoltzmann(system, Ni=Int3D(20, 20, 20))
>>> # will create a cubic lattice box of 20 sites with default spacing parameters in D3Q19 model
```

5.19.22 espresso.integrator.MDIntegrator

class espresso.integrator.MDIntegrator.**MDIntegrator**
Abstract base class for molecular dynamics integrator.

class espresso.integrator.MDIntegrator.**MDIntegratorLocal**
Abstract local base class for molecular dynamics integrator.

5.19.23 espresso.integrator.StochasticVelocityRescaling

class espresso.integrator.StochasticVelocityRescaling.**StochasticVelocityRescalingLocal**(*system*)
The (local) StochasticVelocityRescaling Thermostat.

5.19.24 espresso.integrator.TDforce

class espresso.integrator.TDforce.**TDforceLocal**(*system*, *center*=[ ])
The (local) Velocity Verlet Integrator.

addForce(*itype*, *filename*, *type*)
Each processor takes the broadcasted interpolation type, filename and particle type

5.19.25 espresso.integrator.VelocityVerlet

class espresso.integrator.VelocityVerlet.**VelocityVerletLocal**(*system*)
The (local) Velocity Verlet Integrator.

5.19.26 espresso.integrator.VelocityVerletOnGroup

class espresso.integrator.VelocityVerletOnGroup.**VelocityVerletOnGroupLocal**(*system*, *group*)
The (local) Velocity Verlet Integrator.

5.19.27 espresso.integrator.VelocityVerletOnRadius

class espresso.integrator.VelocityVerletOnRadius.**VelocityVerletOnRadiusLocal**(*system*, *dampingmass*)
The (local) VelocityVerletOnRadius.

5.20 interaction

5.20.1 espresso.interaction.AngularCosineSquared

class espresso.interaction.AngularCosineSquared.**AngularCosineSquared**
The AngularCosineSquared potential.

class espresso.interaction.AngularCosineSquared.**AngularCosineSquaredLocal**(*K=1.0*, *theta0=0.0*)
The (local) AngularCosineSquared potential.

class espresso.interaction.AngularCosineSquared.**FixedTripleListAngularCosineSquaredLocal**(*system*, *vl*, *potential*)
The (local) AngularCosineSquared interaction using FixedTriple lists.

5.20.2 espresso.interaction.AngularHarmonic

class espresso.interaction.AngularHarmonic.**AngularHarmonic**
The AngularHarmonic potential.

class espresso.interaction.AngularHarmonic.**AngularHarmonicLocal**(*K=1.0*, *theta0=0.0*)
The (local) AngularHarmonic potential.

class espresso.interaction.AngularHarmonic.**FixedTripleListAngularHarmonicLocal**(*system*, *vl*, *potential*)
The (local) AngularHarmonic interaction using FixedTriple lists.

5.20.3 espresso.interaction.AngularPotential

5.20.4 espresso.interaction.AngularUniqueCosineSquared

class espresso.interaction.AngularUniqueCosineSquared.**AngularUniqueCosineSquared**
The AngularUniqueCosineSquared potential.

class espresso.interaction.AngularUniqueCosineSquared.**AngularUniqueCosineSquaredLocal**(*K=1.0*)
The (local) AngularUniqueCosineSquared potential.

class espresso.interaction.AngularUniqueCosineSquared.**FixedTripleAngleListAngularUniqueCosine**

The (local) AngularUniqueCosineSquared interaction using FixedTripleAngle lists.

5.20.5 espresso.interaction.AngularUniqueHarmonic

class espresso.interaction.AngularUniqueHarmonic.**AngularUniqueHarmonic**
The AngularUniqueHarmonic potential.

class espresso.interaction.AngularUniqueHarmonic.**AngularUniqueHarmonicLocal**(*K=1.0*)
The (local) AngularUniqueHarmonic potential.

class espresso.interaction.AngularUniqueHarmonic.**FixedTripleAngleListAngularUniqueHarmonicLoc**

The (local) AngularUniqueHarmonic interaction using FixedTriple lists.

5.20.6 espresso.interaction.AngularUniquePotential

5.20.7 espresso.interaction.Cosine

class espresso.interaction.Cosine.**Cosine**
The Cosine potential.

class espresso.interaction.Cosine.**CosineLocal**(*K=1.0*, *theta0=0.0*)
The (local) Cosine potential.

class espresso.interaction.Cosine.**FixedTripleListCosineLocal**(*system*, *vl*, *potential*)
The (local) Cosine interaction using FixedTriple lists.

5.20.8 CoulombKSpaceEwald - Coulomb potential and interaction Objects (*K* space part)

This is the *K* space part of potential of Coulomb long range interaction according to the Ewald summation technique. Good explanation of Ewald summation could be found here [Allen89], [Deserno98].

Example:

```
>>> ewaldK_pot = espresso.interaction.CoulombKSpaceEwald(system, coulomb_prefactor, alpha, kspacecut
>>> ewaldK_int = espresso.interaction.CellListCoulombKSpaceEwald(system.storage, ewaldK_pot)
>>> system.addInteraction(ewaldK_int)
```

!IMPORTANT Coulomb interaction needs *R* space part as well CoulombRSpace.

Definition:

It provides potential object *CoulombKSpaceEwald* and interaction object *CellListCoulombKSpaceEwald* based on all particles list.

The *potential* is based on the system information (System) and parameters: Coulomb prefactor (coulomb_prefactor), Ewald parameter (alpha), and the cutoff in K space (kspacecutoff).

```
>>> ewaldK_pot = espresso.interaction.CoulombKSpaceEwald(system, coulomb_prefactor, alpha, kspac
```

Potential Properties:

- *ewaldK_pot.prefactor*

  The property 'prefactor' defines the Coulomb prefactor.

- *ewaldK_pot.alpha*

  The property 'alpha' defines the Ewald parameter α.

- *ewaldK_pot.kmax*

  The property 'kmax' defines the cutoff in *K* space.

The *interaction* is based on the all particles list. It needs the information from Storage and *K* space part of potential.

```
>>> ewaldK_int = espresso.interaction.CellListCoulombKSpaceEwald(system.storage, ewaldK_pot)
```

Interaction Methods:

- *getPotential()*

  Access to the local potential.

Adding the interaction to the system:

```
>>> system.addInteraction(ewaldK_int)
```

References:

5.20.9 CoulombKSpaceP3M - Coulomb potential and interaction Objects (*K* space part)

This is the *K* space part of potential of Coulomb long range interaction according to the P3M summation technique. Good explanation of P3M summation could be found here [Allen89], [Deserno98].

Example:

```
>>> ewaldK_pot = espresso.interaction.CoulombKSpaceP3M(system, coulomb_prefactor, alpha, kspacecutof
>>> ewaldK_int = espresso.interaction.CellListCoulombKSpaceP3M(system.storage, ewaldK_pot)
>>> system.addInteraction(ewaldK_int)
```

!IMPORTANT Coulomb interaction needs *R* space part as well CoulombRSpace.

Definition:

It provides potential object *CoulombKSpaceP3M* and interaction object *CellListCoulombKSpaceP3M* based on all particles list.

The *potential* is based on the system information (System) and parameters: Coulomb prefactor (coulomb_prefactor), P3M parameter (alpha), and the cutoff in K space (kspacecutoff).

```
>>> ewaldK_pot = espresso.interaction.CoulombKSpaceP3M(system, coulomb_prefactor, alpha, kspacec
```

Potential Properties:

- *ewaldK_pot.prefactor*

  The property 'prefactor' defines the Coulomb prefactor.

- *ewaldK_pot.alpha*

  The property 'alpha' defines the P3M parameter α.

- *ewaldK_pot.kmax*

  The property 'kmax' defines the cutoff in *K* space.

The *interaction* is based on the all particles list. It needs the information from Storage and *K* space part of potential.

```
>>> ewaldK_int = espresso.interaction.CellListCoulombKSpaceP3M(system.storage, ewaldK_pot)
```

Interaction Methods:

- *getPotential()*

  Access to the local potential.

Adding the interaction to the system:

```
>>> system.addInteraction(ewaldK_int)
```

5.20.10 CoulombRSpace - Coulomb potential and interaction Objects (*R* space part)

This is the *R* space part of potential of Coulomb long range interaction according to the Ewald summation technique. Good explanation of Ewald summation could be found here [Allen89], [Deserno98].

Example:

```
>>> vl = espresso.VerletList(system, rspacecutoff+skin)
>>> coulombR_pot = espresso.interaction.CoulombRSpace(coulomb_prefactor, alpha, rspacecutoff)
>>> coulombR_int = espresso.interaction.VerletListCoulombRSpace(vl)
>>> coulombR_int.setPotential(type1=0, type2=0, potential = coulombR_pot)
>>> system.addInteraction(coulombR_int)
```

!IMPORTANT Coulomb interaction needs k-space part as well EwaldKSpace.

Definition:

It provides potential object *CoulombRSpace* and interaction object *VerletListCoulombRSpace*

The *potential* is based on parameters: Coulomb prefactor (coulomb_prefactor), Ewald parameter (alpha), and the cutoff in R space (rspacecutoff).

```
>>> coulombR_pot = espresso.interaction.CoulombRSpace(coulomb_prefactor, alpha, rspacecutoff)
```

Potential Properties:

- *coulombR_pot.prefactor*

  The property 'prefactor' defines the Coulomb prefactor.

- *coulombR_pot.alpha*

  The property 'alpha' defines the Ewald parameter α.

- *coulombR_pot.cutoff*

  The property 'cutoff' defines the cutoff in R space.

The *interaction* is based on the Verlet list (VerletList)

```
>>> vl = espresso.VerletList(system, rspacecutoff+skin)
>>> coulombR_int = espresso.interaction.VerletListCoulombRSpace(vl)
```

It should include at least one potential

```
>>> coulombR_int.setPotential(type1=0, type2=0, potential = coulombR_pot)
```

Interaction Methods:

- *setPotential(type1, type2, potential)*

  This method sets the *potential* for the particles of *type1* and *type2*. It could be a bunch of potentials for the different particle types.

- *getVerletListLocal()*

  Access to the local Verlet list.

Adding the interaction to the system:

```
>>> system.addInteraction(coulombR_int)
```

5.20.11 espresso.interaction.CoulombTruncated

class espresso.interaction.CoulombTruncated.**CellListCoulombTruncatedLocal**(*stor*)
The (local) CoulombTruncated interaction using cell lists.

class espresso.interaction.CoulombTruncated.**CoulombTruncated**
The CoulombTruncated potential.

class espresso.interaction.CoulombTruncated.**CoulombTruncatedLocal**(*qq=1.0*, *cutoff=inf*, *shift='auto'*)
The (local) CoulombTruncated potential.

class espresso.interaction.CoulombTruncated.**FixedPairListCoulombTruncatedLocal**(*system*, *vl*, *potential*)
The (local) CoulombTruncated interaction using FixedPair lists.

class espresso.interaction.CoulombTruncated.**VerletListCoulombTruncatedLocal**(*vl*)
The (local) CoulombTruncated interaction using Verlet lists.

5.20.12 espresso.interaction.DihedralHarmonicCos

class espresso.interaction.DihedralHarmonicCos.**DihedralHarmonicCos**
The DihedralHarmonicCos potential.

class espresso.interaction.DihedralHarmonicCos.**DihedralHarmonicCosLocal**(*K=0.0*, *phi0=0.0*)
The (local) DihedralHarmonicCos potential.

class espresso.interaction.DihedralHarmonicCos.**FixedQuadrupleListDihedralHarmonicCosLocal**(*syst*, *fql*, *potential*)
The (local) DihedralHarmonicCos interaction using FixedQuadruple lists.

5.20.13 espresso.interaction.DihedralHarmonicUniqueCos

class espresso.interaction.DihedralHarmonicUniqueCos.**DihedralHarmonicUniqueCos**
The DihedralHarmonicUniqueCos potential.

class espresso.interaction.DihedralHarmonicUniqueCos.**DihedralHarmonicUniqueCosLocal**(*K=0.0*)
The (local) DihedralHarmonicUniqueCos potential.

class espresso.interaction.DihedralHarmonicUniqueCos.**FixedQuadrupleAngleListDihedralHarmonicU**

The (local) DihedralHarmonicUniqueCos interaction using FixedQuadruple lists.

5.20.14 espresso.interaction.DihedralPotential

5.20.15 espresso.interaction.DihedralUniquePotential

5.20.16 espresso.interaction.FENE

class espresso.interaction.FENE.**FENE**
The FENE potential.

class espresso.interaction.FENE.**FENELocal**(*K=1.0*, *r0=0.0*, *rMax=1.0*, *cutoff=inf*, *shift=0.0*)
The (local) FENE potential.

class espresso.interaction.FENE.**FixedPairListFENELocal**(*system*, *vl*, *potential*)
The (local) FENE interaction using FixedPair lists.

5.20.17 espresso.interaction.FENECapped

class espresso.interaction.FENECapped.**FENECapped**
The FENECapped potential.

class espresso.interaction.FENECapped.**FENECappedLocal**(*K=1.0*, *r0=0.0*, *rMax=1.0*, *cutoff=inf*, *caprad=1.0*, *shift=0.0*)
The (local) FENECapped potential.

class espresso.interaction.FENECapped.**FixedPairListFENECappedLocal**(*system*, *vl*, *potential*)
The (local) FENECapped interaction using FixedPair lists.

5.20.18 GravityTruncated

This is an implementation of a truncated (cutoff) Gravity Potential

5.20.19 espresso.interaction.Harmonic

class espresso.interaction.Harmonic.**FixedPairListHarmonicLocal**(*system*, *vl*, *potential*)
The (local) Harmonic interaction using FixedPair lists.

class espresso.interaction.Harmonic.**Harmonic**
The Harmonic potential.

class espresso.interaction.Harmonic.**HarmonicLocal**(*K=1.0*, *r0=0.0*, *cutoff=inf*, *shift=0.0*)
The (local) Harmonic potential.

5.20.20 espresso.interaction.HarmonicUnique

class espresso.interaction.HarmonicUnique.**FixedPairDistListHarmonicUniqueLocal**(*system*, *fpl*, *potential*)
The (local) HarmonicUnique interaction using FixedPair lists.

class espresso.interaction.HarmonicUnique.**HarmonicUnique**
The HarmonicUnique potential.

class espresso.interaction.HarmonicUnique.**HarmonicUniqueLocal**(*K=1.0*)
The (local) HarmonicUnique potential.

5.20.21 espresso.interaction.Interaction

class espresso.interaction.Interaction.**Interaction**
Abstract base class for interaction.

class espresso.interaction.Interaction.**InteractionLocal**
Abstract local base class for interactions.

5.20.22 espresso.interaction.LJcos

class espresso.interaction.LJcos.**CellListLJcosLocal**(*stor*)
The (local) Lennard Jones interaction using cell lists.

class espresso.interaction.LJcos.**FixedPairListLJcosLocal**(*system*, *vl*, *potential*)
The (local) Lennard-Jones interaction using FixedPair lists.

class espresso.interaction.LJcos.**LJcos**
The Lennard-Jones potential.

class espresso.interaction.LJcos.**LJcosLocal**(*phi=1.0*)
The (local) Lennard-Jones potential.

class espresso.interaction.LJcos.**VerletListAdressLJcosLocal**(*vl*, *fixedtupleList*)
The (local) Lennard Jones interaction using Verlet lists.

class espresso.interaction.LJcos.**VerletListHadressLJcosLocal**(*vl*, *fixedtupleList*, *KTI=False*)
The (local) Lennard Jones interaction using Verlet lists.

class espresso.interaction.LJcos.**VerletListLJcosLocal**(*vl*)
The (local) Lennard Jones interaction using Verlet lists.

5.20.23 espresso.interaction.LennardJones

class espresso.interaction.LennardJones.**CellListLennardJonesLocal**(*stor*)
The (local) Lennard Jones interaction using cell lists.

class espresso.interaction.LennardJones.**FixedPairListLennardJonesLocal**(*system*, *vl*, *potential*)
The (local) Lennard-Jones interaction using FixedPair lists.

class espresso.interaction.LennardJones.**LennardJones**
The Lennard-Jones potential.

class espresso.interaction.LennardJones.**LennardJonesLocal**(*epsilon=1.0*, *sigma=1.0*, *cutoff=inf*, *shift='auto'*)
The (local) Lennard-Jones potential.

class espresso.interaction.LennardJones.**VerletListAdressLennardJones2Local**(*vl*, *fixedTupleList*)
The (local) Lennard Jones interaction using Verlet lists.

class espresso.interaction.LennardJones.**VerletListAdressLennardJonesLocal**(*vl*, *fixedTupleList*)
The (local) Lennard Jones interaction using Verlet lists.

class espresso.interaction.LennardJones.**VerletListHadressLennardJones2Local**(*vl*, *fixedTupleList*, *KTI=False*)
The (local) Lennard Jones interaction using Verlet lists.

class espresso.interaction.LennardJones.**VerletListHadressLennardJonesLocal**(*vl*, *fixedTupleList*, *KTI=False*)
The (local) Lennard Jones interaction using Verlet lists.

class espresso.interaction.LennardJones.**VerletListLennardJonesLocal**(*vl*)
The (local) Lennard Jones interaction using Verlet lists.

5.20.24 espresso.interaction.LennardJonesAutoBonds

class espresso.interaction.LennardJonesAutoBonds.**CellListLennardJonesAutoBondsLocal**(*stor*)
The (local) Lennard Jones auto bonds interaction using cell lists.

class espresso.interaction.LennardJonesAutoBonds.**FixedPairListLennardJonesAutoBondsLocal**(*syste vl*, *potential*)
The (local) Lennard-Jones auto bonds interaction using FixedPair lists.

class espresso.interaction.LennardJonesAutoBonds.**LennardJonesAutoBonds**
The Lennard-Jones auto bonds potential.

class espresso.interaction.LennardJonesAutoBonds.**LennardJonesAutoBondsLocal**(*epsilon=1.0*, *sigma=1.0*, *cutoff=inf*, *bondlist=None*, *maxcrosslinks=2*)
The (local) Lennard-Jones auto bond potential.

class espresso.interaction.LennardJonesAutoBonds.**VerletListAdressLennardJonesAutoBondsLocal**(*v*
f
t
p

The (local) Lennard Jones auto bonds interaction using Verlet lists.

class espresso.interaction.LennardJonesAutoBonds.**VerletListHadressLennardJonesAutoBondsLocal**

The (local) Lennard Jones auto bonds interaction using Verlet lists.

class espresso.interaction.LennardJonesAutoBonds.**VerletListLennardJonesAutoBondsLocal**(*vl*)
The (local) Lennard Jones auto bonds interaction using Verlet lists.

5.20.25 espresso.interaction.LennardJonesCapped

class espresso.interaction.LennardJonesCapped.**CellListLennardJonesCappedLocal**(*stor*)
The (local) Lennard Jones interaction using cell lists.

class espresso.interaction.LennardJonesCapped.**FixedPairListLennardJonesCappedLocal**(*system*, *vl*, *potential*)
The (local) Lennard-Jones interaction using FixedPair lists.

class espresso.interaction.LennardJonesCapped.**LennardJonesCapped**
The Lennard-Jones potential.

class espresso.interaction.LennardJonesCapped.**LennardJonesCappedLocal**(*epsilon=1.0*, *sigma=1.0*, *cutoff=inf*, *caprad=0.0*, *shift='auto'*)
The (local) Lennard-Jones potential with force capping.

class espresso.interaction.LennardJonesCapped.**VerletListAdressLennardJonesCappedLocal**(*vl*, *fixedtupleList*)
The (local) Lennard Jones interaction using Verlet lists.

class espresso.interaction.LennardJonesCapped.**VerletListHadressLennardJonesCappedLocal**(*vl*, *fixedtupleList*, *KTI=Fa*
The (local) Lennard Jones interaction using Verlet lists.

class espresso.interaction.LennardJonesCapped.**VerletListLennardJonesCappedLocal**(*vl*)
The (local) Lennard Jones interaction using Verlet lists.

5.20.26 espresso.interaction.LennardJonesEnergyCapped

class espresso.interaction.LennardJonesEnergyCapped.**CellListLennardJonesEnergyCappedLocal**(*stor*
The (local) Lennard Jones interaction using cell lists.

class espresso.interaction.LennardJonesEnergyCapped.**FixedPairListLennardJonesEnergyCappedLoca*

The (local) Lennard-Jones interaction using FixedPair lists.

class espresso.interaction.LennardJonesEnergyCapped.**LennardJonesEnergyCapped**
The Lennard-Jones potential.

class espresso.interaction.LennardJonesEnergyCapped.**LennardJonesEnergyCappedLocal**(*epsilon=1.0*, *sigma=1.0*, *cutoff=inf*, *caprad=0.0*, *shift='auto'*)
The (local) Lennard-Jones potential with energy capping.

class espresso.interaction.LennardJonesEnergyCapped.**VerletListAdressLennardJonesEnergyCappedL*

The (local) Lennard Jones interaction using Verlet lists.

class espresso.interaction.LennardJonesEnergyCapped.**VerletListHadressLennardJonesEnergyCapped*

The (local) Lennard Jones interaction using Verlet lists.

class espresso.interaction.LennardJonesEnergyCapped.**VerletListLennardJonesEnergyCappedLocal**(*v*
The (local) Lennard Jones interaction using Verlet lists.

5.20.27 espresso.interaction.LennardJonesExpand

class espresso.interaction.LennardJonesExpand.**CellListLennardJonesExpandLocal**(*stor*)
The (local) LennardJonesExpand interaction using cell lists.

class espresso.interaction.LennardJonesExpand.**FixedPairListLennardJonesExpandLocal**(*system*, *vl*, *potential*)
The (local) LennardJonesExpand interaction using FixedPair lists.

class espresso.interaction.LennardJonesExpand.**LennardJonesExpand**
The LennardJonesExpand potential.

class espresso.interaction.LennardJonesExpand.**LennardJonesExpandLocal**(*epsilon=1.0*, *sigma=1.0*, *delta=0.0*, *cutoff=inf*, *shift='auto'*)
The (local) LennardJonesExpand potential.

class espresso.interaction.LennardJonesExpand.**VerletListLennardJonesExpandLocal**(*vl*)
The (local) LennardJonesExpand interaction using Verlet lists.

5.20.28 espresso.interaction.LennardJonesGromacs

class espresso.interaction.LennardJonesGromacs.**CellListLennardJonesGromacsLocal**(*stor*)
The (local) LennardJonesGromacs interaction using cell lists.

class espresso.interaction.LennardJonesGromacs.**FixedPairListLennardJonesGromacsLocal**(*system*, *vl*, *potential*)
The (local) LennardJonesGromacs interaction using FixedPair lists.

class espresso.interaction.LennardJonesGromacs.**LennardJonesGromacs**
The LennardJonesGromacs potential.

class espresso.interaction.LennardJonesGromacs.**LennardJonesGromacsLocal**(*epsilon=1.0*, *sigma=1.0*, *r1=0.0*, *cutoff=inf*, *shift='auto'*)
The (local) LennardJonesGromacs potential.

class espresso.interaction.LennardJonesGromacs.**VerletListLennardJonesGromacsLocal**(*vl*)
The (local) LennardJonesGromacs interaction using Verlet lists.

5.20.29 espresso.interaction.Morse

class espresso.interaction.Morse.**CellListMorseLocal**(*stor*)
The (local) Morse interaction using cell lists.

class espresso.interaction.Morse.**FixedPairListMorseLocal**(*system*, *vl*, *potential*)
The (local) Morse interaction using FixedPair lists.

class espresso.interaction.Morse.**Morse**
The Morse potential.

class espresso.interaction.Morse.**MorseLocal**(*epsilon=1.0*, *alpha=1.0*, *rMin=0.0*, *cutoff=inf*, *shift='auto'*)
The (local) Morse potential.

class espresso.interaction.Morse.**VerletListAdressMorseLocal**(*vl*, *fixedtupleList*)
The (local) Morse interaction using Verlet lists.

class espresso.interaction.Morse.**VerletListHadressMorseLocal**(*vl*, *fixedtupleList*, *KTI=False*)
The (local) Morse interaction using Verlet lists.

class espresso.interaction.Morse.**VerletListMorseLocal**(*vl*)
The (local) Morse interaction using Verlet lists.

5.20.30 espresso.interaction.OPLS

class espresso.interaction.OPLS.**FixedQuadrupleListOPLSLocal**(*system*, *vl*, *potential*)
The (local) OPLS interaction using FixedQuadruple lists.

class espresso.interaction.OPLS.**OPLS**
The OPLS potential.

class espresso.interaction.OPLS.**OPLSLocal**(*K1=1.0*, *K2=0.0*, *K3=0.0*, *K4=0.0*)
The (local) OPLS potential.

5.20.31 espresso.interaction.Potential

5.20.32 espresso.interaction.PotentialUniqueDist

5.20.33 espresso.interaction.PotentialVSpherePair

5.20.34 espresso.interaction.Quartic

class espresso.interaction.Quartic.**FixedPairListQuarticLocal**(*system*, *vl*, *potential*)
The (local) Quartic interaction using FixedPair lists.

class espresso.interaction.Quartic.**Quartic**
The Quartic potential.

class espresso.interaction.Quartic.**QuarticLocal**(*K=1.0*, *r0=0.0*, *cutoff=inf*, *shift=0.0*)
The (local) Quartic potential.

5.20.35 espresso.interaction.ReactionFieldGeneralized

class espresso.interaction.ReactionFieldGeneralized.**CellListReactionFieldGeneralizedLocal**(*stor*
The (local) ReactionFieldGeneralized interaction using cell lists.

class espresso.interaction.ReactionFieldGeneralized.**ReactionFieldGeneralized**
The ReactionFieldGeneralized potential.

class espresso.interaction.ReactionFieldGeneralized.**ReactionFieldGeneralizedLocal**(*prefactor=1.0*, *kappa=0.0*, *epsilon1=1.0*, *epsilon2=80.0*, *cutoff=inf*, *shift='auto'*)
The (local) ReactionFieldGeneralized potential.

class espresso.interaction.ReactionFieldGeneralized.**VerletListAdressReactionFieldGeneralizedL**

The (local) ReactionFieldGeneralized interaction using Verlet lists.

class espresso.interaction.ReactionFieldGeneralized.**VerletListHadressReactionFieldGeneralized**

The (local) ReactionFieldGeneralized interaction using Verlet lists.

class espresso.interaction.ReactionFieldGeneralized.**VerletListReactionFieldGeneralizedLocal**(*v*
The (local) ReactionFieldGeneralized interaction using Verlet lists.

5.20.36 espresso.interaction.SoftCosine

class espresso.interaction.SoftCosine.**CellListSoftCosineLocal**(*stor*)
The (local) SoftCosine interaction using cell lists.

class espresso.interaction.SoftCosine.**FixedPairListSoftCosineLocal**(*system*, *vl*, *potential*)
The (local) SoftCosine interaction using FixedPair lists.

class espresso.interaction.SoftCosine.**SoftCosine**
The SoftCosine potential.

class espresso.interaction.SoftCosine.**SoftCosineLocal**(*A=1.0*, *cutoff=inf*, *shift='auto'*)
The (local) SoftCosine potential.

class espresso.interaction.SoftCosine.**VerletListSoftCosineLocal**(*stor*)
The (local) SoftCosine interaction using cell lists.

5.20.37 espresso.interaction.StillingerWeberPairTerm

class espresso.interaction.StillingerWeberPairTerm.**CellListStillingerWeberPairTermLocal**(*stor*)
The (local) Lennard Jones interaction using cell lists.

class espresso.interaction.StillingerWeberPairTerm.**FixedPairListStillingerWeberPairTermLocal**

The (local) Lennard-Jones interaction using FixedPair lists.

class espresso.interaction.StillingerWeberPairTerm.**StillingerWeberPairTerm**
The Lennard-Jones potential.

class espresso.interaction.StillingerWeberPairTerm.**StillingerWeberPairTermLocal**(*A*, *B*, *p*, *q*, *epsilon=1.0*, *sigma=1.0*, *cutoff=inf*)
The (local) Lennard-Jones potential.

class espresso.interaction.StillingerWeberPairTerm.**VerletListAdressStillingerWeberPairTermLoc**

The (local) Lennard Jones interaction using Verlet lists.

class espresso.interaction.StillingerWeberPairTerm.**VerletListHadressStillingerWeberPairTermLo**

The (local) Lennard Jones interaction using Verlet lists.

class espresso.interaction.StillingerWeberPairTerm.**VerletListStillingerWeberPairTermLocal**(*vl*)
The (local) Lennard Jones interaction using Verlet lists.

5.20.38 espresso.interaction.StillingerWeberPairTermCapped

class espresso.interaction.StillingerWeberPairTermCapped.**CellListStillingerWeberPairTermCappe**
The (local) Lennard Jones interaction using cell lists.

class espresso.interaction.StillingerWeberPairTermCapped.**FixedPairListStillingerWeberPairTerm**

The (local) Lennard-Jones interaction using FixedPair lists.

class espresso.interaction.StillingerWeberPairTermCapped.**StillingerWeberPairTermCapped**
The Lennard-Jones potential.

class espresso.interaction.StillingerWeberPairTermCapped.**StillingerWeberPairTermCappedLocal**(*A*
E
p
q
e
s
s
c
c
c

The (local) Lennard-Jones potential.

class espresso.interaction.StillingerWeberPairTermCapped.**VerletListAdressStillingerWeberPairT**

The (local) Lennard Jones interaction using Verlet lists.

class espresso.interaction.StillingerWeberPairTermCapped.**VerletListHadressStillingerWeberPair**

The (local) Lennard Jones interaction using Verlet lists.

class espresso.interaction.StillingerWeberPairTermCapped.**VerletListStillingerWeberPairTermCap**
The (local) Lennard Jones interaction using Verlet lists.

5.20.39 espresso.interaction.StillingerWeberTripleTerm

class espresso.interaction.StillingerWeberTripleTerm.**FixedTripleListStillingerWeberTripleTerm**

The (local) StillingerWeberTripleTerm interaction using FixedTriple lists.

class espresso.interaction.StillingerWeberTripleTerm.**StillingerWeberTripleTerm**
The StillingerWeberTripleTerm potential.

class espresso.interaction.StillingerWeberTripleTerm.**StillingerWeberTripleTermLocal**(*gamma=0.0*, *theta0=0.0*, *lmbd=0.0*, *epsilon=1.0*, *sigma=1.0*, *cutoff=inf*)

The (local) StillingerWeberTripleTerm potential.

class espresso.interaction.StillingerWeberTripleTerm.**VerletListStillingerWeberTripleTermLocal**

The (local) StillingerWeberTripleTerm interaction using VerletListTriple.

5.20.40 espresso.interaction.Tabulated

class espresso.interaction.Tabulated.**CellListTabulatedLocal**(*stor*)
The (local) tabulated interaction using cell lists.

class espresso.interaction.Tabulated.**FixedPairListTabulatedLocal**(*system*, *vl*, *potential*)

The (local) tabulated interaction using FixedPair lists.

class espresso.interaction.Tabulated.**Tabulated**
The Tabulated potential.

class espresso.interaction.Tabulated.**TabulatedLocal**(*itype*, *filename*, *cutoff=inf*)
The (local) tabulated potential.

class espresso.interaction.Tabulated.**VerletListAdressTabulatedLocal**(*vl*, *fixedtupleList*)

The (local) tabulated interaction using Verlet lists.

class espresso.interaction.Tabulated.**VerletListHadressTabulatedLocal**(*vl*, *fixedtupleList*, *KTI=False*)

The (local) tabulated interaction using Verlet lists.

class espresso.interaction.Tabulated.**VerletListTabulatedLocal**(*vl*)
The (local) tabulated interaction using Verlet lists.

5.20.41 espresso.interaction.TabulatedAngular

class espresso.interaction.TabulatedAngular.**FixedTripleListTabulatedAngularLocal**(*system*, *vl*, *potential*)

The (local) tanulated angular interaction using FixedTriple lists.

class espresso.interaction.TabulatedAngular.**TabulatedAngular**

The TabulatedAngular potential.

class espresso.interaction.TabulatedAngular.**TabulatedAngularLocal**(*itype*, *filename*)

The (local) tabulated angular potential.

5.20.42 espresso.interaction.TabulatedDihedral

class espresso.interaction.TabulatedDihedral.**FixedQuadrupleListTabulatedDihedralLocal**(*system*, *vl*, *potential*)

The (local) tanulated dihedral interaction using FixedQuadruple lists.

class espresso.interaction.TabulatedDihedral.**TabulatedDihedral**

The TabulatedDihedral potential.

class espresso.interaction.TabulatedDihedral.**TabulatedDihedralLocal**(*itype*, *filename*)

The (local) tabulated dihedral potential.

5.20.43 espresso.interaction.TersoffPairTerm

class espresso.interaction.TersoffPairTerm.**CellListTersoffPairTermLocal**(*stor*)

The (local) Lennard Jones interaction using cell lists.

class espresso.interaction.TersoffPairTerm.**FixedPairListTersoffPairTermLocal**(*system*, *vl*, *potential*)

The (local) Lennard-Jones interaction using FixedPair lists.

class espresso.interaction.TersoffPairTerm.**TersoffPairTerm**

The Lennard-Jones potential.

class espresso.interaction.TersoffPairTerm.**TersoffPairTermLocal**(*A*, *lambda1*, *R*, *D*, *cutoff=inf*)

The (local) Lennard-Jones potential.

class espresso.interaction.TersoffPairTerm.**VerletListTersoffPairTermLocal**(*vl*)

The (local) Lennard Jones interaction using Verlet lists.

5.20.44 espresso.interaction.TersoffTripleTerm

class espresso.interaction.TersoffTripleTerm.**FixedTripleListTersoffTripleTermLocal**(*system*, *ftl*, *potential*)
The (local) TersoffTripleTerm interaction using FixedTriple lists.

class espresso.interaction.TersoffTripleTerm.**TersoffTripleTerm**
The TersoffTripleTerm potential.

class espresso.interaction.TersoffTripleTerm.**TersoffTripleTermLocal**(*B=0.0*, *lambda2=0.0*, *R=0.0*, *D=0.0*, *n=1.0*, *beta=1.0*, *m=1.0*, *lambda3=1.0*, *gamma=0.0*, *c=1.0*, *d=1.0*, *theta0=0.0*, *cutoff1=inf*, *cutoff2=inf*)
The (local) TersoffTripleTerm potential.

class espresso.interaction.TersoffTripleTerm.**VerletListTersoffTripleTermLocal**(*system*, *vl3*)
The (local) TersoffTripleTerm interaction using VerletListTriple.

5.20.45 espresso.interaction.VSpherePair

class espresso.interaction.VSpherePair.**VSpherePair**
The Lennard-Jones potential.

class espresso.interaction.VSpherePair.**VSpherePairLocal**(*epsilon=1.0*, *cutoff=inf*, *shift='auto'*)
The (local) Lennard-Jones potential.

class espresso.interaction.VSpherePair.**VerletListVSpherePairLocal**(*vl*)
The (local) Lennard Jones interaction using Verlet lists.

5.20.46 espresso.interaction.VSphereSelf

class espresso.interaction.VSphereSelf.**SelfVSphereLocal**(*system*, *potential*)
The (local) VSphere interaction using Cell List lists.

class espresso.interaction.VSphereSelf.**VSphereSelf**
The VSphereSelf potential.

class espresso.interaction.VSphereSelf.**VSphereSelfLocal**(*e1=0.0*, *a1=1.0*, *a2=0.0*, *Nb=1*, *cutoff=inf*, *shift=0.0*)
The (local) VSphereSelf potential.

5.20.47 espresso.interaction.Zero

class espresso.interaction.Zero.**Zero**
The Zero potential.

5.21 io

5.21.1 DumpGRO - IO Object

- *dump()* write configuration to trajectory GRO file. By default filename is "out.gro", coordinates are folded.

  Properties

- *filename* Name of trajectory file. By default trajectory file name is "out.gro"
- *unfolded* False if coordinates are folded, True if unfolded. By default - False
- *append* True if new trajectory data is appended to existing trajectory file. By default - True
- *length_factor* If length dimension in current system is nm, and unit is 0.23 nm, for example, then length_factor should be 0.23
- *length_unit* It is length unit. Can be LJ, nm or A. By default - LJ

usage:

writing down trajectory

```
>>> dump_conf_gro = espresso.io.DumpGRO(system, integrator, filename='trajectory.gro')
>>> for i in range (200):
>>>   integrator.run(10)
>>>   dump_conf_gro.dump()
```

writing down trajectory using ExtAnalyze extension

```
>>> dump_conf_gro = espresso.io.DumpGRO(system, integrator, filename='trajectory.gro')
>>> ext_analyze = espresso.integrator.ExtAnalyze(dump_conf_gro, 10)
>>> integrator.addExtension(ext_analyze)
>>> integrator.run(2000)
```

Both exapmles will give the same result: 200 configurations in trajectory .gro file.

setting up length scale

For example, the Lennard-Jones model for liquid argon with $\sigma = 0.34[nm]$

```
>>> dump_conf_gro = espresso.io.DumpGRO(system, integrator, filename='trj.gro', unfolded=False, lengt
```

will produce trj.gro with in nanometers

class espresso.io.DumpGRO.**DumpGROLocal** (*system*, *integrator*, *filename='out.gro'*, *unfolded=False*, *length_factor=1.0*, *length_unit='LJ'*, *append=True*)
The (local) storage of configurations.

5.21.2 DumpXYZ - IO Object

- *dump()* write configuration to trajectory XYZ file. By default filename is "out.xyz", coordinates are folded.

  Properties

- *filename* Name of trajectory file. By default trajectory file name is "out.xyz"
- *unfolded* False if coordinates are folded, True if unfolded. By default - False
- *append* True if new trajectory data is appended to existing trajectory file. By default - True
- *length_factor* If length dimension in current system is nm, and unit is 0.23 nm, for example, then length_factor should be 0.23
- *length_unit* It is length unit. Can be LJ, nm or A. By default - LJ

usage:

writing down trajectory

```
>>> dump_conf_xyz = espresso.io.DumpXYZ(system, integrator, filename='trajectory.xyz')
>>> for i in range (200):
>>>   integrator.run(10)
>>>   xyz.dump()
```

writing down trajectory using ExtAnalyze extension

```
>>> dump_conf_xyz = espresso.io.DumpXYZ(system, integrator, filename='trajectory.xyz')
>>> ext_analyze = espresso.integrator.ExtAnalyze(dump_conf_xyz, 10)
>>> integrator.addExtension(ext_analyze)
>>> integrator.run(2000)
```

Both exapmles will give the same result: 200 configurations in trajectory .xyz file.

setting up length scale

For example, the Lennard-Jones model for liquid argon with $\sigma = 0.34[nm]$

```
>>> dump_conf_xyz = espresso.io.DumpXYZ(system, integrator, filename='trj.xyz', unfolded=False, lengt
```

will produce trj.xyz with in nanometers

class espresso.io.DumpXYZ.**DumpXYZLocal**(*system*, *integrator*, *filename='out.xyz'*, *unfolded=False*, *length_factor=1.0*, *length_unit='LJ'*, *append=True*)

The (local) storage of configurations.

5.22 espresso

5.22.1 PMI - Parallel Method Invocation

PMI allows users to write serial Python scripts that use functions and classes that are executed in parallel.

PMI is intended to be used in data-parallel environments, where several threads run in parallel and can communicate via MPI.

In PMI mode, a single thread of control (a python script that runs on the *controller*, i.e. the MPI root task) can invoke arbitrary functions on all other threads (the *workers*) in parallel via *call()*, *invoke()* and *reduce()*. When the function on the workers return, the control is returned to the controller.

This model is equivalent to the "Fork-Join execution model" used e.g. in OpenMP.

PMI also allows to create parallel instances of object classes via *create()*, i.e. instances that have a corresponding object instance on all workers. *call()*, *invoke()* and *reduce()* can be used to call arbitrary methods of these instances.

to execute arbitrary code on all workers, *exec_()* can be used, and to import python modules to all workers, use 'import_()'.

Main program

On the workers, the main program of a PMI script usually consists of a single call to the function *startWorkerLoop()*. On the workers, this will start an infinite loop on the workers that waits to receive the next PMI call, while it will immediately return on the controller. On the workers, the loop ends only, when one of the commands *finalizeWorkers()* or *stopWorkerLoop()* is issued on the controller. A typical PMI main program looks like this:

```
>>> # compute 2*factorial(42) in parallel
>>> import pmi
>>>
>>> # start the worker loop
>>> # on the controller, this function returns immediately
>>> pmi.startWorkerLoop()
>>>
>>> # Do the parallel computation
>>> pmi.import_('math')
>>> pmi.reduce('lambda a,b: a+b', 'math.factorial', 42)
>>>
>>> # exit all workers
>>> pmi.finalizeWorkers()
```

Instead of using *finalizeWorkers()* at the end of the script, you can call *registerAtExit()* anywhere else, which will cause *finalizeWorkers()* to be called when the python interpreter exits.

Alternatively, it is possible to use PMI in an SPMD-like fashion, where each call to a PMI command on the controller must be accompanied by a corresponding call on the worker. This can be either a simple call to *receive()* that accepts any PMI command, or a call to the identical PMI command. In that case, the arguments of the call to the PMI command on the workers are ignored. In this way, it is possible to write SPMD scripts that profit from the PMI communication patterns.

```
>>> # compute 2*factorial(42) in parallel
>>> import pmi
>>>
>>> pmi.exec_('import math')
>>> pmi.reduce('lambda a,b: a+b', 'math.factorial', 42)
```

To start the worker loop, the command *startWorkerLoop()* can be issued on the workers. To stop the worker loop, *stopWorkerLoop()* can be issued on the controller, which will end the worker loop without exiting the workers.

Controller commands

These commands can be called in the controller script. When any of these commands is issued on a worker during the worker loop, a *UserError* is raised.

- *call()*, *invoke()*, *reduce()* to call functions and methods in parallel
- *create()* to create parallel object instances
- *exec_()* and *import_()* to execute arbitrary python code in parallel and to import classes and functions into the global namespace of pmi.
- *sync()* to make sure that all deleted PMI objects have been deleted.
- *finalizeWorkers()* to stop and exit all workers
- *registerAtExit()* to make sure that finalizeWorkers() is called when python exits on the controller
- *stopWorkerLoop()* to interrupt the worker loop an all workers and to return control to the single workers

Worker commands

These commands can be called on a worker.

- *startWorkerLoop()* to start the worker loop
- *receive()* to receive a single PMI command
- *call()*, *invoke()*, *reduce()*, *create()* and *exec_()* to receive a single corresponding PMI command. Note that these commands will ignore any arguments when called on a worker.

PMI Proxy metaclass

The *Proxy* metaclass can be used to easily generate front-end classes to distributed PMI classes. . . .

Useful constants and variables

The pmi module defines the following useful constants and variables:

- *isController* is True when used on the controller, False otherwise
- *isWorker* = not isController
- *ID* is the rank of the MPI task
- *CONTROLLER* is the rank of the Controller (normally the MPI root)
- *workerStr* is a string describing the thread ('Worker #' or 'Controller')
- *inWorkerLoop* is True, if PMI currently executes the worker loop on the workers.

espresso.pmi.**exec_**(**args*)
Controller command that executes arbitrary python code on all (active) workers.

exec_() allows to execute arbitrary Python code on all workers. It can be used to define classes and functions on all workers. Modules should not be imported via exec_(), instead import_() should be used.

Each element of args should be string that is executed on all workers.

Example:

```
>>> pmi.exec_('import hello')
>>> hw = pmi.create('hello.HelloWorld')
```

espresso.pmi.**import_**(**args*)
Controller command that imports python modules on all (active) workers.

Each element of args should be a module name that is imported to all workers.

Example:

```
>>> pmi.import_('hello')
>>> hw = pmi.create('hello.HelloWorld')
```

espresso.pmi.**create**(*cls=None*, **args*, ***kwds*)
Controller command that creates an object on all workers.

cls describes the (new-style) class that should be instantiated. args are the arguments to the constructor of the class. Only classes that are known to PMI can be used, that is, classes that have been imported to pmi via *exec_()* or *import_()*.

Example:

```
>>> pmi.exec_('import hello')
>>> hw = pmi.create('hello.HelloWorld')
>>> print(hw)
MPI process #0: Hello World!
MPI process #1: Hello World!
...
```

Alternative: Note that in this case the class has to be imported to the calling module *and* via PMI.

```
>>> import hello
>>> pmi.exec_('import hello')
>>> hw = pmi.create(hello.HelloWorld)
>>> print(hw)
MPI process #0: Hello World!
MPI process #1: Hello World!
...
```

espresso.pmi.**call**(**args*, ***kwds*)

Call a function on all workers, returning only the return value on the controller.

function denotes the function that is to be called, args and kwds are the arguments to the function. If kwds contains keys that start with with the prefix '__pmictr_', they are stripped of the prefix and are passed only to the controller. If the function should return any results, it will be locally returned. Only functions that are known to PMI can be used, that is functions that have been imported to pmi via *exec_()* or *import_()*.

Example:

```
>>> pmi.exec_('import hello')
>>> hw = pmi.create('hello.HelloWorld')
>>> pmi.call(hw.hello)
>>> # equivalent:
>>> pmi.call('hello.HelloWorld', hw)
```

Note, that you can use only functions that are know to PMI when *call()* is called, i.e. functions in modules that have been imported via *exec_()*.

espresso.pmi.**invoke**(**args*, ***kwds*)

Invoke a function on all workers, gathering the return values into a list.

function denotes the function that is to be called, args and kwds are the arguments to the function. If kwds contains keys that start with with the prefix '__pmictr_', they are stripped of the prefix and are passed only to the controller.

On the controller, invoke() returns the results of the different workers as a list. On the workers, invoke returns None. Only functions that are known to PMI can be used, that is functions that have been imported to pmi via *exec_()* or *import_()*.

Example:

```
>>> pmi.exec_('import hello')
>>> hw = pmi.create('hello.HelloWorld')
>>> messages = pmi.invoke(hw.hello())
>>> # alternative:
>>> messages = pmi.invoke('hello.HelloWorld.hello', hw)
```

espresso.pmi.**reduce**(**args*, ***kwds*)

Invoke a function on all workers, reducing the return values to a single value.

reduceOp is the (associative) operator that is used to process the return values, function denotes the function that is to be called, args and kwds are the arguments to the function. If kwds contains keys that start with with the prefix '__pmictr_', they are stripped of the prefix and are passed only to the controller.

reduce() reduces the results of the different workers into a single value via the operation reduceOp. reduceOp is assumed to be associative. Both reduceOp and function have to be known to PMI, that is they must have been imported to pmi via *exec_()* or *import_()*.

Example:

```
>>> pmi.exec_('import hello')
>>> pmi.exec_('joinstr=lambda a,b: "\n".join(a,b)')
>>> hw = pmi.create('hello.HelloWorld')
>>> print(pmi.reduce('joinstr', hw.hello()))
>>> # equivalent:
>>> print(
...   pmi.reduce('lambda a,b: "\n".join(a,b)',
...              'hello.HelloWorld.hello', hw)
...              )
```

espresso.pmi.**sync**()
Controller command that deletes the PMI objects on the workers that have already been deleted on the controller.

espresso.pmi.**receive**(*expected=None*)
Worker command that receives and handles the next PMI command.

This function waits to receive and handle a single PMI command. If expected is not None and the received command does not equal expected, raise a *UserError*.

espresso.pmi.**startWorkerLoop**()
Worker command that starts the main worker loop.

This function starts a loop that expects to receive PMI commands until *stopWorkerLoop()* or *finalizeWorkers()* is called on the controller.

espresso.pmi.**finalizeWorkers**()
Controller command that stops and exits all workers.

espresso.pmi.**stopWorkerLoop**(*doExit=False*)
Controller command that stops all workers.

If doExit is set, the workers exit afterwards.

espresso.pmi.**registerAtExit**()
Controller command that registers the function *finalizeWorkers()* via atexit.

class espresso.pmi.**Proxy**(*name*, *bases*, *dict*)
A metaclass to be used to create frontend serial objects.

exception espresso.pmi.**UserError**(*msg*)
Raised when PMI has encountered a user error.

5.22.2 espresso.Exceptions

exception espresso.Exceptions.**Error**(*msg*)
Raised to show unrecoverable espresso errors.

exception espresso.Exceptions.**MissingFixedPairList**(*msg*)
Raised to indicate, that a FixedPairList object is missing

exception espresso.Exceptions.**ParticleDoesNotExistHere**(*msg*)
Raised to indicate, that a certain Particle does not exist on a CPU

exception espresso.Exceptions.**UnknownParticleProperty**(*msg*)
Raised to indicate, that a certain Particle property does not exists

5.22.3 espresso.FixedPairDistList

class espresso.FixedPairDistList.**FixedPairDistListLocal**(*storage*)
The (local) fixed pair list.

add(*pid1*, *pid2*)
add pair to fixed pair list

addPairs(*bondlist*)
Each processor takes the broadcasted bondlist and adds those pairs whose first particle is owned by this processor.

getPairs()
return the bonds of the GlobalPairList

getPairsDist()
return the bonds of the GlobalPairList

size()
count number of bonds in GlobalPairList, involves global reduction

5.22.4 espresso.FixedPairList

class espresso.FixedPairList.**FixedPairListLocal**(*storage*)
The (local) fixed pair list.

add(*pid1*, *pid2*)
add pair to fixed pair list

addBonds(*bondlist*)
Each processor takes the broadcasted bondlist and adds those pairs whose first particle is owned by this processor.

getBonds()
return the bonds of the GlobalPairList

getLongtimeMaxBondLocal()
return the maximum bond length this pairlist ever had (since reset or construction)

resetLongtimeMaxBond()
reset long time maximum bond to 0.0

size()
count number of bonds in GlobalPairList, involves global reduction

5.22.5 FixedPairListAdress - Object

The FixedPairListAdress is the Fixed Pair List to be used for AdResS or H-AdResS simulations. When creating the FixedPairListAdress one has to provide the storage and the tuples. Afterwards the bonds can be added. In the example "bonds" is a python list of the form ((pid1, pid2), (pid3, pid4), ...) where each inner pair defines a bond between the particles with the given particle ids.

Example - creating the FixedPairListAdress and adding bonds:

```
>>> ftpl = espresso.FixedTupleList(system.storage)
>>> fpl = espresso.FixedPairListAdress(system.storage, ftpl)
>>> fpl.addBonds(bonds)
```

class espresso.FixedPairListAdress.**FixedPairListAdressLocal**(*storage*, *fixedtupleList*)
The (local) fixed pair list.

add(*pid1*, *pid2*)
add pair to fixed pair list

addBonds(*bondlist*)
Each processor takes the broadcasted bondlist and adds those pairs whose first particle is owned by this processor.

getBonds()
return the bonds of the GlobalPairList

5.22.6 espresso.FixedQuadrupleAngleList

class espresso.FixedQuadrupleAngleList.**FixedQuadrupleAngleListLocal**(*storage*)
The (local) fixed quadruple list.

add(*pid1*, *pid2*, *pid3*, *pid4*)
add quadruple to fixed quadruple list

addQuadruples(*quadruplelist*)
Each processor takes the broadcasted quadruplelist and adds those quadruples whose first particle is owned by this processor.

getQuadruples()
return the quadruples of the GlobalQuadrupleList

getQuadruplesAngles()
return the quadruples with angle

size()
count number of Quadruples in GlobalQuadrupleList, involves global reduction

5.22.7 espresso.FixedQuadrupleList

class espresso.FixedQuadrupleList.**FixedQuadrupleListLocal**(*storage*)
The (local) fixed quadruple list.

add(*pid1*, *pid2*, *pid3*, *pid4*)
add quadruple to fixed quadruple list

addQuadruples(*quadruplelist*)
Each processor takes the broadcasted quadruplelist and adds those quadruples whose first particle is owned by this processor.

getQuadruples()
return the quadruples of the GlobalQuadrupleList

size()
count number of Quadruples in GlobalQuadrupleList, involves global reduction

5.22.8 espresso.FixedSingleList

class espresso.FixedSingleList.**FixedSingleListLocal**(*storage*)
The (local) fixed single list.

add(*pid1*)
add particle to fixed single list

addSingles(*singlelist*)
Each processor takes the broadcasted singlelist and adds those particles that are owned by this processor.

getSingles()
return the singles of the GlobalSingleList

size()
count number of particles in GlobalSingleList, involves global reduction

5.22.9 espresso.FixedTripleAngleList

class espresso.FixedTripleAngleList.**FixedTripleAngleListLocal**(*storage*)
The (local) fixed triple list.

add(*pid1*, *pid2*, *pid3*)
add triple to fixed triple list

addTriples(*triplelist*)
Each processor takes the broadcasted triplelist and adds those triples whose first particle is owned by this processor.

getTriples()
return the triples of the GlobalTripleList

getTriplesAngles()
return the triples of the GlobalTripleList

size()
count number of Triples in GlobalTripleList, involves global reduction

5.22.10 espresso.FixedTripleList

class espresso.FixedTripleList.**FixedTripleListLocal**(*storage*)
The (local) fixed triple list.

add(*pid1*, *pid2*, *pid3*)
add triple to fixed triple list

addTriples(*triplelist*)
Each processor takes the broadcasted triplelist and adds those triples whose first particle is owned by this processor.

getTriples()
return the triples of the GlobalTripleList

size()
count number of Triples in GlobalTripleList, involves global reduction

5.22.11 espresso.FixedTripleListAdress

class espresso.FixedTripleListAdress.**FixedTripleListAdressLocal**(*storage*, *fixedtupleList*)
The (local) fixed triple list.

add(*pid1*, *pid2*)
add pair to fixed triple list

addTriples(*triplelist*)
Each processor takes the broadcasted triplelist and adds those pairs whose first particle is owned by this processor.

5.22.12 espresso.FixedTupleList

class espresso.FixedTupleList.**FixedTupleListLocal**(*storage*)
The (local) fixed tuple list.

size()
count number of Tuple in GlobalTupleList, involves global reduction

5.22.13 FixedTupleListAdress - Object

The FixedTupleListAdress is important for AdResS and H-AdResS simulations. It is the connection between the atomistic and coarse-grained particles. It defines which atomistic particles belong to which coarse-grained particle. In the following example "tuples" is a python list of the form ((pid_CG1, pidAT11, pidAT12, pidAT13, ...), (pid_CG2, pidAT21, pidAT22, pidAT23, ...), ...). Each inner list (pid_CG1, pidAT11, pidAT12, pidAT13, ...) defines a tuple. The first number is the particle id of the coarse-grained particle while the following numbers are the particle ids of the corresponding atomistic particles.

Example - creating the FixedTupleListAdress:

```
>>> ftpl = espresso.FixedTupleListAdress(system.storage)
>>> ftpl.addTuples(tuples)
>>> system.storage.setFixedTuples(ftpl)
```

class espresso.FixedTupleListAdress.**FixedTupleListAdressLocal**(*storage*)
The (local) fixed touple list.

addTuples(*tuplelist*)
Each processor takes the broadcasted tuplelist and adds those tuples whose virtual particle is owned by this processor.

5.22.14 espresso.Int3D

espresso.Int3D.**toInt3D**(**args*)
Try to convert the arguments to a Int3D, returns the argument, if it is already a Int3D.

espresso.Int3D.**toInt3DFromVector**(**args*)
Try to convert the arguments to a Int3D.

This function will only convert to a Int3D if x, y and z are specified.

5.22.15 espresso.MultiSystem

class espresso.MultiSystem.**MultiSystem**
MultiSystemIntegrator to simulate and analyze several systems in parallel.

class espresso.MultiSystem.**MultiSystemLocal**
Local MultiSystem to simulate and analyze several systems in parallel.

5.22.16 espresso.ParallelTempering

5.22.17 espresso.Particle

class espresso.Particle.**ParticleLocal**(*pid*, *storage*)
The local particle.

Throws an exception: * when the particle does not exists locally

TODO: Should throw an exception: * when a ghost particle is to be written * when data is to be read from a ghost that is not available

5.22.18 ParticleAccess - abstract base class for analysis/measurement/io

class espresso.ParticleAccess.**ParticleAccess**
Abstract base class

class espresso.ParticleAccess.**ParticleAccessLocal**
Abstract local base class

5.22.19 espresso.ParticleGroup

class espresso.ParticleGroup.**ParticleGroupLocal**(*storage*)
The local particle group.

5.22.20 espresso.Real3D

espresso.Real3D.**toReal3D**(**args*)
Try to convert the arguments to a Real3D, returns the argument, if it is already a Real3D.

espresso.Real3D.**toReal3DFromVector**(**args*)
Try to convert the arguments to a Real3D.

This function will only convert to a Real3D if x, y and z are specified.

5.22.21 RealND -

This is the object which represents N-dimensional vector. It is an extended Real3D, basicly, it hase the same functionallity but in N-dimetions. First of all it is usefull for classes in 'espresso.analysis'.

Description

...

espresso.RealND.**toRealND**(**args*)
Try to convert the arguments to a RealND, returns the argument, if it is already a RealND.

espresso.RealND.**toRealNDFromVector**(**args*)
Try to convert the arguments to a RealND.

This function will only convert to a RealND if x, y and z are specified.

5.22.22 espresso.Settle

class espresso.Settle.**SettleLocal**(*storage*, *integrator*, *mO=16.0*, *mH=1.0*, *distHH=1.58*, *distOH=1.0*)

The (local) settle.

addMolecules(*moleculelist*)

Each processor takes the broadcasted list.

5.22.23 espresso.Tensor

espresso.Tensor.**toTensor**(**args*)

Try to convert the arguments to a Tensor, returns the argument, if it is already a Tensor.

espresso.Tensor.**toTensorFromVector**(**args*)

Try to convert the arguments to a Tensor.

This function will only convert to a Tensor if x, y and z are specified.

5.22.24 espresso.VerletList

class espresso.VerletList.**VerletListLocal**(*system*, *cutoff*, *exclusionlist=*[ ])

The (local) verlet list.

exclude(*exclusionlist*)

Each processor takes the broadcasted exclusion list and adds it to its list.

getAllPairs()

return the pairs of the local verlet list

localSize()

count number of pairs in local VerletList

totalSize()

count number of pairs in VerletList, involves global reduction

5.22.25 VerletListAdress - Object

The VerletListAdress is the Verlet List to be used for AdResS or H-AdResS simulations. When creating the VerletListAdress one has to provide the system and specify both cutoff for the CG interaction and adrcutoff for the atomistic interaction. Often, it is important to set the atomistic adrcutoff much bigger than the actual interaction's cutoff would be, since also the atomistic part of the VerletListAdress (adrPairs) is built based on the coarse-grained particle positions. For a much larger coarse-grained cutoff it is for example possible to also set the atomistic cutoff on the same value as the coarse-grained one.

Furthermore, the sizes of the explicit and hybrid region have to be provided (dEx and dHy in the example below) and the center of the atomistic region has to be set (adrCenter). In the current implementation this results in a resolution change along the x-direction of the box. A spherical symmetry can be obtained by only minor code changes.

Bascially the VerListAdress provides 4 lists:

- adrZone: A list which holds all particles in the atomistic and hybrid region
- cgZone: A list which holds all particles in the coarse-grained region

- adrPairs: A list which holds all pairs which have at least one particle in the adrZone, i.e. in the atomistic or hybrid region
- vlPairs: A list which holds all pairs which have both particles in the cgZone, i.e. in the coarse-grained region

Example - creating the VerletListAdress:

```
>>> vl      = espresso.VerletListAdress(system, cutoff=rc, adrcut=rc, dEx=ex_size, dHy=hy_size, adrCe
```

class espresso.VerletListAdress.**VerletListAdressLocal**(*system*, *cutoff*, *adrcut*, *dEx*, *dHy*, *adrCenter*=[ ], *pids*=[ ], *exclusionlist*=[ ])

The (local) verlet list AdResS

addAdrParticles(*pids*, *rebuild=True*)
Each processor takes the broadcasted atomistic particles and adds it to its list.

exclude(*exclusionlist*)
Each processor takes the broadcasted exclusion list and adds it to its list.

totalSize()
count number of pairs in VerletList, involves global reduction

5.22.26 espresso.VerletListTriple

class espresso.VerletListTriple.**VerletListTripleLocal**(*system*, *cutoff*, *exclusionlist*=[ ])

The (local) verlet triple list

exclude(*exclusionlist*)
Each processor takes the broadcasted exclusion list and adds it to its list.

getAllTriples()
return the triples of the local verlet list

localSize()
count number of triples in local VerletListTriple

totalSize()
count number of triples in VerletListTriple, involves global reduction

5.23 standard_system

5.23.1 espresso.standard_system.Default

espresso.standard_system.Default.**Default**(*box*, *rc=1.12246*, *skin=0.3*, *dt=0.005*, *temperature=None*)

return default system and integrator, no interactions, no particles are set if tempearture is != None then Langevin thermostat is set to temperature (gamma is 1.0)

5.23.2 espresso.standard_system.KGMelt

5.23.3 espresso.standard_system.LennardJones

espresso.standard_system.LennardJones.**LennardJones**(*num_particles*, *box=(0, 0, 0)*, *rc=1.12246*, *skin=0.3*, *dt=0.005*, *epsilon=1.0*, *sigma=1.0*, *shift='auto'*, *temperature=None*, *xyzfilename=None*, *xyzrfilename=None*)

return random Lennard Jones system and integrator: if tempearture is != None then Langevin thermostat is set to temperature (gamma is 1.0)

5.23.4 espresso.standard_system.Minimal

espresso.standard_system.Minimal.**Minimal**(*num_particles*, *box*, *rc=1.12246*, *skin=0.3*, *dt=0.005*, *temperature=None*)

return minimal system and integrator whithout any interactions defined: particles have random positions in box if tempearture is != None then Langevin thermostat is set to temperature (gamma is 1.0)

5.23.5 espresso.standard_system.PolymerMelt

espresso.standard_system.PolymerMelt.**PolymerMelt**(*num_chains*, *monomers_per_chain*, *box=(0, 0, 0)*, *bondlen=0.97*, *rc=1.12246*, *skin=0.3*, *dt=0.005*, *epsilon=1.0*, *sigma=1.0*, *shift='auto'*, *temperature=None*, *xyzfilename=None*, *xyzrfilename=None*)

returns random walk polymer melt system and integrator: if tempearture is != None then Langevin thermostat is set to temperature (gamma is 1.0)

5.24 storage

5.24.1 espresso.storage.DomainDecomposition

class espresso.storage.DomainDecomposition.**DomainDecompositionLocal**(*system*, *nodeGrid*, *cellGrid*)

The (local) DomainDecomposition.

5.24.2 DomainDecompositionAdress - Object

The DomainDecompositionAdress is the Domain Decomposition for AdResS and H- AdResS simulations. It makes sure that tuples (i.e. a coarse-grained particle and its corresponding atomistic particles) are always stored together on one CPU. When setting DomainDecompositionAdress you have to provide the system as well as the nodegrid and the cellgrid.

Example - setting DomainDecompositionAdress:

```
>>> system.storage = espresso.storage.DomainDecompositionAdress(system, nodeGrid, cellGrid)
```

class espresso.storage.DomainDecompositionAdress.**DomainDecompositionAdressLocal**(*system*, *nodeGrid*, *cellGrid*)

The (local) DomainDecomposition.

5.24.3 espresso.storage.DomainDecompositionNonBlocking

class espresso.storage.DomainDecompositionNonBlocking.**DomainDecompositionNonBlockingLocal**(*syst*
nod
Gri
cell
Gri

The (local) DomainDecompositionNonBlocking.

5.24.4 Storage - Storage Object

This is the base class for all storage objects. All derived classes implement at least the following methods:

- *decompose()*

  Send all particles to their corresponding cell/cpu

- *addParticle(pid, pos)*:

  Add a particle to the storage

- *removeParticle(pid)*:

  Remove a particle with id number *pid* from the storage.

  ```
  >>> system.storage.removeParticle(4)
  ```

  There is an example in *examples* folder

- *getParticle(pid)*:

  Get a particle object. This can be used to get specific particle information:

  ```
  >>> particle = system.storage.getParticle(15)
  >>> print "Particle ID is       : ", particle.id
  >>> print "Particle position is : ", particle.pos
  ```

  you cannot use this particle object to modify particle data. You have to use the modifyParticle command for that (see below).

- *addAdrParticle(pid, pos, last_pos)*:

  Add an AdResS Particle to the storage

- *setFixedTuplesAdress(fixed_tuple_list)*:

- *addParticles(particle_list, *properties)*:

  This routine adds particles with certain properties to the storage.

  param particleList list of particles (and properties) to be added

  param properties property strings

Each particle in the list must be itself a list where each entry corresponds to the property specified in properties.

Example:

```
>>> addParticles([[id, pos, type, ... ], ...], 'id', 'pos', 'type', ...)
```

- *modifyParticle(pid, property, value, decompose='yes')*

  This routine allows to modify any properties of an already existing particle.

  Example:

  ```
  >>> modifyParticle(pid, 'pos', Real3D(new_x, new_y, new_z))
  ```

- *removeAllParticles()*:

  This routine removes all particles from the storage.

- 'system':

  The property 'system' returns the System object of the storage.

Examples:

```
>>> s.storage.addParticles([[1, espresso.Real3D(3,3,3)], [2, espresso.Real3D(4,4,4)]],'id','pos')
>>> s.storage.decompose()
>>> s.storage.modifyParticle(15, 'pos', Real3D(new_x, new_y, new_z))
```

CHAPTER

SIX

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 espresso
>>> 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

SEVEN

REFERENCES

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

BIBLIOGRAPHY

[Berendsen84] Berendsen et al., *J. Chem. Phys.*, 81, **1984**, p. 3684

[Quigley04] 4. Quigley, M.I.J. Probert, *J. Chem. Phys.*, 120, **2004**, p. 11432

[Martyna94] 7. Martyna, D. Tobias, M. Klein, *J. Chem. Phys.*, 101, **1994**, p. 4177

[Allen89] M.P.Allen, D.J.Tildesley, *Computer simulation of liquids*, *Clarendon Press*, **1989** 385 p.

[Deserno98] M.Deserno and C.Holm, *J. Chem. Phys.*, 109(18), **1998**, p.7678

[Berendsen84] Berendsen et al., *J. Chem. Phys.*, 81, **1984**, p. 3684

[Quigley04] 4. Quigley, M.I.J. Probert, *J. Chem. Phys.*, 120, **2004**, p. 11432

[Martyna94] 7. Martyna, D. Tobias, M. Klein, *J. Chem. Phys.*, 101, **1994**, p. 4177

[Allen89] M.P.Allen, D.J.Tildesley, *Computer simulation of liquids*, *Clarendon Press*, **1989** 385 p.

[Deserno98] M.Deserno and C.Holm, *J. Chem. Phys.*, 109(18), **1998**, p.7678

[frenkel02b] Understanding Molecular Simulation, Academic Press, Frenkel, Daan and Smit, Berend, San Diego, 2nd edition

PYTHON MODULE INDEX

e

INDEX

A

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

X

Z