
Pysic

Release 0.4.3

Teemu Hynninen

June 07, 2012

CONTENTS

1	Physical background	3
1.1	Physics of Pysic	3
1.2	Applications and goals	3
1.3	Atomistic interactions	3
1.4	Dynamic charges	4
1.5	Pysic approach	4
2	Getting Pysic	7
2.1	Getting Pysic	7
2.2	Compiling Pysic	7
2.3	External resources	8
3	Performing simulations with Pysic	9
3.1	Running Pysic	9
3.2	Why Python?	9
3.3	The Atomic Simulation Environment	10
3.4	Thinking in objects	10
3.5	Examples	11
3.6	Screencasts	14
4	Structure and syntax in Pysic	17
4.1	Structure and syntax of Pysic	17
4.2	Pysic module	18
4.3	Pysic Utility modules	64
4.4	Pysic Fortran module	69
5	Development of Pysic	145
5.1	Version history	145
5.2	Roadmap	146
5.3	Development	147
	Python Module Index	149
	Index	151

Teemu Hynninen (2011-2012)

Tampere University of Technology Aalto University, Helsinki

Contact: teemu.hynninen@tut.fi, @thynnine

Pysic is a calculator incorporating various empirical pair and many-body potentials in an object-based Python environment and user interface while implementing an efficient numeric core written in Fortran. The immediate aim of the Pysic project is to implement advanced variable charge potentials.

Pysic is designed to interface with the [ASE](#) simulation environment.

The code is developed as part of the [Mordred](#) project.

Pysic is an open source code with emphasis on making the program simple to learn and control as well as the source code readable, extendable and conforming to good programming standards. The source code is freely available at the [github](#) repository. **Note the relocation since June 2012.**

Pysic is in development. Simulations can be run with it, but many parts of the program are not yet fully tested and so bugs must be expected. Similarly, also this documentation is being constantly updated.

PHYSICAL BACKGROUND

1.1 Physics of Pysic

In this section we briefly describe the physical motivation and main ideas behind Pysic. The details of actual algorithms, functions, and implementations included in Pysic are discussed later in sections *Running Pysic* and *Structure and syntax of Pysic*.

1.2 Applications and goals

Phenomena such as Si covalent bonding and charge redistribution at defects and interfaces make semiconductors a very challenging group of materials to describe computationally. Usually quantum mechanical methods such as density functional theory are used, but these approaches are limited by their high computational cost. More sophisticated empirical potentials are being developed for these materials, however, combining fairly accurate precision with a reasonable cost. Notably, the ReaxFF¹ and COMB^{2 3} variable charge potentials have been recently introduced and shown to reproduce the structural properties of for instance Si, SiO₂ and HfO₂. Still these methods are efficient enough to allow the simulation of semiconductor systems at size and time scales relevant for actual device performance (hundreds of thousands of atoms).

1.3 Atomistic interactions

To accurately model condensed matter systems, one often needs to know how the electrons behave, as electrons determine the chemical properties of atoms. Even if the primary interest is the atomic structure, not the electronic one, it may be necessary to include electrons in the simulations in order to calculate the atomistic interactions. The electronic structure must be treated on a quantum mechanical level making these kinds of calculations quite heavy, and so one would often wish to bypass the electronic level of detail and only work with the atomistic structure and a classical description. If we assume that the atomic and electronic degrees of freedom are separate (this is the Born-Oppenheimer approximation, and it is often justified), then in principle the electronic ground state $|\psi\rangle$ is determined by the atomic coordinates $\{\mathbf{R}_i\}$ and nuclear charges $\{Q_i\}$ through the Schrödinger equation

$$H(\{\mathbf{R}_i\}, \{Q_i\})|\psi\rangle = E|\psi\rangle \quad (1.1)$$

$$\Rightarrow |\psi\rangle = \psi(\{\mathbf{R}_i\}, \{Q_i\}), \quad (1.2)$$

¹ A. van Duin, S. Dasgupta, F. Lorant, and W. Goddard, J Phys Chem A 105, 9396 (2001).

² T.-R. Shan, B. D. Devine, T. W. Kemper, S. B. Sinnott, and S. R. Phillpot, Phys Rev B 81, 125328 (2010).

³ T.-R. Shan, D. Bryce, J. Hawkins, A. Asthagiri, S. Phillpot, and S. Sinnott, Phys Rev B 82, 235302 (2010).

where $H = H(\{\mathbf{R}_i\}, \{Q_i\})$ is the Hamiltonian. As the total energy of the system, E , is determined by the electronic state, it too is a function of the atomic configuration

$$E = \langle \psi | H | \psi \rangle = E(\{\mathbf{R}_i\}, \{Q_i\}).$$

So, in principle, the energy of the system and the atomic forces $F_\alpha = -\nabla_{\mathbf{R}_\alpha} E$ can be determined from the atomic configuration. Construction of the energy function $E(\{\mathbf{R}_i\}, \{Q_i\})$ is very challenging though. It is a very complicated function of the coordinates of *all* the atoms in the system, and in practice one needs to further assume that the system can be split in local substructures for which the energy function can be parameterized, and that the total energy is obtained as a sum of the local contributions. For n -body local interactions, this could be written

$$E \approx \sum_{(i_1, i_2, \dots, i_n)} E_{\text{local}}(\mathbf{R}_{i_1}, \dots, \mathbf{R}_{i_n}, Q_{i_1}, \dots, Q_{i_n}), \quad (1.3)$$

where (i_1, i_2, \dots, i_n) are all the sets of n atoms in the system.

1.4 Dynamic charges

In practice, calculations get exponentially more complex when the number of bodies included in the local n -body energy $E_{\text{local}}(\mathbf{R}_{i_1}, \dots, \mathbf{R}_{i_n}, Q_{i_1}, \dots, Q_{i_n})$ in (1.3) increases. Therefore usually only few particles close to each other are included in the local energy function, and all configurations with distant atoms are given a zero energy. This approach does not work, however, if the system exhibits long-ranged collective electronic reconstruction such as local charging or polarization, for instance, due to the presence of defects or interfaces. Changes in charge distribution can drastically alter the local energy function E_{local} . Increasing the number of bodies n in the local interaction function does not solve this problem in general, since the charge redistribution may be a system wide phenomenon that no local function can properly capture.

One way to treat the redistribution of charge is to make the local energy also a function of the total atomic charge $q_i = Q_i - \eta_i e$, where η_i is the (possibly fractional) number of electrons associated with the atom

$$E_{\text{local}} = E_{\text{local}}(\mathbf{R}_{i_1}, \dots, \mathbf{R}_{i_n}, Q_{i_1}, \dots, Q_{i_n}, q_{i_1}, \dots, q_{i_n}).$$

Although electrons do not specifically belong to any atom making η_i ambiguous, this approach allows the inclusion of long ranged charge distribution in the model with reasonable computational cost.

In addition, having the local charge as a parameter of the energy function allows for the optimization of the energy with respect to the local charges, allowing one to search for an equilibrium charge distribution. This is analogous to finding equilibrium structures by optimizing the energy with respect to the atomic coordinates \mathbf{R}_i .

1.5 Pysic approach

The immediate aim in the development of Pysic is to implement atomistic potentials with variable local atomic charges and apply them in the study of semiconductor interfaces, e.g., silicon-hafnia. In the long term, Pysic will include a full library of different atomistic potentials.

Pysic implements a very general framework for calculating energies and forces due to arbitrary local atomistic pair or many body potentials. It is straightforward to implement new types of interactions in the code, and mixing different potentials during the simulations is simple. Furthermore, one can easily evaluate the contribution of different interactions on the total energy and forces by switching on and off specific interactions. So called bond order, or Tersoff, potentials⁴ are also supported, and the user is free to scale any potential with a bond-dependent factor. In addition, in a system with local charges, long ranged Coulomb interactions need to be evaluated. Such $1/r$ -potentials are calculated with the standard Ewald summation algorithm. Implementation of other algorithms such as Particle mesh Ewald⁵ or Wolf summation⁶ is also planned.

⁴ J. Tersoff, Phys Rev B 37, 6991 (1988).

⁵ T. Darden, D. York, and L. Pedersen, Journal of Chemical Physics 98, 10089 (1993).

⁶ D. Wolf, P. Keblinski, S. Phillpot, and J. Eggebrecht, Journal of Chemical Physics 110, 8254 (1999).

In addition, it is planned that various advanced analysis tools are included with the Pysic package. These would include tools for tasks such as potential parametrization or structural analysis using techniques like evolutionary algorithms, machine learning, or Bayesian methods.

GETTING PYSIC

2.1 Getting Pysic

Pysic is currently in development and not yet properly tested. Nonetheless, the source code is available through [github](#) if you wish to try it out. The code is provided with no warranty or support.

2.1.1 Contents of the Pysic repository

When you download Pysic, you should receive a package containing three directories *doc*, *fortran*, and *pysic*. The directory *doc* contains the reStructuredText (`.rst`) source files used for generating the documentation (with [Sphinx](#)). For readability, check the [online version](#). The directories *fortran* and *pysic* contain the Fortran and Python source code, respectively. In addition to these folders, there should be a `readme.txt` file containing installation instructions and a `Makefile` for compiling with `make`.

2.2 Compiling Pysic

This section gives brief instructions on how to compile Pysic.

2.2.1 Requirements

To run Pysic, you will need [Python 2.x](#) and the [numpy](#) and [scipy](#) modules. For dynamic simulation, the [ASE](#) package is required. Some plotting tools also require the [matplotlib](#) package, but Pysic will work without it.

To compile the Fortran core, one needs a Fortran 90 compiler and [f2py](#) (the latter is part of [numpy](#).) For compiling an MPI compatible version, an MPI-Fortran compiler is needed.

2.2.2 Quick compilation

To compile Pysic, go to the directory where you have the `Makefile` and folders *fortran* and *pysic*, and enter the command `make <target>` on command line, where `<target>` is for instance `serial` or `parallel`. A list of available targets are shown with the command `make help`.

The default compilers listed in the `Makefile` are `gfortran` and `mpif90`. You may need to edit the names of the compilers to match those available on your system.

By default, `make` creates the *pysic* Python module as a folder named *pysic* (or *pysic_debug* etc., depending on the type of compilation) in the folder *build*. To run Pysic, copy the folder *pysic* to wherever you wish to run your simulations and import it in Python with `import pysic`.

2.2.3 Compilation under the hood

The Python part of Pysic does not have to be compiled separately, but the Fortran core must be. In order to link the Fortran and Python sides together, you need to compile the code with the [f2py](#) tool, which is part of the [numpy](#) package.

The Fortran part is compiled with the commands:

```
f2py -m pysic_fortran -h pysic_fortran.pyf PyInterface.F90
f2py -c --fcompiler=gfortran --f90exec=mpif90 --f90flags="-D MPI" \
pysic_fortran.pyf Mersenne.F90 MPI.F90 Quaternions.F90 Utility.F90 \
Geometry.F90 Potentials.F90 Core.F90 PyInterface.F90
```

The first run creates the interface file `pysic_fortran.pyf` from the Fortran source file `PyInterface.F90`. The second run compiles the rest of the code in a native Fortran mode and creates the Python accessible binary file `pysic_fortran.so`. From this file, the subroutines of `PyInterface.F90` are callable as Python functions.

Above, the `gfortran` and `mpif90` compilers are used, but change the names to whichever compilers you wish to call. The flag `--f90flags="-D MPI"` invokes the preprocessor which picks the MPI part of the source code. To compile a serial version, leave this flag out.

The created `pysic_fortran.so` appears in Python as a module `pysic_fortran.pysic_interface`. The package structure of `pysic` expects to find it at `pysic.pysic_fortran.pysic_interface`, i.e., the `.so` file should be in the folder `pysic` (the root folder of the Python package). The `Makefile` should take care of putting it in the right place, but if you decide to compile the Fortran part manually, keep in mind that the file needs to be moved there.

For further information on compiling with [f2py](#), consult the [f2py manual](#).

2.3 External resources

Below is a list of tools one may find useful or even necessary when using Pysic:

- [Python](#): The Python language
- [Python documentation](#): The official documentation for Python
- [Python tutorial](#): The official tutorials for Python
- [F2py](#): Fortran to Python interface generator
- [F2py manual](#): The (old) official F2py manual
- [github](#): Version control and repository storing Pysic
- [ASE](#): Atomistic Simulation Environment (ASE)
- [ASE tutorials](#): The official ASE tutorials
- [NumPy](#): Numerical Python
- [SciPy](#): Scientific Python
- [Matplotlib](#): Matplotlib Python plotting library
- [iPython](#): iPython enhanced Python interpreter

PERFORMING SIMULATIONS WITH PYSIC

3.1 Running Pysic

Once you have Python and Pysic working, it's time to learn how to use them. Next we will go through the basic concepts and ideas behind running simulations with Pysic. A collection of examples will demonstrate how to set up simulations in practice.

3.2 Why Python?

[Python](#) is a programming language with a simple human readable syntax yet powerful features and an extensive library. Python is an interpreted language meaning it does not need to be compiled. One can simply feed the source code to an interpreter which reads and interprets it during run. The Python interpreter can also perform calculations, read and write files, render graphics, etc. making Python a powerful tool for scripting. Python is also an object based language enabling sophisticated *Thinking in objects* programming.

In computational physics codes, the most common method of performing the calculations is to have the program read input files to extract the necessary parameters, perform the simulation based on the given input, and print output based on the results. The generated data is then analysed using separate tools. In some cases, more common in commercial programs, the user can control the program through a graphical user interface.

In Pysic, another approach is chosen. Pysic is not a “black box” simulator that turns input data to output data. Instead, Pysic is a *Python module*. In essence, it is a library of tools one can use within Python to perform complicated calculations. Instead of writing an input file - often a complicated and error prone collection of numbers - the user needs to build the simulation in Python. Pysic can then be used in evaluating the energies, forces and other physical quantities of the given system. Python syntax is in general simple and simplicity and user friendliness has also been a key goal in the design of Pysic syntax.

Since Python is a programming language, having Pysic be a part of the language makes it straightforward to write scripts that generate the physical system to be studied and also extract the needed information from the simulations. Instead of generating enormous amounts of data which would then have to be fed to some other analysis program, one can precisely control what kind of data should be produced in the simulations and even analyse the results simultaneously as the simulation is run. Even controlling the simulation based on the produced data is not only possible but easy.

The downside of Pysic being a Python module instead of an independent program is that one has to know the basics of how to run Python. [Python documentation](#) is the best resource for getting started, but some simple first step instructions and *Examples* are also given in this document.

3.3 The Atomic Simulation Environment

The Atomistic Simulation Environment (ASE¹) is a simulation tool developed originally at CAMd, DTU (Technical University of Denmark). The package defines a full atomistic simulation environment for Python, including utilities such as a [graphical user interface](#). Like Pysic, ASE is a collection of Python modules. These modules allow the user to construct atomistic systems, run [molecular dynamics](#) or [geometry optimization](#), do calculations and analysis, etc.

ASE is easily extendable, and in fact, the common way to use ASE is join it with an external calculator which produces the needed physical quantities based on the structures defined by ASE. Pysic is such an extension. In the terms used in ASE, Pysic is a [calculator](#). The main role of calculators in ASE is to calculate forces and energies, and this is also what Pysic does. In addition, Pysic incorporates variable charge potentials with dynamic charge equilibration routines. Since ASE does not contain such routines, they are also handled by Pysic, making it more than just an extension of ASE. ASE does contain efficient dynamics and optimization routines including [constrained algorithms](#) and [nudged elastic band](#) as well as a choice of [thermostats](#), and so Pysic does not implement any such functionality.

The central object in ASE, also used by Pysic, is the [Atoms](#) class. This class defines the complete system geometry including atomic species, coordinates, momenta, supercell, and boundary conditions. Pysic interprets instances of this class as the simulation geometry.

3.4 Thinking in objects

If you are already familiar with Python or languages such as Java or C++, you probably know what object oriented programming is. If you are more of a Fortran77 type, maybe not. Since Pysic relies heavily on the object paradigm, a few words should be said about it.

Let's say we want to simulate a bunch of atoms. To do this, we need to know where they are (coordinates), what element they are, what are their charges, momenta, etc. One way to store the information would be to assign indices to the atoms and create arrays containing the data. One for coordinates, another for momenta, third for charges etc. However, this type of bookkeeping approach gets more and more complicated the more structured data one needs to store. E.g., if we have a neighbor list as an array of atomic indices and wish to find the coordinates of a neighbor of a given atom, we first need to find the correct entry in the list of neighbors to find the index of the neighbor, and then find the entry corresponding to this index in the list of coordinates. In code, it could look like this:

```
neighbor_atom_index = neighbor_lists[atom_index, neighbor_index]
neighbor_atom_coordinates = coordinates[neighbor_atom_index, 1:3]
```

For complicated data hierarchies one may need to do this kind of index juggling for several rounds, which leads to code that is difficult to read and very susceptible to bugs. Furthermore, if one would want to edit the lists of atoms by, say, removing an atom from the system, all the arrays containing data related to atoms would have to be checked in case they contain the to be deleted particle and updated accordingly.

In the object oriented approach, one defines a data structure (called a *class* in Python) capable of storing various types of information in a single instance. So one can define an 'atom' datatype which contains the coordinates (as real numbers), momenta, etc. in one neat package. One can also define a 'neighbor list' datatype which contains a list of 'atom' datatypes. And the 'atom' datatype can contain a 'neighbor list' (although, this is not exactly how ASE handles neighbor lists). Now, the problem of finding the coordinates of a neighbor is solved in a more intuitive way by asking the atom who the neighbor is and the neighbor its coordinates. This might look something like this:

```
neighbor_atom_coordinates = atom.get_neighbor(neighbor_index).get_coordinates()
```

The above example also demonstrates *methods* - object specific functions allowing one to essentially give orders to objects. Objects and methods make it easy to write code that is simple to read and understand, since we humans

¹ ASE: Comput. Sci. Eng., Vol. 4, 56-66, 2002; <https://wiki.fysik.dtu.dk/ase/>

intuitively see the world as objects, not as arrays of data. Another great benefit of the object based model is that when an object is modified, the changes automatically propagate everywhere where that object is referred.

The classes and their methods defined in Pysic are documented in detail in *Structure and syntax of Pysic*, and their basic use is shown in the collection of provided *Examples*. The central class in Pysic is `Pysic`, which is an energy and force calculator for ASE. The interactions according to which the energies are calculated are constructed through the class `Potential`. Utilizing these classes is necessary to run meaningful calculations, though also other classes are defined for special purposes.

3.5 Examples

Next, we go through some basic examples on how to use Pysic, starting from launching Python, finishing with relatively advanced scripting techniques.

3.5.1 First steps with Python

Once you have installed Python - chances are it is already preinstalled in your system - you can launch the Python interpreter with the command `python` on the command line:

```
> python
Python 2.7.2 (#1, Feb 28 2010, 00:02:06)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This will start up Python in interactive mode and you can start writing commands. For instance:

```
>>> a = 1+1
>>> a
2
>>> b = 'hello'
>>> c = b + ' world!'
>>> c
'hello world!'
```

Another way to run Python is by feeding a source file to the interpreter. Say you have the file `first_step.py` containing:

```
left_first = False
if left_first:
    print "Left, right, left!"
else:
    print "Right, left, right!"
```

This can be run with either:

```
> python first_step.py
Right, left, right!
```

or simply:

```
> ./first_step.py
Right, left, right!
```

assuming you have execution permissions for the script. You can also pass command line arguments to a Python script when launching one.

That's it, basically! In the following examples it is shown how Pysic is set up within Python and how simulations can be run. However, as Python is a powerful language, you will certainly get the most out of Pysic by learning some of the basic features of Python. Taking a look at the official [Python tutorial](#) is a good place to start.

3.5.2 Importing Pysic to Python

Modules are accessed in Python by importing them with the `import` command. To get access to Pysic, simply write:

```
>>> import pysic
```

Then, you can access all the functionality in the module `pysic`:

```
>>> pysic_calculator = pysic.Pysic()
>>> pysic.get_names_of_parameters('LJ')
['epsilon', 'sigma']
```

The Fortran interface module is imported with `pysic` as `pysic.pf.pysic_interface`, but it is strongly recommended you do not touch the Fortran core directly.

Altogether, when the module `pysic` is imported, it imports the following non-standard modules:

```
>>> import pysic.pysic_fortran as pf
>>> import numpy as np
>>> import ase.calculators.neighborlist as nbl
```

and defines the functions and classes as documented in the syntax description of `pysic`. In addition, the `start_potentials()` and `start_bond_order_factors()` routines of the Fortran core are automatically invoked in order to initialize the descriptors in the core (see *potentials (Potentials.f90)*). In an MPI environment, the MPI initialization routines `start_mpi()` are also called from the Fortran core. Finally, the random number generator is initialized in the Fortran core by `start_rng()`.

3.5.3 Minimal example of running Pysic

Here is an example of setting up a Pysic calculator for ASE:

```
>>> from ase import Atoms
>>> import pysic
>>> system = Atoms('He2', [[0.0, 0.0, 0.0], [0.0, 0.0, 3.0]])
>>> calc = pysic.Pysic()
>>> system.set_calculator(calc)
>>> physics = pysic.Potential('LJ', cutoff = 10.0)
>>> physics.set_symbols(['He', 'He'])
>>> physics.set_parameter_value('epsilon', 0.1)
>>> physics.set_parameter_value('sigma', 2.5)
>>> calc.add_potential(physics)
```

The example above creates a system of two helium atoms interacting via a Lennard-Jones potential

$$V(r) = \varepsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right]$$
$$\varepsilon = 0.1$$
$$\sigma = 2.5$$

In the code above, `system` is an `ASE Atoms` object containing the structure of the system to be calculated - two He atoms in this case. The object `calc` is an instance of `Pysic`, the `ASE calculator` class defined by Pysic. The

interactions governing the system are defined by the `physics` object, which is an instance of the `Potential` class of `Pysic`.

Now, the potential energy of the system and the forces acting on the atoms can be calculated with:

```
>>> system.get_potential_energy()
-0.022274132189576905
>>> system.get_forces()
array([[ 0.          ,  0.          ,  0.02211693],
       [ 0.          ,  0.          , -0.02211693]])
```

These commands are addressed to the `system` object, but under the hood `system` asks `calc`, i.e., `Pysic`, to do the actual calculations. In order to evaluate the requested quantities, `Pysic` needs the parameters contained in `physics`.

A more compact way to create the calculator would be:

```
>>> physics = pysic.Potential('LJ',
...                           cutoff = 10.0,
...                           symbols = ['He', 'He'],
...                           parameters = [0.1, 2.5])
>>> calc = pysic.Pysic(system, physics)
```

Setting up more complicated interactions works similarly, as is shown in later examples.

3.5.4 Running molecular dynamics

This example shows how to set up a dynamic simulation with `ASE` (also see the [ASE MD tutorial](#)).

First set up a `MgO` crystal:

```
from ase.lattice.compounds import Rocksalt

# Set up a crystal
size = 4
lattice_constant = 4.212
system = Rocksalt(size=(size,size,size), symbol=("Mg", "O"),
                  latticeconstant=lattice_constant, pbc=True)

# Set charges for the atoms
for i in range(len(system)):
    if system[i].get_symbol() == "Mg":
        system[i].set_charge(2)
    elif system[i].get_symbol() == "O":
        system[i].set_charge(-2)
```

Create a `Pysic` calculator. We set up pairwise interactions with the *Buckingham potential* and a Coulomb interaction:

```
import pysic

# Set up a calculator
calc = pysic.Pysic()

# Mg-O pair potential
pot_mgo = pysic.Potential('Buckingham', symbols=['Mg','O'], cutoff=8.0, cutoff_margin=1.0)
pot_mgo.set_parameter_value('A', 1284.38)
pot_mgo.set_parameter_value('C', 0.0)
pot_mgo.set_parameter_value('sigma', 0.2997)
calc.add_potential(pot_mgo)

# O-O pair potential
```

```
pot_oo = pysic.Potential('Buckingham', symbols=['O','O'], cutoff=10.0, cutoff_margin=1.0)
pot_oo.set_parameter_value('A', 9574.96)
pot_oo.set_parameter_value('C', 288474.00)
pot_oo.set_parameter_value('sigma', 0.2192)
calc.add_potential(pot_oo)

# Coulomb interaction
ewald = pysic.CoulombSummation()
ewald.set_parameter_value('epsilon', 0.00552635) # epsilon_0 in e**2/(eV A)
ewald.set_parameter_value('k_cutoff', 0.7) # one should always test the Ewald parameters for convergence
ewald.set_parameter_value('real_cutoff', 10.0) # and optimal speed - long cutoffs can substantially increase
ewald.set_parameter_value('sigma', 1.4) # needed cpu time while short cutoffs give incorrect results
calc.set_coulomb_summation(ewald)

system.set_calculator(calc)
```

Now, let us set up the dynamic simulation.

We initialize the velocity distribution:

```
from ase.md.velocitydistribution import MaxwellBoltzmannDistribution
from ase import units

# Set the momenta corresponding to T=300K
MaxwellBoltzmannDistribution(system, 300*units.kB)
```

And define the dynamics as a VelocityVerlet object:

```
from ase.md.verlet import VelocityVerlet

# We want to run MD with constant energy using the VelocityVerlet algorithm.
dyn = VelocityVerlet(system, 5*units.fs) # 5 fs time step.
```

In order to record the ASE trajectory and print information during simulation, we also define observers and attach them to the dynamics. In a similar fashion we can collect whatever information we need:

```
# Function to print the potential, kinetic and total energy.
def print_energy(a=system):
    epot = a.get_potential_energy() / len(a)
    ekin = a.get_kinetic_energy() / len(a)
    print ("Energy per atom: Epot = %.3feV Ekin = %.3feV (T=%.3fK) Etot = %.3feV" %
          (epot, ekin, ekin/(1.5*units.kB), epot+ekin))
dyn.attach(print_energy, interval=100) # print after every 100 steps

# Save a trajectory
from ase.io.trajectory import PickleTrajectory

traj = PickleTrajectory("example.traj", "w", system) # write a trajectory to the file 'example.traj'
dyn.attach(traj.write, interval=10) # save every 10 step
```

Finally, we run the dynamics:

```
dyn.run(1000) # run for 1000 steps
traj.close() # close the trajectory recording
```

3.6 Screenscasts

We have also prepared screencasts demonstrating the use of Pysic.

3.6.1 Pysic basics

This is a demonstration of a very simple simulation setup while running Pysic in an interactive Python session.

[YouTube link](#)

3.6.2 Pysic in a script

This is a demonstration of running Pysic with a script, including an MPI parallel run.

[YouTube link \(high definition\)](#)

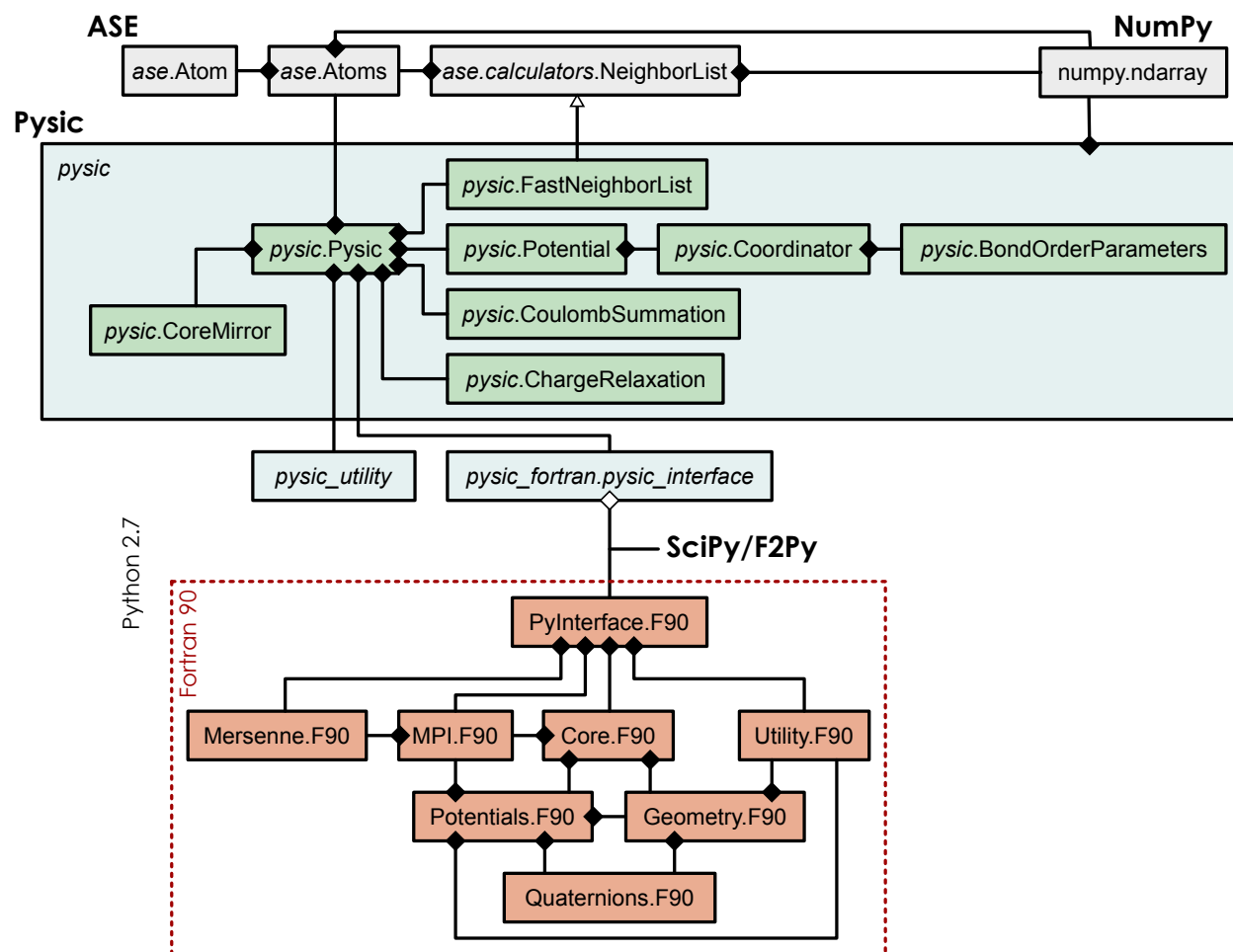
STRUCTURE AND SYNTAX IN PYSIC

4.1 Structure and syntax of Pysic

Pysic can be used with basic functionality with just a few commands. To give access to all the functionality in Pysic, the full API of Pysic is documented in the following section. All the classes and methods including their arguments are explained in detail. Also the types of potentials and bond order factors available are documented, including their mathematical descriptions and the keywords needed for access.

In addition to the Python documentation, also the variables, types, and routines in the Fortran core are listed with comments and explanations. Besides the interface module *pysic_interface* (*PyInterface.f90*), this part of the code is not accessible through Python. Thus, you need not know what the core contains. However, if you plan to modify the core, studying also the Fortran documentation is useful.

The graph below show the main dependancies between the Python and Fortran classes and modules. (This is not a full UML diagram, just a schematic presentation.)



4.2 Pysic module

Pysic is an object based calculator for atomistic many-body interactions. It is controlled via several classes: `Pysic`, which defines a calculator for the ASE simulation environment ¹, `Potential`, which defines potentials to be used by `Pysic` for calculating atomic interactions, `Coordinator` which can be used for modifying potentials with bond order factors, and `BondOrderParameters` which are used as a container of parameters for the `Coordinator`.

4.2.1 Classes of the pysic module

Pysic class

This class provides an interface both to the ASE environment and the Fortran core of Pysic through the `pysic_fortran` module. In ASE terms, Pysic is a *calculator*, i.e., an object that calculates forces and energies of given atomistic structures provided as an ASE `Atoms` object.

Pysic is not a fixed potential calculator, where the interactions are determined solely based on the elements present in the system. Instead, Pysic allows and requires the user to specify the interactions governing the system. This is done by providing the calculator with one or several `Potential` objects.

¹ ASE: Comput. Sci. Eng., Vol. 4, 56-66, 2002; <https://wiki.fysik.dtu.dk/ase/>

List of methods

Below is a list of methods in `Pysic`, grouped according to the type of functionality.

Structure handling

- `create_neighbor_lists()` (meant for internal use)
- `get_atoms()`
- `get_neighbor_lists()`
- `neighbor_lists_expanded()` (meant for internal use)
- `set_atoms()`

Potential handling

- `add_potential()`
- `get_individual_cutoffs()`
- `get_potentials()`
- `set_potentials()`

Coulomb handling

- `get_coulomb_summation()`
- `set_coulomb_summation()`

Charge relaxation handling

- `get_charge_relaxation()`
- `set_charge_relaxation()`

Calculations

- `calculate_electronegativities()` (meant for internal use)
- `calculate_energy()` (meant for internal use)
- `calculate_forces()` (meant for internal use)
- `calculate_stress()` (meant for internal use)
- `calculation_required()` (meant for internal use)
- `get_electronegativities()`
- `get_electronegativity_differences()`
- `get_forces()`
- `get_numerical_bond_order_gradient()` (for testing)
- `get_numerical_energy_gradient()` (for testing)
- `get_numerical_electronegativity()` (for testing)
- `get_potential_energy()`

- `get_stress()`

Core

- `core`
- `core_initialization_is_forced()`
- `force_core_initialization()`
- `initialize_fortran_core()`
- `set_core()`
- `update_core_charges()` (meant for internal use)
- `update_core_coordinates()` (meant for internal use)
- `update_core_coulomb()` (meant for internal use)
- `update_core_neighbor_lists()` (meant for internal use)
- `update_core_potential_lists()` (meant for internal use)
- `update_core_supercell()` (meant for internal use)

Full documentation of the Pysic class

class `pysic.calculator.Pysic` (*atoms=None, potentials=None, charge_relaxation=None, coulomb=None, full_initialization=False*)

A calculator class providing the necessary methods for interfacing with ASE.

Pysic is a calculator for evaluating energies and forces for given atomic structures according to the given `Potential` set. Neither the geometry nor the potentials have to be specified upon creating the calculator, as they can be specified or changed later. They are necessary for actual calculation, of course.

Simulation geometries must be defined as `ASE Atoms`. This object contains both the atomistic coordinates and supercell parameters.

Potentials must be defined as a list of `Potential` objects. The total potential of the system is then the sum of the individual potentials.

Parameters:

atoms: `ASE Atoms` object an Atoms object containing the full simulation geometry

potentials: list of `Potential` objects list of potentials for describing interactions

force_initialization: `boolean` If true, calculations always fully initialize the Fortran core. If false, the Pysic tries to evaluate what needs updating by consulting the `core` instance of `CoreMirror`.

add_potential (*potential*)

Add a potential to the list of potentials.

Parameters:

potential: `Potential` object a new potential to describe interactions

calculate_electronegativities ()

Calculates electronegativities.

Calls the Fortran core to calculate forces for the currently assigned structure.

calculate_energy()

Calculates the potential energy.

Calls the Fortran core to calculate the potential energy for the currently assigned structure.

If a link exists to a [ChargeRelaxation](#), it is first made to relax the atomic charges before the forces are calculated.

calculate_forces()

Calculates forces (and the potential part of the stress tensor).

Calls the Fortran core to calculate forces for the currently assigned structure.

If a link exists to a [ChargeRelaxation](#), it is first made to relax the atomic charges before the forces are calculated.

calculate_stress()

Calculates the potential part of the stress tensor (and forces).

Calls the Fortran core to calculate the stress tensor for the currently assigned structure.

calculation_required(*atoms=None, quantities=['forces', 'energy', 'stress', 'electronegativities']*)

Check if a calculation is required.

When forces or energy are calculated, the calculator saves the result in case it is needed several times. This method tells if a wanted quantity is not yet calculated for the current structure and needs to be calculated explicitly. If a list of several quantities is given, the method returns true if any one of them needs to be calculated.

Parameters:

atoms: [ASE Atoms](#) object ignored at the moment

quantities: list of strings list of keywords 'energy', 'forces', 'stress', 'electronegativities'

core = CoreMirror()

An object storing the data passed to the core.

Whenever a `Pysic` calculator alters the Fortran core, it should also modify the `core` object so that it is always a valid representation of the actual core. Then, whenever `Pysic` needs to check if the representation in the core is up to date, it only needs to compare against `core` instead of accessing the Fortran core itself.

core_initialization_is_forced()

Returns true if the core is always fully initialized, false otherwise.

create_neighbor_lists(*cutoffs=None, marginal=0.5*)

Initializes the neighbor lists.

In order to do calculations at reasonable speed, the calculator needs a list of neighbors for each atom. For this purpose, the [ASE NeighborList](#) are used. This method initializes these lists according to the given cutoffs.

Parameters:

cutoffs: list of doubles a list containing the cutoff distance for each atom

marginal: double the skin width of the neighbor list

force_core_initialization(*new_mode*)

Set the core initialization mode.

Parameters:

new_mode: logical true if full initialization is required, false if not

get_atoms()

Returns the [ASE Atoms](#) object assigned to the calculator.

get_charge_relaxation()

Returns the [ChargeRelaxation](#) object connected to the calculator.

get_coulomb_summation()

Returns the Coulomb summation algorithm of this calculator.

get_electronegativities (*atoms=None*)

Returns the electronegativities of atoms.

get_electronegativity_differences (*atoms=None*)

Returns the electronegativity differences of atoms from the average of the entire system.

get_forces (*atoms=None*)

Returns the forces.

If the atoms parameter is given, it will be used for updating the structure assigned to the calculator prior to calculating the forces. Otherwise the structure already associated with the calculator is used.

The calculator checks if the forces have been calculated already via `calculation_required()`. If the structure has changed, the forces are calculated using `calculate_forces()`

Parameters:

atoms: [ASE atoms](#) object the structure for which the forces are determined

get_individual_cutoffs (*scaler=1.0*)

Get a list of maximum cutoffs for all atoms.

For each atom, the interaction with the longest cutoff is found and the associated maximum cutoffs are returned as a list. In case the a list of scaled values are required, the scaler can be adjusted. E.g., scaler = 0.5 will return the cutoffs halved.

Parameters:

scaler: **double** a number for scaling all values in the generated list

get_neighbor_lists()

Returns the [FastNeighborList](#) or [ASE NeighborList](#) object assigned to the calculator.

The neighbor lists are generated according to the given [ASE Atoms](#) object and the [Potential](#) objects of the calculator. Note that the lists are created when the core is set or if the method `create_neighbor_lists()` is called.

get_numerical_bond_order_gradient (*coordinator, atom_index, moved_index, shift=0.001, atoms=None*)

Numerically calculates the gradient of a bond order factor with respect to moving a single particle.

This is for debugging the bond orders.

get_numerical_electronegativity (*atom_index, shift=0.001, atoms=None*)

Numerically calculates the derivative of energy with respect to charging a single particle.

This is for debugging the electronegativities.

get_numerical_energy_gradient (*atom_index, shift=0.0001, atoms=None*)

Numerically calculates the negative gradient of energy with respect to moving a single particle.

This is for debugging the forces.

get_potential_energy (*atoms=None, force_consistent=False*)

Returns the potential energy.

If the `atoms` parameter is given, it will be used for updating the structure assigned to the calculator prior to calculating the energy. Otherwise the structure already associated with the calculator is used.

The calculator checks if the energy has been calculated already via `calculation_required()`. If the structure has changed, the energy is calculated using `calculate_energy()`

Parameters:

atoms: [ASE atoms object](#) the structure for which the energy is determined

force_consistent: `logical` ignored at the moment

get_potentials()

Returns the list of potentials assigned to the calculator.

get_stress (*atoms=None*)

Returns the stress tensor in the format $[\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{yz}, \sigma_{xz}, \sigma_{xy}]$

If the `atoms` parameter is given, it will be used for updating the structure assigned to the calculator prior to calculating the stress. Otherwise the structure already associated with the calculator is used.

The calculator checks if the stress has been calculated already via `calculation_required()`. If the structure has changed, the stress is calculated using `calculate_stress()`

Stress (potential part) and force are evaluated in tandem. Therefore, invoking the evaluation of one automatically leads to the evaluation of the other. Thus, if you have just evaluated the forces, the stress will already be known.

This is because the stress tensor is formally defined as

$$\sigma_{AB} = -\frac{1}{V} \sum_i [m_i(v_i)_A(v_i)_B + (r_i)_A(f_i)_B],$$

where m , v , r , and f are mass, velocity, position and force of atom i , and A , B denote the cartesian coordinates x, y, z . (The minus sign is there just to be consistent with the NPT routines in [ASE](#).) However, if periodic boundaries are used, the absolute coordinates cannot be used (there would be discontinuities at the boundaries of the simulation cell). Instead, the potential energy terms $(r_i)_A(f_i)_B$ must be evaluated locally for pair, triplet, and many body forces using the relative coordinates of the particles involved in the local interactions. These coordinates are only available during the actual force evaluation when the local interactions are looped over. Thus, calculating the stress requires doing the full force evaluation cycle. On the other hand, calculating the stress is not a great effort compared to the force evaluation, so it is convenient to evaluate the stress always when the forces are evaluated.

Parameters:

atoms: [ASE atoms object](#) the structure for which the stress is determined

initialize_fortran_core()

Fully initializes the Fortran core, creating the atoms, supercell, potentials, and neighbor lists.

neighbor_lists_expanded (*cutoffs*)

Check if the cutoffs have been expanded.

If the cutoffs have been made longer than before, the neighbor lists have to be recalculated. This method checks the individual cutoffs of all atoms to check if the cutoffs have changed.

Parameters:

cutoffs: `list of doubles` new cutoffs

set_atoms (*atoms=None*)

Assigns the calculator with the given structure.

This method is always called when any method is given the atomic structure as an argument. If the argument is missing or None, nothing is done. Otherwise a copy of the given structure is saved (according to the instructions in [ASE API](#).)

If a structure is already in memory and it is different to the given one (as compared with `__ne__`), it is noted that all quantities are unknown for the new system. If the structure is the same as the one already known, nothing is done. This is because if one wants to access the energy of forces of the same system repeatedly, it is unnecessary to always calculate them from scratch. Therefore the calculator saves the computed values along with a flag stating that the values have been computed.

Parameters:

atoms: [ASE atoms object](#) the structure to be calculated

set_charge_relaxation (*charge_relaxation*)

Add a charge relaxation algorithm to the calculator.

If a charge relaxation scheme has been added to the `Pysic` calculator, it will be automatically asked to do the charge relaxation before the calculation of energies or forces via `charge_relaxation()`.

It is also possible to pass the `Pysic` calculator to the `ChargeRelaxation` algorithm without creating the opposite link using `set_calculator()`. In that case, the calculator does not automatically relax the charges, but the user can manually trigger the relaxation with `charge_relaxation()`.

If you wish to remove automatic charge relaxation, just call this method again with None as argument.

Parameters:

charge_relaxation: [ChargeRelaxation object](#) the charge relaxation algorithm

set_core ()

Sets up the Fortran core for calculation.

If the core is not initialized, if the number of atoms has changed, or if full initialization is forced, the core is initialized from scratch. Otherwise, only the atomic coordinates and momenta are updated. Potentials, neighbor lists etc. are also updated if they have been edited.

set_coulomb_summation (*coulomb*)

Set the Coulomb summation algorithm for the calculator.

If a Coulomb summation algorithm is set, the Coulomb interactions between all charged atoms are evaluated automatically during energy and force evaluation. If not, the charges do not directly interact.

Parameters:

coulomb: [CoulombSummation](#) the Coulomb summation algorithm

set_cutoffs (*cutoffs*)

Copy and save the list of individual cutoff radii.

Parameters:

cutoffs: list of doubles new cutoffs

set_potentials (*potentials*)

Assign a list of potentials to the calculator.

Parameters:

potentials: list of [Potential](#) objects a list of potentials to describe interactions

update_core_charges ()

Updates atomic charges in the core.

update_core_coordinates()

Updates the positions and momenta of atoms in the Fortran core.

The core must be initialized and the number of atoms must match. Upon the update, it is automatically checked if the neighbor lists should be updated as well.

update_core_coulomb()

Updates the Coulomb summation parameters in the Fortran core.

update_core_neighbor_lists()

Updates the neighbor lists in the Fortran core.

If uninitialized, the lists are created first via `create_neighbor_lists()`.

update_core_potential_lists()

Initializes the potential lists.

Since one often runs `Pysic` with a set of potentials, the core pre-analyzes which potentials affect each atom and saves a list of such potentials for every particle. This method asks the core to generate these lists.

update_core_potentials()

Generates potentials for the Fortran core.

update_core_supercell()

Updates the supercell in the Fortran core.

Potential class

This class defines an atomistic potential to be used by `Pysic`. An interaction between two or more particles can be defined and the targets of the interaction can be specified by chemical symbol, tag, or index. The available types of potentials are always inquired from the Fortran core to ensure that any changes made to the core are automatically reflected in the Python interface.

There are a number of utility functions in `pysic` for inquiring the keywords and other data needed for creating the potentials. For example:

- Inquire the names of available potentials: `list_valid_potentials()`
- Inquire the names of parameters for a potential: `names_of_parameters()`
- Ask for a short description of a potential: `description_of_potential()`

Cutoffs

Many potentials decay towards zero in infinity, but in a numeric simulation they are cut at a finite range as specified by a cutoff radius. However, if the potential is not exactly zero at this range, a discontinuity will be introduced. It is possible to avoid this by including a smoothening factor in the potential to force a decay to zero in a finite interval:

$$\tilde{V}(r) = f(r)V(r),$$

where the smoothening factor is (for example)

$$f(r) = \begin{cases} 1, & r < r_{\text{soft}} \\ \frac{1}{2} \left(1 + \cos \pi \frac{r - r_{\text{soft}}}{r_{\text{hard}} - r_{\text{soft}}} \right), & r_{\text{soft}} < r < r_{\text{hard}} \\ 0, & r > r_{\text{hard}} \end{cases}.$$

`Pysic` allows one to specify both a hard and a soft cutoff for all potentials to include such a smooth cutoff. If no soft cutoff is given or it is zero (or equal to the hard cutoff), no smoothening is applied.

List of currently available potentials

Below is a list of potentials currently implemented.

- *Constant potential*
- *Constant force potential*
- *Power decay potential*
- *Harmonic potential*
- *Lennard-Jones potential*
- *Buckingham potential*
- *Charge dependent exponential potential*
- *Bond bending potential*
- *Dihedral angle potential*

Constant potential 1-body potential defined as

$$V(\mathbf{r}) = V,$$

i.e., a constant potential.

A constant potential is of course irrelevant in force calculation since the gradient is zero. However, one can add a `BondOrderParameters` bond order factor with the potential to create essentially a bond order potential.

The constant potential can also be used for assigning an energy offset which depends on the number of atoms of required types.

Keywords:

```
>>> names_of_parameters('constant')
['V']
```

Fortran routines:

- `create_potential_characterizer_constant_potential()`
- `evaluate_energy_constant_potential()`

Constant force potential 1-body potential defined as

$$V(\mathbf{r}) = -\mathbf{F} \cdot \mathbf{r},$$

i.e., a constant force \mathbf{F} .

Keywords:

```
>>> names_of_parameters('force')
['Fx', 'Fy', 'Fz']
```

Fortran routines:

- `create_potential_characterizer_constant_force()`
- `evaluate_energy_constant_force()`
- `evaluate_force_constant_force()`

Power decay potential 2-body interaction defined as

$$V(r) = \varepsilon \left(\frac{a}{r} \right)^n$$

where ε is an energy scale constant, a is a length scale constant or lattice parameter, and n is an exponent.

There are many potentials defined in Pysic which already incorporate power law terms, and so this potential is often not needed. Still, one can build, for instance, the *Lennard-Jones potential* by stacking two power decay potentials. Note that n should be large enough for the potential to be sensible. Especially if one is creating a Coulomb potential with $n = 1$, one should not define the potential through `Potential` objects, which are directly summed, but with the `CoulombSummation` class instead.

Keywords:

```
>>> names_of_parameters('power')
['epsilon', 'a', 'n']
```

Fortran routines:

- `create_potential_characterizer_power()`
- `evaluate_energy_power()`
- `evaluate_force_power()`

Harmonic potential 2-body interaction defined as

$$V(r) = \frac{1}{2}k(r - R_0)^2 - \frac{1}{2}k(r_{\text{cut}} - R_0)^2,$$

where k is a spring constant, R_0 is the equilibrium distance, and r_{cut} is the potential cutoff. The latter term is a constant whose purpose is to remove the discontinuity at cutoff.

Keywords:

```
>>> names_of_parameters('spring')
['k', 'R_0']
```

Fortran routines:

- `create_potential_characterizer_spring()`
- `evaluate_energy_spring()`
- `evaluate_force_spring()`

Lennard-Jones potential 2-body interaction defined as

$$V(r) = \varepsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right],$$

where ε is an energy constant defining the depth of the potential well and σ is the distance where the potential changes from positive to negative in the repulsive region.

Keywords:

```
>>> names_of_parameters('LJ')
['epsilon', 'sigma']
```

Fortran routines:

- `create_potential_characterizer_LJ()`

- `evaluate_energy_LJ()`
- `evaluate_force_LJ()`

Buckingham potential 2-body interaction defined as

$$V(r) = Ae^{-\frac{r}{\sigma}} - C \left(\frac{\sigma}{r} \right)^6$$

where σ is a length scale constant, and A and C are the energy scale constants for the exponential and van der Waals parts, respectively.

Keywords:

```
>>> names_of_parameters('Buckingham')
['A', 'C', 'sigma']
```

Fortran routines:

- `create_potential_characterizer_buckingham()`
- `evaluate_energy_buckingham()`
- `evaluate_force_buckingham()`

Charge dependent exponential potential 2-body interaction defined as

$$V(r, q) = \varepsilon_{ij} \exp \left(-\zeta_{ij} r + \frac{\xi_i D_i(q_i) + \xi_j D_j(q_j)}{2} \right)$$

$$D_i(q) = R_{i,\max} + |\beta_i (Q_{i,\max} - q)|^{\eta_i}$$

$$\beta_i = \frac{(R_{i,\min} - R_{i,\max})^{\frac{1}{\eta_i}}}{Q_{i,\max} - Q_{i,\min}}$$

$$\eta_i = \frac{\ln \frac{R_{i,\max}}{R_{i,\max} - R_{i,\min}}}{\ln \frac{Q_{i,\max}}{Q_{i,\max} - Q_{i,\min}}},$$

where ε is an energy scale constant, ζ is a length decay constant, ξ_i are charge decay constants, and $R_{i,\min/\max}$ and $Q_{i,\min/\max}$ are the changes in valence radii and charge, respectively, of the ions for the minimum and maximum charge. $D_i(q)$ is the effective atomic radius for the charge q .

Keywords:

```
>>> names_of_parameters('exponential')
['epsilon', 'zeta',
 'Rmax1', 'Rmin1', 'Qmax1', 'Qmin1',
 'Rmax2', 'Rmin2', 'Qmax2', 'Qmin2',
 'xi1', 'xi2']
```

Fortran routines:

- `create_potential_characterizer_charge_exp()`
- `evaluate_energy_charge_exp()`
- `evaluate_force_charge_exp()`
- `evaluate_electronegativity_charge_exp()`

Bond bending potential 3-body interaction defined as

$$V(\theta) = \frac{k}{2}(\cos \theta - \cos \theta_0)^2,$$

where k is an angular spring constant, θ is an angle defined by three points in space (atomic positions) and θ_0 is the equilibrium angle. The potential therefore describes an angular spring force related to bending of bonds.

Keywords:

```
>>> names_of_parameters('bond_bend')
['k', 'theta_0']
```

Three bodies form a triangle and so there are three possible angles the potential could bend. To remove this ambiguity, the angle is defined so that as the potential is given a list of targets, the middle target is considered to be at the tip of the angle.

Example:

```
>>> pot = Potential('bond_bend')
>>> pot.set_symbols(['H', 'O', 'H'])
```

This creates a potential for H-O-H angles, but not for H-H-O angles.

Also remember that the bond bending potential does not include any actual bonding potential between particles - it only generates an angular force component. It must be coupled with other potentials to build a full bonding potential.

Fortran routines:

- `create_potential_characterizer_bond_bending()`
- `evaluate_energy_bond_bending()`
- `evaluate_force_bond_bending()`

Dihedral angle potential 4-body interaction defined as

$$V(r) = \frac{k}{2}(\cos \theta - \cos \theta_0)^2$$

where k is a spring constant and θ is the dihedral angle, with θ_0 denoting the equilibrium angle. (So, this is the cosine harmonic variant of the dihedral angle potential.)

For an atom chain 1-2-3-4 the dihedral angle is the angle between the bonds 1-2 and 3-4 when projected on the plane perpendicular to the bond 2-3. In other words, it is the torsion angle of the bond chain. If we write $\mathbf{r}_{ij} = \mathbf{R}_j - \mathbf{R}_i$, where \mathbf{R}_i is the coordinate vector of the atom i , the angle is given by

$$\begin{aligned}\cos \theta &= \frac{\mathbf{p} \cdot \mathbf{p}'}{|\mathbf{p}| |\mathbf{p}'|} \\ \mathbf{p} &= -\mathbf{r}_{12} + \frac{\mathbf{r}_{12} \cdot \mathbf{r}_{23}}{|\mathbf{r}_{23}|^2} \mathbf{r}_{23} \\ \mathbf{p}' &= \mathbf{r}_{34} - \frac{\mathbf{r}_{34} \cdot \mathbf{r}_{23}}{|\mathbf{r}_{23}|^2} \mathbf{r}_{23}\end{aligned}$$

Keywords:

```
>>> names_of_parameters('dihedral')
['k', 'theta_0']
```

Fortran routines:

- `create_potential_characterizer_dihedral()`

- `evaluate_energy_dihedral()`
- `evaluate_force_dihedral()`

List of methods

Below is a list of methods in `Potential`, grouped according to the type of functionality.

Interaction handling

- `get_cutoff()`
- `get_cutoff_margin()`
- `get_parameter_names()`
- `get_parameter_value()`
- `get_parameter_values()`
- `get_potential_type()`
- `get_soft_cutoff()`
- `set_cutoff()`
- `set_cutoff_margin()`
- `set_parameter_value()`
- `set_parameter_values()`
- `set_soft_cutoff()`

Coordinator handling

- `get_coordinator()`
- `set_coordinator()`

Target handling

- `accepts_target_list()`
- `add_indices()`
- `add_symbols()`
- `add_tags()`
- `get_different_indices()`
- `get_different_symbols()`
- `get_different_tags()`
- `get_indices()`
- `get_number_of_targets()`
- `get_symbols()`
- `get_tags()`
- `set_indices()`

- `set_symbols()`
- `set_tags()`

Description

- `describe()`

Full documentation of the Potential class

class `pysic.Potential` (*potential_type*, *symbols=None*, *tags=None*, *indices=None*, *parameters=None*, *cutoff=0.0*, *cutoff_margin=0.0*, *coordinator=None*)
 Class for representing a potential.

Several types of potentials can be defined by specifying the type of the potential as a keyword. The potentials contain a host of parameters and information on what types of particles they act on. To view a list of available potentials, use the method `list_valid_potentials()`.

A potential may be a pair or many-body potential: here, the bodies a potential acts on are called targets. Thus specifying the number of targets of a potential also determines if the potential is a many-body potential.

A potential may be defined for atom types or specifically for certain atoms. These are specified by the symbols, tags, and indices. Each of these should be either 'None' or a list of lists of n values where n is the number of targets. For example, if:

```
indices = [[0, 1], [1, 2], [2, 3]]
```

the potential will be applied between atoms 0 and 1, 1 and 2, and 2 and 3.

Parameters:

symbols: list of string the chemical symbols (elements) on which the potential acts

tags: integer atoms with specific tags on which the potential acts

indices: list of integers atoms with specific indices on which the potential acts

potential_type: string a keyword specifying the type of the potential

parameters: list of doubles a list of parameters for characterizing the potential; their meaning depends on the type of potential

cutoff: double the maximum atomic separation at which the potential is applied

accepts_target_list (*targets*)

Tests whether a list is suitable as a list of targets, i.e., symbols, tags, or indices and returns True or False accordingly.

A list of targets should be of the format:

```
targets = [[a, b], [c, d]]
```

where the length of the sublists must equal the number of targets.

It is not tested that the values contained in the list are valid.

Parameters:

targets: list of strings or integers a list whose format is checked

add_indices (*indices*)

Adds the given indices to the list of indices.

Parameters:

indices: list of integers list of additional indices on which the potential acts

add_symbols (*symbols*)

Adds the given symbols to the list of symbols.

Parameters:

symbols: list of strings list of additional symbols on which the potential acts

add_tags (*tags*)

Adds the given tags to the list of tags.

Parameters:

tags: list of integers list of additional tags on which the potential acts

describe ()

Prints a short description of the potential using the method `describe_potential()`.

get_coordinator ()

Returns the Coordinator.

get_cutoff ()

Returns the cutoff.

get_cutoff_margin ()

Returns the margin for a smooth cutoff.

get_different_indices ()

Returns a list containing each index the potential affects once.

get_different_symbols ()

Returns a list containing each symbol the potential affects once.

get_different_tags ()

Returns a list containing each tag the potential affects once.

get_indices ()

Return a list of indices on which the potential acts on.

get_number_of_targets ()

Returns the number of targets.

get_parameter_names ()

Returns a list of the names of the parameters of the potential.

get_parameter_value (*param_name*)

Returns the value of the given parameter.

Parameters:

param_name: string name of the parameter

get_parameter_values ()

Returns a list containing the current parameter values of the potential.

get_potential_type ()

Returns the keyword specifying the type of the potential.

get_soft_cutoff ()

Returns the lower limit for a smooth cutoff.

get_symbols()

Return a list of the chemical symbols (elements) on which the potential acts on.

get_tags()

Return the tags on which the potential acts on.

set_coordinator(*coordinator*)

Sets a new Coordinator.

set_cutoff(*cutoff*)

Sets the cutoff to a given value.

This method affects the hard cutoff. For a detailed explanation on how to define a soft cutoff, see [set_cutoff_margin\(\)](#).

Parameters:

cutoff: double new cutoff for the potential

set_cutoff_margin(*margin*)

Sets the margin for smooth cutoff to a given value.

Many potentials decay towards zero in infinity, but in a numeric simulation they are cut at a finite range as specified by the cutoff radius. If the potential is not exactly zero at this range, a discontinuity will be introduced. It is possible to avoid this by including a smoothening factor in the potential to force a decay to zero in a finite interval.

This method defines the decay interval $r_{\text{hard}} - r_{\text{soft}}$. Note that if the soft cutoff value is made smaller than 0 or larger than the hard cutoff value an [InvalidPotentialError](#) is raised.

Parameters:

margin: double The new cutoff margin

set_indices(*indices*)

Sets the list of indices to equal the given list.

Parameters:

indices: list of integers list of integers on which the potential acts

set_parameter_value(*parameter_name*, *value*)

Sets a given parameter to the desired value.

Parameters:

parameter_name: string name of the parameter

value: double the new value of the parameter

set_parameter_values(*values*)

Sets the numeric values of all parameters.

Parameters:

values: list of doubles list of values to be assigned to parameters

set_parameters(*values*)

Sets the numeric values of all parameters.

Equivalent to [set_parameter_values\(\)](#).

Parameters:

values: list of doubles list of values to be assigned to parameters

set_soft_cutoff (*cutoff*)

Sets the soft cutoff to a given value.

For a detailed explanation on the meaning of a soft cutoff, see `set_cutoff_margin()`. Note that actually the cutoff margin is recorded, so changing the hard cutoff (see `set_cutoff()`) will also affect the soft cutoff.

Parameters:

cutoff: double The new soft cutoff

set_symbols (*symbols*)

Sets the list of symbols to equal the given list.

Parameters:

symbols: list of strings list of element symbols on which the potential acts

set_tags (*tags*)

Sets the list of tags to equal the given list.

Parameters:

tags: list of integers list of tags on which the potential acts

CoulombSummation class

If a periodic system contains charges interacting via the $\frac{1}{r}$ Coulomb potential, direct summation of the interactions

$$E = \sum_{(i,j)} \frac{1}{4\pi\epsilon_0} \frac{q_i q_j}{r_{ij}}, \quad (4.1)$$

where the sum is over pairs of charges q_i, q_j (charges of the entire system, not just the simulation cell) and the distance between the charges is $r_{ij} = |\mathbf{r}_j - \mathbf{r}_i|$, does not work in general because the sum (4.1) converges very slowly (it actually converges only conditionally). Therefore truncating the sum may lead to severe errors. More advanced techniques must be used in order to accurately evaluate such sums.

This class represents the algorithms used for evaluating the $1/r$ sums. It wraps the summation parameters and activates the summation of Coulomb interactions. If an instance of `CoulombSummation` is given to the `Pysic` calculator, Coulomb interactions between all charged atoms are automatically included in the calculations, regardless of possible `Potential` potentials the calculator may also contain. Otherwise the charges do not directly interact. This is due to two reasons: First, the direct Coulomb interaction is usually always required and it is convenient that it is easily enabled. Second, the specific potentials described by `Potential` are evaluated by direct summation and so the Coulomb summation is separate also on algorithm level in the core.

List of currently available summation algorithms

Below is a list of summation algorithms currently implemented.

Ewald summation The standard technique for overcoming the problem of summing long ranged periodic potentials is the so called Ewald summation method. The idea is to split the long ranged and singular Coulomb potential to a short ranged singular and long ranged smooth parts, and calculate the long ranged part in reciprocal space via Fourier transformations. This is possible (for a smooth potential) since the system is periodic and the same supercell repeats infinitely in all directions. In practice the calculation can be done by adding (and subtracting) Gaussian charge densities over the point charges to screen the potential in real space. That is, the original charge density $\rho(\mathbf{r}) = \sum_i q_i \delta(\mathbf{r} - \mathbf{r}_i)$ is split by

$$\rho(\mathbf{r}) = \rho_s(\mathbf{r}) + \rho_l(\mathbf{r}) \quad (4.2)$$

$$\rho_s(\mathbf{r}) = \sum_i [q_i \delta(\mathbf{r} - \mathbf{r}_i) - q_i G_\sigma(\mathbf{r} - \mathbf{r}_i)] \quad (4.3)$$

$$\rho_l(\mathbf{r}) = \sum_i q_i G_\sigma(\mathbf{r} - \mathbf{r}_i) \quad (4.4)$$

$$G_\sigma(\mathbf{r}) = \frac{1}{(2\pi\sigma^2)^{3/2}} \exp\left(-\frac{|\mathbf{r}|^2}{2\sigma^2}\right) \quad (4.5)$$

Here ρ_l generates a long range interaction since at large distances the Gaussian densities G_σ appear the same as point charges ($\lim_{\sigma/r \rightarrow 0} G_\sigma(\mathbf{r}) = \delta(\mathbf{r})$). Since the charge density is smooth, so will be the potential it creates. The density ρ_s exhibits short ranged interactions for the same reason: At distances longer than the width of the Gaussians the point charges are screened by the Gaussians which exactly cancel them ($\lim_{\sigma/r \rightarrow 0} \delta(\mathbf{r}) - G_\sigma(\mathbf{r}) = 0$).

The short ranged interactions are directly calculated in real space

$$E_s = \frac{1}{4\pi\epsilon_0} \int \frac{\rho_s(\mathbf{r})\rho_s(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} d^3r d^3r' \quad (4.6)$$

$$= \frac{1}{4\pi\epsilon_0} \sum_{(i,j)} \frac{q_i q_j}{r_{ij}} \operatorname{erfc}\left(\frac{r_{ij}}{\sigma\sqrt{2}}\right). \quad (4.7)$$

The complementary error function $\operatorname{erfc}(r) = 1 - \operatorname{erf}(r) = 1 - \frac{2}{\sqrt{\pi}} \int_0^r e^{-t^2/2} dt$ makes the sum converge rapidly as $\frac{r_{ij}}{\sigma} \rightarrow \infty$.

The long ranged interaction

$$E_l = \frac{1}{4\pi\epsilon_0} \int \frac{\rho_l(\mathbf{r})\rho_l(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} d^3r d^3r'$$

can be calculated in reciprocal space by Fourier transformation. The result is

$$E_l = \frac{1}{2V\epsilon_0} \sum_{\mathbf{k} \neq 0} \frac{e^{-\sigma^2 k^2/2}}{k^2} |S(\mathbf{k})|^2 - \frac{1}{4\pi\epsilon_0} \frac{1}{\sqrt{2\pi}\sigma} \sum_i q_i^2 \quad (4.8)$$

$$S(\mathbf{k}) = \sum_i q_i e^{i\mathbf{k} \cdot \mathbf{r}_i} \quad (4.9)$$

The first sum in E_l runs over the reciprocal lattice $\mathbf{k} = k_1 \mathbf{b}_1 + k_2 \mathbf{b}_2 + k_3 \mathbf{b}_3$ where \mathbf{b}_i are the vectors spanning the reciprocal cell ($[\mathbf{b}_1 \mathbf{b}_2 \mathbf{b}_3] = ([\mathbf{v}_1 \mathbf{v}_2 \mathbf{v}_3]^{-1})^T$ where \mathbf{v}_i are the real space cell vectors). The latter sum is the self energy of each point charge in the potential of the particular Gaussian that screens the charge, and the sum runs over all charges in the supercell spanning the periodic system. (The self energy must be removed because it is present in the first sum even though when evaluating the potential at the position of a charge due to the other charges, no screening Gaussian function should be placed over the charge itself.) Likewise the sum in the structure factor $S(\mathbf{k})$ runs over all charges in the supercell.

The total energy is then the sum of the short and long range energies

$$E = E_s + E_l.$$

If the system carries a net charge, the total Coulomb potential of the infinite periodic system is infinite. Excess charge can be neutralized by imposing a uniform background charge of opposite sign, which results in the correction term

$$E_c = -\frac{\sigma^2}{4V\varepsilon_0} \left| \sum_i q_i \right|^2.$$

This correction is applied automatically.

Forces are obtained as the gradient of the total energy. For atom α , the force is

$$\mathbf{F}_\alpha = -\nabla_\alpha E = -\nabla_\alpha E_s - \nabla_\alpha E_l.$$

(There is no contribution from E_c .) The short ranged interactions are easily calculated in real space

$$-\nabla_\alpha E_s = \frac{q_\alpha}{4\pi\varepsilon_0} \sum_j q_j \left[\operatorname{erfc} \left(\frac{r_{\alpha j}}{\sigma\sqrt{2}} \right) \frac{1}{r_{\alpha j}^2} + \frac{1}{\sigma} \sqrt{\frac{2}{\pi}} \exp \left(-\frac{r_{\alpha j}^2}{2\sigma^2} \right) \frac{1}{r_{\alpha j}} \right] \hat{r}_{\alpha j},$$

where $\hat{r}_{\alpha j} = \mathbf{r}_{\alpha j}/r_{\alpha j}$ is the unit vector pointing from atom α to j . The long range forces are obtained by differentiating the structure factor

$$-\nabla_\alpha E_l = -\frac{1}{2V\varepsilon_0} \sum_{\mathbf{k} \neq 0} \frac{e^{-\sigma^2 k^2/2}}{k^2} 2\operatorname{Re}[S^*(\mathbf{k}) \nabla_\alpha S(\mathbf{k})] \quad (4.10)$$

$$\nabla_\alpha S(\mathbf{k}) = q_\alpha \mathbf{k} (-\sin \mathbf{k} \cdot \mathbf{r}_\alpha + i \cos \mathbf{k} \cdot \mathbf{r}_\alpha). \quad (4.11)$$

List of methods

Below is a list of methods in `CoulombSummation`, grouped according to the type of functionality.

Initialization

- `initialize_parameters()` (meant for internal use)
- `get_summation()`
- `set_summation()`
- `summation_modes`
- `summation_parameter_descriptions`
- `summation_parameters`

Parameter handling

- `get_parameters()`
- `set_parameter_value()`
- `set_parameter_values()`
- `set_parameters()`

Miscellaneous

- `get_realspace_cutoff()`
- `get_scaling_factors()`
- `set_scaling_factors()`

Full documentation of the CoulombSummation class

class `pysic.CoulombSummation` (*method='ewald', parameters=None, scaler=None*)

Class for representing a collection of parameters for evaluating Coulomb potentials.

Summing $1/r$ potentials in periodic systems requires more advanced techniques than just direct summation of pair interactions. The standard method for evaluating these kinds of potentials is through Ewald summation, where the long range part of the potential is evaluated in reciprocal space.

Instances of this class are used for wrapping the parameters controlling the summations. Passing such an instance to the `Pysic` calculator activates the evaluation of Coulomb interactions.

Currently, only Ewald summation is available as a calculation method.

Parameters:

method: **string** keyword specifying the method of summation

parameters: **list of doubles** numeric values of summation parameters

scaler: **list of doubles** numeric values for scaling the atomic charges in summation

get_parameters ()

Returns a list containing the numeric values of the parameters.

get_realspace_cutoff ()

Returns the real space cutoff.

get_scaling_factors ()

Returns the list of scaling parameters for atomic charges.

get_summation ()

Returns the mode of summation.

initialize_parameters ()

Creates a dictionary of parameters and initializes all values to 0.0.

set_parameter_value (*parameter_name, value*)

Sets a given parameter to the desired value.

Parameters:

parameter_name: **string** name of the parameter

value: **double** the new value of the parameter

set_parameter_values (*parameters*)

Sets the numeric values for all parameters.

Parameters:

parameters: **list of doubles** list of values to be assigned to parameters

set_parameters (*parameters*)

Sets the numeric values for all parameters.

Equivalent to `set_parameter_values()`

Parameters:

parameters: **list of doubles** list of values to be assigned to parameters

set_scaling_factors (*scaler*)

Set the list of scaling parameters for atomic charges.

Parameters:

scaler: list of doubles the list of scaling factors

set_summation (*method*)

Sets the summation method.

The method also creates a dictionary of parameters initialized to 0.0 by invoking `initialize_parameters()`.

Parameters:

method: string a keyword specifying the mode of summation

summation_modes = ['ewald']

Names of the summation methods. These are keywords used for setting up the summation algorithms.

summation_parameter_descriptions = {'ewald': ['real space cutoff radius', 'reciprocal space cutoff radius', 'ewa

Short descriptions of the parameters of the summation algorithm.

summation_parameters = {'ewald': ['real_cutoff', 'k_cutoff', 'sigma', 'epsilon']}]

Names of the parameters of the summation algorithm.

Coordinator class

Coordinator is short for 'Coordination Calculator'.

This class provides a utility for calculating and storing bond order factors needed for bond order or Tersoff-like potentials. Here, bond order refers roughly to the number of neighbors of an atom, however, the bond order factors may depend also on the atomic distances, angles and other local geometric factors.

To use a `Coordinator`, one must pass it first to a `Potential` object, which is further given to a `Pysic` calculator. Then, one can use the calculator to calculate forces or just the bond order factors. When a `Coordinator` is added to a `Potential`, the potential is multiplied by the bond order factors as defined by the Coordinator.

Bond order potentials

A bond order factor can be added to any `Potential` object. This means that the potential is multiplied by the bond order factor. The factors are always defined by atom, but for a two and many body potentials the average is applied. To put in other words, if we have, say, a three-body potential

$$V = \sum_{(i,j,k)} v_{ijk},$$

where the sum goes over all atom triplets (i,j,k), adding a bond order factor 'b' will modify this to

$$V = \sum_{(i,j,k)} \frac{1}{3} (b_i + b_j + b_k) v_{ijk}.$$

The corresponding modified force (acting on atom alpha) would be

$$F_\alpha = -\nabla_\alpha V = - \sum_{(i,j,k)} \frac{1}{3} (\nabla_\alpha b_i + \nabla_\alpha b_j + \nabla_\alpha b_k) v_{ijk} + \sum_{(i,j,k)} \frac{1}{3} (b_i + b_j + b_k) f_{\alpha,ijk}.$$

where

$$f_{\alpha,ijk} = -\nabla_\alpha v_{ijk}$$

is the gradient of the unmodified potential.

Note that since the bond factor of an atom usually depends on its whole neighborhood, moving a neighbor of an atom may change the bond factors. In other words, the gradients

$$\nabla_{\alpha} b_i + \nabla_{\alpha} b_j + \nabla_{\alpha} b_k$$

can be non-zero for values of alpha other than i, j, k. Thus adding a bond factor to a potential effectively increases the number of bodies in the interaction.

Parameter wrapping

Bond order factors are defined by atomic elements (chemical symbols). Unlike potentials, however, they may incorporate different parameters and cutoffs for different elements and in addition, they may contain parameters separately for single elements, pairs of elements, element triplets etc. Due to this, a bond order factor can contain plenty of parameters.

To ease the handling of all the parameters, a wrapper class `BondOrderParameters` is defined. A single instance of this class defines the type of bond order factor and contains the cutoffs and parameters for one set of elements. The `Coordinator` object then collects these parameters in one bundle.

The bond order types and all associated parameters are explained in the documentation of `BondOrderParameters`.

Bond order mixing

In general, bond order factors are of the form

$$b_i = s_i \left(\sum_{(i,j,\dots)} c_{ij\dots} \right)$$

where $c_{ij\dots}$ are local environment contributors and s_i is a per-atom scaling function. For example, if one would define a factor

$$b_i = 1 + \sum_{(i,j)} f(r_{ij}),$$

then

$$\begin{aligned} c_{ij} &= f(r_{ij}) \\ s_i(x) &= 1 + x. \end{aligned}$$

When bond order factors are evaluated, the sums $\sum_{(i,j,\dots)} b_{ij\dots}$ are always calculated first and only then the scaling s_i is applied atom-by-atom.

When a `Coordinator` contains several `BondOrderParameters`:

```
>>> crd = pysic.Coordinator( [ bond1, bond2, bond3 ] )
```

they are all added together in the bond order sums $\sum_{(i,j,\dots)} b_{ij\dots}$. Mixing different types of bond order factors is possible but not recommended as the results may be unexpected.

The scaling is always carried out at most only once per atom. This is done as follows. The list of bond order parameters is searched for a parameter set which requires scaling and which contains 1-body parameters for the correct element. (That is, the first atomic symbol of the list of targets of the parameter must equal the element of the atom for which the scaling is done.) Once such parameters are found, they are used for scaling and the rest of the parameters are ignored. In practice this means that the first applicable set of parameters in the list of `BondOrderParameters` in the `Coordinator` is used.

Because of this behaviour, the default scaling can be overridden as shown in the following example.

Let's say we wish to create a potential to bias the coordination number of Cu-O bonds of Cu atoms, n , according to

$$V(n) = \varepsilon \frac{\Delta N}{1 + \exp(\gamma \Delta N)}$$
$$\Delta N = C(n - N).$$

In general, this type of a potential tries to push n towards N , a given parameter.

We can define this in pysic by overriding the scaling of the coordination bond order factor.:

```
>>> bond_sum = pysic.BondOrderParameters( 'neighbors', cutoff = 4.0,
...                                     cutoff_margin = 1.0,
...                                     symbols = [['Cu', 'O']] )
>>> bond_scale = pysic.BondOrderParameters( 'c_scale', symbols = ['Cu'],
...                                     parameters=[epsilon,
...                                     N,
...                                     C,
...                                     gamma] )
>>> crd = pysic.Coordinator( [bond_scale, bond_sum] )
>>> pot = pysic.Potential( 'constant', symbols = ['Cu'],
...                       parameters = [1.0], coordinator = crd )
```

In the final step, the `Coordinator` is attached to a `Potential` with a constant value of 1.0. Since the result is a product between the bond order factor and the potential, the resulting potential is just the bond order factor.

List of methods

Below is a list of methods in `Coordinator`, grouped according to the type of functionality.

Parameter handling

- `add_bond_order_parameters()`
- `set_bond_order_parameters()`
- `get_bond_order_parameters()`

Coordination and bond order

- `calculate_bond_order_factors()`
- `get_bond_order_factors()`
- `get_bond_order_gradients()`
- `get_bond_order_gradients_of_factor()`

Miscellaneous

- `get_group_index()` (meant for internal use)
- `set_group_index()` (meant for internal use)

Full documentation of the Coordinator class

class `pysic.Coordinator` (*bond_order_parameters=None*)

Class for representing a calculator for atomic coordination numbers and bond order factors.

Pysic can utilise ‘ Tersoff-like ’ potentials which are locally scaled according to bond order factors, related to the number of neighbors of each atom. The coordination calculator keeps track of updating the bond order factors and holds the parameters for calculating the values.

When calculating forces also the derivatives of the coordination numbers are needed. Coordination numbers may be used repeatedly when calculating energies and forces even within one evaluation of the forces and therefore they are stored by the calculator. Derivatives are not stored since storing them could potentially require an $N \times N$ matrix, where N is the number of particles.

The calculation of coordination is an operation on the geometry, not the complete physical system including the interactions, and so one can define coordination calculators as standalone objects as well. They always operate on the geometry currently allocated in the core.

Parameters:

bond_order_parameters: list of `BondOrderParameters` objects Parameters for calculating bond order factors.

add_bond_order_parameters (*params*)

Adds the given parameters to this Coordinator.

Parameters:

params: `BondOrderCoordinator` new bond order parameters

calculate_bond_order_factors ()

Recalculates the bond order factors for all atoms and stores them.

Similarly to coordination numbers (`calculate_coordination()`), this method only calculates the factors and stores them but does not return them.

get_bond_order_factors ()

Returns an array containing the bond order factors of all atoms.

This method does not calculate the bond order factors but returns the precalculated array.

get_bond_order_gradients (*atom_index*)

Returns an array containing the gradients of bond order factors of all atoms with respect to moving one atom.

Parameters:

atom_index: integer the index of the atom the position of which is being differentiated with

get_bond_order_gradients_of_factor (*atom_index*)

Returns an array containing the gradients of the bond order factor of one atom with respect to moving any atom.

Parameters:

atom_index: integer the index of the atom the position of which is being differentiated with

get_bond_order_parameters ()

Returns the bond order parameters of this Coordinator.

get_group_index ()

Returns the group index of the Coordinator.

set_bond_order_parameters (*params*)

Assigns new bond order parameters to this Coordinator.

Parameters:

params: **BondOrderCoordinator** new bond order parameters

set_group_index (*index*)

Sets an index for the coordinator object.

In the fortran core, bond order parameters are calculated by bond order parameters. Since a coordinator contains many, they are grouped to a coordinator via a grouping index when the core is initialized. This method is meant to be used for telling the Coordinator of this index. That allows the bond orders can be calculated by calling the Coordinator itself, since the index tells which bond parameters in the core are needed.

Parameters:

index: **integer** an index for grouping bond order parameters in the core

BondOrderParameters class

This class defines a set of parameters for a bond order factor, to be used in conjunction with the `Coordinator` class.

Similarly to the potentials, the available types of bond order factors are always inquired from the Fortran core to ensure that any changes made to the core are automatically reflected in the Python interface.

The same utility functions in `pysic` for inquiring keywords and other data needed for creating the potentials also work for fetching information on bond order factors, if applicable. The functions check automatically if the inquired name matches a potential or a bond order factor and gather the correct type of information based on this.

For example:

- Inquire the names of available bond order factors: `list_valid_bond_order_factors()`
- Inquire the names of parameters for a bond order factors: `names_of_parameters()`

Bond order cutoffs

Atomic coordination is an example of a simple bond order factor. It is calculated by checking all atom pairs and counting which ones are “close” to each others. Close naturally means closer than some predefined cutoff distance. However, in order to make the coordination a continuous and differentiable function, a continuous cutoff has to be applied. This is done similarly to the smooth cutoffs used in `Potential` by defining a proximity function which is 1 for small separations and 0 for large distances.

$$f(r) = \begin{cases} 1, & r < r_{\text{soft}} \\ \frac{1}{2} \left(1 + \cos \pi \frac{r - r_{\text{soft}}}{r_{\text{hard}} - r_{\text{soft}}} \right), & r_{\text{soft}} < r < r_{\text{hard}} \\ 0, & r > r_{\text{hard}} \end{cases}$$

Since bond order factors such as atomic coordination need not decay as a function of distance, one must always define a margin for continuous cutoff in bond order factors.

Defining parameters

A `BondOrderParameters` instance defines the type of the bond order factor, the cutoffs, and parameters for one set of elements. The parameters are formally split according to the number of atoms they act on. So, an n-body factor can have parameters which are applied for 1-body, 2-body, etc. terms. Bond order factors are applied and

parameterized by atomic element types (chemical symbols). An n-body factor must always have one or several sets of n symbols to designate the atoms it affects. So, 2- and 3-body factors could accept for instance the following lists of symbols, respectively:

```
>>> two_body_targets = [['H', 'H'], ['H', 'O'], ['O', 'O']]
>>> three_body_targets = [['Si', 'O', 'H']]
```

As a rule of thumb, if an n-body bond order factor incorporates parameters for m bodies ($m \leq n$), then these parameters are targeted at the first m symbols of the target list. For instance, if a 3-body factor has the targets of the above example (three_body_targets) and it contains 1- and 2-body parameters, then the 1-body parameters are targeted at Si atoms and the 2-body parameters at Si-O bonds.

As an example, the Tersoff bond order factor

$$b_i = \left[1 + \left(\beta_i \sum_{j \neq i} \sum_{k \neq i, j} \xi_{ijk} g_{ijk} \right)^{\eta_i} \right]^{-\frac{1}{2\eta_i}}$$

$$\xi_{ijk} = f(r_{ij})f(r_{ik}) \exp [a_{ij}^{\mu_i}(r_{ij} - r_{ik})^{\mu_i}]$$

$$g_{ijk} = 1 + \frac{c_{ij}^2}{d_{ij}^2} - \frac{c_{ij}^2}{d_{ij}^2 + (h_{ij} - \cos \theta_{ijk})^2}$$

is a three-body factor (it includes terms depending on atom triplets i, j, k) and therefore requires a set of three elements as its target. It incorporates three single body and four two body parameters. Such a bond order factor could be created with the following command:

```
>>> bonds = pysic.BondOrderParameters('tersoff', cutoff=3.2, cutoff_margin=0.4,
...                                   symbols=[['Si', 'Si', 'Si']],
...                                   parameters=[beta, eta, mu],
...                                   [a, c, d, h],
...                                   [ ])
```

or alternatively in pieces by a series of commands:

```
>>> bonds = pysic.BondOrderParameters('tersoff')
>>> bonds.set_cutoff(3.2)
>>> bonds.set_cutoff_margin(0.4)
>>> bonds.set_symbols(['Si', 'Si', 'Si'])
>>> bonds.set_parameter_value('beta', beta)
>>> bonds.set_parameter_value('eta', eta)
>>> bonds.set_parameter_value('mu', mu)
>>> bonds.set_parameter_value('a', a)
>>> bonds.set_parameter_value('c', c)
>>> bonds.set_parameter_value('d', d)
>>> bonds.set_parameter_value('h', h)
```

To be used in calculations, this is then passed on to a `Coordinator`, `Potential`, and `Pysic` with:

```
>>> crd = pysic.Coordinator( bonds )
>>> pot = pysic.Potential( ... , coordinator=crd )
>>> cal = pysic.Pysic( potentials=pot )
```

The above example creates a bond order factor which is applied to all Si triplets (symbols=[['Si','Si','Si']]). The command also assigns 1-body parameters beta, eta, and mu, and 2-body parameters a, c, d, and h. If there are other elements in the system besides silicon, they will be completely ignored: The bond order factors are calculated as if the other elements do not exist. If one wishes to include, say, Si-O bonds in the bond order factor calculation, the list of symbols needs to be expanded by:

```
>>> bonds.add_symbols(['Si', 'Si', 'O'],
...                   ['Si', 'O', 'Si'],
...                   ['Si', 'O', 'O'])
>>> bonds.get_symbols()
[['Si', 'Si', 'Si'], ['Si', 'Si', 'O'], ['Si', 'O', 'Si'], ['Si', 'O', 'O']]
```

The format of the symbol list is as follows. In each triplet, the first symbol determines the element on which the factor is calculated. Since above the first symbol of each triplet is 'Si', the factor will only be applied on Si atoms. The other symbols define the other elements in the triplets which are taken in to account. The second and third symbols are not, however, symmetric. As the bond order factor is defined using 2-body parameters (a_{ij} etc.), the first two symbols determine the elements of those two atoms (atoms i and j). The third symbol determines the element of the third atom (atom k) of the triplet. I.e., in the example above, Si-O bond parameters are included with:

```
>>> [['Si', 'O', 'Si'], ['Si', 'O', 'O']]
```

where the first works for triplets O-Si-Si and the second for O-Si-O. However, one should note especially that a triplet A-B-C is only taken in to account if both bonds (A-B) and (B-C) have parameters associated with them. Therefore, ['Si', 'O', 'O'] is enough to fully define O-Si-O bond triplets, but to fully describe Si-Si-O, one also has to define the Si-Si bond with O as the third partner, which is given by:

```
>>> [['Si', 'Si', 'O']]
```

Instead of giving a list of symbols to a single `BondOrderParameters`, one can define many instances with different symbols and different parameters, and feed a list of these to a `Coordinator` object.:

```
>>> bond_siosi = pysic.BondOrderParameters('tersoff', cutoff=3.2, cutoff_margin=0.4,
...                                       symbols=['Si', 'O', 'Si'],
...                                       parameters=[beta_si, eta_si, mu_si,
...                                                  [a_sio, c_sio, d_sio, h_sio],
...                                                  [ ]])
>>> bond_sisi = pysic.BondOrderParameters('tersoff', cutoff=3.5, cutoff_margin=0.5,
...                                       symbols=['Si', 'Si', 'O'],
...                                       parameters=[beta_si, eta_si, mu_si,
...                                                  [a_sisi, c_sisi, d_sisi, h_sisi],
...                                                  [ ]])
>>> bond_list = [bond_siosi, bond_sisi]
>>> crd = pysic.Coordinator( bond_list )
```

The above example would assign the parameter values

$$\begin{aligned}\beta_{\text{Si}} &= \text{beta_si} \\ \eta_{\text{Si}} &= \text{eta_si} \\ \mu_{\text{Si}} &= \text{mu_si} \\ a_{\text{Si-O}} &= \text{a_sio} \\ c_{\text{Si-O}} &= \text{c_sio} \\ d_{\text{Si-O}} &= \text{d_sio} \\ h_{\text{Si-O}} &= \text{h_sio} \\ a_{\text{Si-Si}} &= \text{a_sisi} \\ c_{\text{Si-Si}} &= \text{c_sisi} \\ d_{\text{Si-Si}} &= \text{d_sisi} \\ h_{\text{Si-Si}} &= \text{h_sisi}\end{aligned}$$

This gives the user the possibility to precisely control the parameters, including cutoffs, for different elements.

Note that the beta, eta, and mu parameters are the same for both `BondOrderParameters` objects defined in the above example. They could be different in principle, but when the factors are calculated, the 1-body parameters are

taken from the first object in the list of bonds (`bond_list`) for which the first element is of the correct type. Because of this, the 1-body parameters in `bond_sisio` are in fact ignored. This feature can be exploited for mixing different types of bond order factors, as explained below.

For three different elements, say C, O, and H, the possible triplets are:

```
>>> [ ['H', 'H', 'H'], # H-H bond in an H-H-H triplet
...   ['H', 'H', 'C'], # H-H bond in an H-H-C triplet
...   ['H', 'H', 'O'], # H-H bond in an H-H-O triplet
...   ['H', 'O', 'H'], # H-O bond in an H-H-O triplet
...   ['H', 'O', 'C'], # H-O bond in an O-H-C triplet
...   ['H', 'O', 'O'], # H-O bond in an O-H-O triplet
...   ['H', 'C', 'H'], # etc.
...   ['H', 'C', 'C'],
...   ['H', 'C', 'O'],
...   ['O', 'H', 'H'],
...   ['O', 'H', 'C'],
...   ['O', 'H', 'O'],
...   ['O', 'O', 'H'],
...   ['O', 'O', 'C'],
...   ['O', 'O', 'O'],
...   ['O', 'C', 'H'],
...   ['O', 'C', 'C'],
...   ['O', 'C', 'O'],
...   ['C', 'H', 'H'],
...   ['C', 'H', 'C'],
...   ['C', 'H', 'O'],
...   ['C', 'O', 'H'],
...   ['C', 'O', 'C'],
...   ['C', 'O', 'O'],
...   ['C', 'C', 'H'],
...   ['C', 'C', 'C'],
...   ['C', 'C', 'O'] ]
```

In principle, one can attach a different set of parameters to each of these. Often though the parameters are mostly the same, and writing these kinds of lists for all possible combinations is cumbersome. To help in generating the tables, the utility method `expand_symbols_table()` can be used. For instance, the full list of triplets above can be created with:

```
>>> pysic.utility.convenience.expand_symbols_table(['C', 'O', 'H'],
...                                                  ['C', 'O', 'H'],
...                                                  ['C', 'O', 'H']])
```

List of currently available bond order factors

Below is a list of bond order factors currently implemented.

- *Coordination scaling function*
- *Square root scaling function*
- *Coordination bond order factor*
- *Power decay bond order factor*
- *Tersoff bond order factor*

Coordination scaling function 1-body bond order factor defined as

$$b_i(\Sigma_i) = \varepsilon_i \frac{\Delta \Sigma_i}{1 + \exp(\gamma_i \Delta \Sigma_i)}$$

$$\Delta \Sigma_i = C_i(\Sigma_i - N_i).$$

where Σ_i is the bond order sum.

In other words, this factor only overrides the scaling function of another bond order factor when mixed. Especially, it is zero if not paired with other bond order factors.

Keywords:

```
>>> names_of_parameters('c_scale')
[['epsilon', 'N', 'C', 'gamma']]
```

Square root scaling function 1-body bond order factor defined as

$$b_i(\Sigma_i) = \varepsilon_i \sqrt{\Sigma_i}$$

where Σ_i is the bond order sum and ε is a scaling constant.

Keywords:

```
>>> names_of_parameters('sqrt_scale')
[['epsilon']]
```

Coordination bond order factor 2-body bond order factor defined as

$$b_i = \sum_{j \neq i} f(r_{ij}).$$

The coordination of an atom is simply the sum of the proximity functions. This is a parameterless (besides cutoffs) 2-body bond order factor.

Keywords:

```
>>> names_of_parameters('neighbors')
[[], []]
```

Power decay bond order factor 2-body bond order factor defined as

$$b_i = \sum_{j \neq i} f(r_{ij}) \varepsilon_{ij} \left(\frac{a_{ij}}{r_{ij}} \right)^{n_{ij}}.$$

This is a density-like bond factor, where the contributions of atomic pairs decay with interatomic distance according to a power law. In form, it is similar to the *Power decay potential* potential.

Keywords:

```
>>> names_of_parameters('power_bond')
[[], [epsilon, a, n]]
```

Tersoff bond order factor 3-body bond order factor defined as

$$b_i = \left[1 + \left(\beta_i \sum_{j \neq i} \sum_{k \neq i, j} \xi_{ijk} g_{ijk} \right)^{\eta_i} \right]^{-\frac{1}{2\eta_i}}$$

$$\xi_{ijk} = f(r_{ij})f(r_{ik}) \exp [a_{ij}^{\mu_i}(r_{ij} - r_{ik})^{\mu_i}]$$

$$g_{ijk} = 1 + \frac{c_{ij}^2}{d_{ij}^2} - \frac{c_{ij}^2}{d_{ij}^2 + (h_{ij} - \cos \theta_{ijk})^2}$$

where r and theta are distances and angles between the atoms. This rather complicated bond factor takes also into account the directionality of bonds in its angle dependency.

Keywords:

```
>>> names_of_parameters('tersoff')
[['beta', 'eta', 'mu'], ['a', 'c', 'd', 'h'], []]
```

List of methods

Below is a list of methods in `BondOrderParameters`, grouped according to the type of functionality.

Parameter handling

- `accepts_parameters()`
- `get_bond_order_type()`
- `get_cutoff()`
- `get_cutoff_margin()`
- `get_number_of_parameters()`
- `get_parameter_names()`
- `get_parameter_value()`
- `get_parameter_values()`
- `get_parameters_as_list()`
- `get_soft_cutoff()`
- `set_cutoff()`
- `set_cutoff_margin()`
- `set_parameter_value()`
- `set_parameter_values()`
- `set_parameters()`
- `set_soft_cutoff()`

Target handling

- `accepts_target_list()`
- `add_symbols()`
- `get_different_symbols()`
- `get_number_of_targets()`
- `get_symbols()`
- `set_symbols()`

Full documentation of the `BondOrderParameters` class

class `pysic.BondOrderParameters` (*bond_order_type*, *cutoff=0.0*, *cutoff_margin=0.0*, *parameters=None*, *symbols=None*)

Class for representing a collection of parameters for bond order calculations.

Calculating bond order factors using Tersoff-like methods defined in `Coordinator` requires several parameters per element and element pair. To facilitate the handling of all these parameters, they are wrapped in a `BondOrderParameters` object.

The object can be created empty and filled later with the parameters. Alternatively, a list of parameters can be given upon initialization in which case it is passed to the `set_parameters()` method.

Parameters:

bond_order_type: string a keyword specifying the type of the bond order factor

soft_cut: double The soft cutoff for calculating partial coordination. Any atom closer than this is considered a full neighbor.

hard_cut: double The hard cutoff for calculating partial coordination. Any atom closer than this is considered (at least) a partial neighbor and will give a fractional contribution to the total coordination. Any atom farther than this will not contribute to the neighbor count.

parameters: list of doubles a list of parameters to be contained in the parameter object

symbols: list of strings a list of elements on which the factor is applied

accepts_parameters (*params*)

Test if the given parameters array has the correct dimensions.

A bond order parameter can contain separate parameters for single, pair etc. elements and each class can have a different number of parameters. This method checks if the given list has the correct dimensions.

Parameters:

params: list of doubles list of parameters

accepts_target_list (*targets*)

Tests whether a list is suitable as a list of targets, i.e., element symbols and returns True or False accordingly.

A list of targets should be of the format:

```
targets = [[a, b], [c, d]]
```

where the length of the sublists must equal the number of targets.

It is not tested that the values contained in the list are valid.

Parameters:

targets: list of strings or integers a list whose format is checked

add_symbols (*symbols*)
 Adds the given symbols to the list of symbols.

Parameters:

symbols: list of strings list of additional symbols on which the bond order factor acts

get_bond_order_type ()
 Returns the keyword specifying the type of the bond order factor.

get_cutoff ()
 Returns the cutoff.

get_cutoff_margin ()
 Returns the margin for a smooth cutoff.

get_different_symbols ()
 Returns a list containing each symbol the potential affects once.

get_number_of_parameters ()
 Returns the number of parameters the bond order parameter object contains.

get_number_of_targets ()
 Returns the (maximum) number of targets the bond order factor affects.

get_parameter_names ()
 Returns a list of the names of the parameters of the potential.

get_parameter_value (*param_name*)
 Returns the value of the given parameter.

Parameters:

param_name: string name of the parameter

get_parameter_values ()
 Returns a list containing the current parameter values of the potential.

get_parameters_as_list ()
 Returns the parameters of the bond order factor as a single list.

The generated list first contains the single element parameters, then pair parameters, etc.

get_soft_cutoff ()
 Returns the lower limit for a smooth cutoff.

get_symbols ()
 Returns the symbols the bond parameters affect.

set_cutoff (*cutoff*)
 Sets the cutoff to a given value.

This method affects the hard cutoff.

Parameters:

cutoff: double new cutoff for the bond order factor

set_cutoff_margin (*margin*)
 Sets the margin for smooth cutoff to a given value.

This method defines the decay interval $r_{\text{hard}} - r_{\text{soft}}$. Note that if the soft cutoff value is made smaller than 0 or larger than the hard cutoff value an `InvalidParametersError` is raised.

Parameters:

margin: double The new cutoff margin

set_parameter_value (*parameter_name*, *value*)
Sets a given parameter to the desired value.

Parameters:

parameter_name: string name of the parameter

value: double the new value of the parameter

set_parameter_values (*values*)
Sets the numeric values of all parameters.

Parameters:

params: list of doubles list of values to be assigned to parameters

set_parameters (*params*)
Sets the numeric values of all parameters.

Equivalent to `set_parameter_values()`.

Parameters:

params: list of doubles list of values to be assigned to parameters

set_soft_cutoff (*cutoff*)
Sets the soft cutoff to a given value.

Note that actually the cutoff margin is recorded, so changing the hard cutoff (see `set_cutoff()`) will also affect the soft cutoff.

Parameters:

cutoff: double The new soft cutoff

set_symbols (*symbols*)
Sets the list of symbols to equal the given list.

Parameters:

symbols: list of strings list of element symbols on which the bond order factor acts

ChargeRelaxation class

This class controls equilibration of atomic charges in the system.

It is possible for the user to define the charges of atoms in ASE. If a system exhibits charge transfer, polarization, charged defects etc., one may not know the charges beforehand or the charges may change dynamically during simulation. To handle such systems, it is possible to let the charges in the system develop dynamically.

Since charge dynamics are usually much faster than dynamics of the ions, it is usually reasonable to allow the charges to equilibrate between ionic steps. This does not conserve energy exactly, however, since the charge equilibration drives the system charge distribution towards a lower energy. The energy change in charge redistribution is lost unless it is fed back to the system.

Connecting the structure, calculator and relaxation algorithm

Special care must be taken when setting up links between the atomic structure (`ASE Atoms`), the calculator (`Pysic`), and the charge relaxation algorithm (`ChargeRelaxation`). While some of the objects must know the others, in some cases the behavior of the simulator changes depending on whether or not they have access to the other objects.

The atoms and the calculator are linked as required in the `ASE calculator interface`: One can link the two by either the `set_atoms()` method of `Pysic`, or the `set_calculator` method of `ASE Atoms`. In either case, the atomic structure is given a link to the calculator, and a **copy** of the structure is stored in the calculator. This must be done in order to do any calculations on the system.

Also the relaxation algorithm has to know the `Pysic` calculator, since the relaxation is done according to the `Potential` interactions stored in the calculator. The algorithm can be made to know the calculator via the `set_calculator()` method of `ChargeRelaxation`. By default, this does not make the calculator know the relaxation algorithm, however. Only if the optional argument `reciprocal=True` is given the backwards link is also made. `Pysic` can be made to know the relaxation algorithm also by calling the method `set_charge_relaxation()`. Unlike the opposite case, by making the link from the calculator, the backwards link from the relaxation algorithm is always made automatically. In fact, even though linking an algorithm to a calculator does not automatically link the calculator to the algorithm, if a different calculator was linked to the algorithm, the link is automatically removed.

This slightly complicated behavior is summarized as follows: The algorithm should always have a link to a calculator, but a calculator need not have a link to an algorithm. If a calculator does link to an algorithm, the algorithm must link back to the same calculator. Clearly one does not always want to perform charge relaxation on the system and so it makes sense that the calculator need not have a link to a charge relaxation algorithm. If such a link does exist, then the relaxation is *automatically* invoked prior to each energy and force evaluation. This is necessary in simulations such as molecular dynamics (MD). A charge relaxation can be linked to a calculator in order to do charge equilibration, but if one does not wish to trigger the charge relaxation automatically, then it is enough to just not let the calculator know the relaxation algorithm.

The atomic structure cannot be given a link to the relaxation algorithm since the charge relaxation is not part of the ASE API and so the atoms object does not know how to interact with it. In essence, from the point of view of the structure, the charge relaxation is fully contained in the calculator.

The charge relaxation algorithm always acts on the structure contained in the calculator. The atomic charges of this structure are automatically updated during the relaxation. Since the calculator only stores a copy of the original structure, the original is not updated. This may be desired if, for instance, one wishes to revert back to the original charges. However, during structural dynamics simulations such as MD, it is necessary that the relaxed charges are saved between structural steps. This is a problem, since structural dynamics are handled by ASE, and ASE invokes the calculation of forces with the original `ASE Atoms` object. Therefore, if the relaxed charges are not saved, charge relaxation is always started from the original charges, which may be very inefficient. In order to have also the original structure updated automatically, the charge relaxation can be made to know the original structure with `set_atoms()`. Note that the structure given to the algorithm is not used in the actual relaxation; the algorithm always works on the structure in the calculator, which may be different. The given structure is merely updated according to the calculation results.

List of currently available relaxation methods

Below is a list of the charge relaxation methods currently implemented.

Damped dynamics Assigning an inertia, M_q , on the atomic charges, q_i , we can describe the system with the Lagrangian

$$L = \sum_i \frac{1}{2} m_i \dot{\mathbf{r}}_i^2 + \sum_i \frac{1}{2} M_q \dot{q}_i^2 - U(\{q\}, \{\mathbf{r}\}) - \nu \sum_i q_i,$$

where m_i, \mathbf{r}_i are the mass and position of atom i , respectively. The last term is a Lagrange multiplier corresponding to the constraint of fixed total charge, i.e., $\sum_i q_i = Q_{\text{tot}}$ being constant. The total potential energy U is a function of all charges and positions.

The equations of motion for this system are

$$m_i \ddot{\mathbf{r}}_i = -\nabla_i U \quad (4.12)$$

$$M_q \ddot{q}_i = -\frac{\partial U}{\partial q_i} - \nu. \quad (4.13)$$

In the charge equation, the Lagrange multiplier can be shown to equal the average electronegativity of the system, $\nu = \bar{\chi}$, and the derivative is the effective electronegativity of atom i , $-\frac{\partial U}{\partial q_i} = \chi_i$. Thus, the effective force driving the change in atomic charges is the electronegativity difference from the average

$$M_q \ddot{q}_i = -\frac{\partial U}{\partial q_i} - \nu = \chi_i - \bar{\chi} = \Delta\chi_i.$$

In the damped dynamic equilibration, the charges are developed dynamically according to the equation of motion with an added damping (friction) term $-\eta\dot{q}_i$

$$M_q \ddot{q}_i = \Delta\chi_i - \eta\dot{q}_i. \quad (4.14)$$

This leads to the charges being driven towards a state where the driving force vanishes $\Delta\chi_i = 0$, i.e., the electronegativities are equal.

During simulation such as molecular dynamics or geometry optimization, charge equilibration is done by running the damped charge dynamics (4.14) before each force or energy evaluation.

List of methods

Below is a list of methods in `ChargeRelaxation`, grouped according to the type of functionality.

Initialization

- `initialize_parameters()`
- `get_relaxation()`
- `set_relaxation()`
- `relaxation_modes`
- `relaxation_parameter_descriptions`
- `relaxation_parameters`

Atoms handling

- `get_atoms()`
- `set_atoms()`

Calculator handling

- `get_calculator()`
- `set_calculator()`

Parameter handling

- `get_parameters()`
- `set_parameter_value()`
- `set_parameter_values()`
- `set_parameters()`

Charge relaxation

- `charge_relaxation()`

Full documentation of the ChargeRelaxation class

```
class pysic.ChargeRelaxation (relaxation='dynamic',      calculator=None,      parameters=None,
                             atoms=None)
```

A class for handling charge dynamics and relaxation.

Pysic does not implement molecular dynamics or geometric optimization since they are handled by ASE. Conceptually, the structural dynamics of the system are properties of the atomic geometry and so it makes sense that they are handled by ASE, which defines the atomic structure in the first place, in the [ASE Atoms](#) class.

On the other hand, charge dynamics are related to the electronic structure of the system. Since ASE is meant to use methods such as density functional theory (DFT) in the calculators it employs, all electronic properties are left at the responsibility of the calculator. This makes sense since in DFT the electron density is needed for calculations of forces and energies.

Pysic is not a DFT calculator and there is no electron density but the atomic charges can be made to develop dynamically. The ChargeRelaxation class handles these dynamics.

Parameters:

relaxation: **string** a keyword specifying the mode of charge relaxation

calculator: **Pysic object** a Pysic calculator

parameters: **list of doubles** numeric values for parameters

atoms: **ASE Atoms object** The system whose charges are to be relaxed. Note! The relaxation is always done using the atoms copy in Pysic, but if the original structure needs to be updated as well, the relaxation algorithm must have access to it.

charge_relaxation()

Performs the charge relaxation.

The relaxation is always performed on the system associated with the Pysic calculator joint with this ChargeRelaxation. The calculated equilibrium charges are returned as a numeric array.

If an ASE Atoms structure is known by the ChargeRelaxation (given through `set_atoms()`), the charges of the structure are updated according to the calculation result. If the structure is not known, the charges are updated in the structure stored in the Pysic calculator, but not in any other object. Since Pysic only stores a copy of the structure it is given, the original ASE Atoms object will not be updated.

get_atoms()

Returns the atoms object known by the algorithm.

This is the ASE Atoms which will be automatically updated when charge relaxation is invoked.

get_calculator()

Returns the Pysic calculator assigned to this ChargeRelaxation.

get_parameters()

Returns a list containing the numeric values of the parameters.

get_relaxation()

Returns the keyword specifying the mode of relaxation.

initialize_parameters()

Creates a dictionary of parameters and initializes all values to 0.0.

relaxation_modes = ['dynamic']

Names of the charge relaxation algorithms available.

These are keywords needed when creating the `ChargeRelaxation` objects as type specifiers.

relaxation_parameter_descriptions = {'dynamic': ['time steps of charge dynamics between molecular dynamic']}

Short descriptions of the relaxation parameters.

relaxation_parameters = {'dynamic': ['n_steps', 'timestep', 'inertia', 'friction', 'tolerance']}

Names of the parameters of the charge relaxation algorithms.

set_atoms(*atoms*, *pass_to_calculator=False*)

Lets the relaxation algorithm know the atomic structure to be updated.

The relaxation algorithm always works with the structure stored in the `Pysic` calculator it knows. If `pass_to_calculator = True`, this method also updates the structure known by the calculator. However, this is not the main purpose of letting the `ChargeRelaxation` know the structure - it is not even necessary that the structure known by the relaxation algorithm is the same as that known by the calculator.

The structure given to the algorithm is the structure whose charges it automatically updates after relaxing the charges in `charge_relaxation()`. In other words, if no structure is given, the relaxation will update the charges in the structure known by `Pysic`, but this is always just a copy and so the original structure is left untouched.

Parameters:

atoms: ASE Atoms object The system whose charges are to be relaxed. Note! The relaxation is always done using the atoms copy in `Pysic`, but if the original structure needs to be updated as well, the relaxation algorithm must have access to it.

pass_to_calculator: logical if True, the atoms are also set for the calculator via `set_atoms()`

set_calculator(*calculator*, *reciprocal=False*)

Assigns a `Pysic` calculator.

The calculator is necessary for calculation of electronegativities. It is also possible to automatically assign the charge relaxation method to the calculator by setting `reciprocal = True`.

Note though that it does make a difference whether the calculator knows the charge relaxation or not: If the `Pysic` has a connection to the `ChargeRelaxation`, every time force or energy calculations are requested the charges are first relaxed by automatically invoking `charge_relaxation()`. If there is no link, it is up to the user to start the relaxation.

Parameters:

calculator: Pysic object a `Pysic` calculator

reciprocal: logical if True, also the `ChargeRelaxation` is passed to the `Pysic` through `set_charge_relaxation()`.

set_parameter_value(*parameter_name*, *value*)

Sets a given parameter to the desired value.

Parameters:

parameter_name: **string** name of the parameter

value: **double** the new value of the parameter

set_parameter_values (*parameters*)
Sets the numeric values for all parameters.

Parameters:

parameters: **list of doubles** list of values to be assigned to parameters

set_parameters (*parameters*)
Sets the numeric values for all parameters.

Equivalent to `set_parameter_values()`

Parameters:

parameters: **list of doubles** list of values to be assigned to parameters

set_relaxation (*relaxation*)
Sets the relaxation method.

The method also creates a dictionary of parameters initialized to 0.0 by invoking `initialize_parameters()`.

Parameters:

relaxation: **string** a keyword specifying the mode of charge relaxation

FastNeighborList class

This class extends the [ASE NeighborList](#) class to provide a more efficient neighbor finding tool. The neighbor finding routine searches the neighborhoods of all atoms and for each atom records which other atoms are closer than a given cutoff distance.

The benefit of neighbor lists

Atomistic pair and many-body potentials typically depend on the local atomic structure and especially the relative coordinates of the atoms. However, finding the separation vector and distance between coordinates in periodic 3D space is computationally fairly costly operation and the number of atom-atom pairs in the system grows as $\mathcal{O}(n^2)$. Therefore the evaluation of local potentials can be made efficient by storing lists of nearby atoms for all particles to narrow down the scope of search for interacting neighbors.

Typically one chooses a cutoff distance r_{cut} beyond which the atoms do not see each other. Then, the neighbor lists should always contain all the atoms within this cutoff radius $r_{ij} \leq r_{\text{cut}}$. In dynamic simulations where the atoms move, the typical scheme is to list atoms within a slightly longer radius, $r_{\text{cut}} + r_{\text{skin}}$ because then the lists need not be updated until an atom has moved by more than r_{skin} .

Faster neighbor search

There is a built in neighbor searching tool in ASE, [ASE NeighborList](#). It is, however, a pure Python implementation using a brute-force $\mathcal{O}(n^2)$ algorithm making it slow - even prohibitively slow - for large systems especially when periodic boundary conditions are used.

To overcome this performance bottleneck, Pysic implements the [FastNeighborList](#) class. This class inherits other properties from the built-in ASE class except for the `build()` method, which is replaced by a faster algorithm. The fast neighbor search is implemented in Fortran and parallelized with MPI. The algorithm is based on a spatial divisioning, i.e.

- the simulation volume is divided in subvolumes
- for each atom the subvolume where it is contained is found
- for each atom, the neighbors are searched for only in the adjacent subvolumes

For a fixed cutoff, the neighborhood searched for each atom is constant and thus this is an $\mathcal{O}(n)$ algorithm.² The method is also faster the shorter the cutoffs are. For short cutoffs (~ 5 Å), a 10000 atom periodic system is expected to be handled 100 or even 1000 fold faster with `FastNeighborList` than with the ASE method.

Limitations in the implementation

Since the fast algorithm is implemented in Fortran, it operates on the structure allocated in the Fortran core. Therefore, even though the `build()` method takes an `ASE Atoms` object as an argument, it does not analyze the given structure. It does check against `CoreMirror` to see if the given structure matches the one in the core and raises an error if not, but accessing the core has to still be done through `Pysic`. When `Pysic` is run normally, this is automatically taken care of. As the implementation is MPI parallelized, it is also necessary that the MPI environment has been set up - especially the distribution of load (i.e. atoms) between processors must be done before the lists can be built.

Another more profound limitation in the current implementation of the algorithm is the fact that it limits the neighbor finding to neighboring subvolumes. Since the subvolumes are not allowed to be larger than the actual simulation volume, the cutoffs cannot be longer than the shortest perpendicular separation between facets of the subvolume. For rectangular cells, this is just the minimum of the lengths of the vectors spanning the cell, $\mathbf{v}_{i,j,k}$. For inclined cell shapes, the perpendicular distance between cell facets, d , is

$$d_i = \frac{|\mathbf{v}_i \cdot \mathbf{n}_i|}{|\mathbf{n}_i|} \quad (4.15)$$

$$\mathbf{n}_i = \mathbf{v}_j \times \mathbf{v}_k \quad (4.16)$$

where \mathbf{n}_i are the normal vectors of the plane spanned by the vectors $\mathbf{v}_{j,k}$. If one wishes to find neighbors in a radius containing the simulation volume several times, the original `ASE NeighborList` should be used instead. `Pysic` does this choice automatically when building the neighbor lists. One should usually avoid such long cutoffs in the first place, but if your system is very small that may not be possible.

Methods inherited from ASE NeighborList

- `get_neighbors`
- `update`

List of methods

- `build()`

Full documentation of the FastNeighborList class

`class pysic.FastNeighborList (cutoffs, skin=0.5)`

ASE has a neighbor list class built in, but its implementation is currently inefficient, and building of the list is an $\mathcal{O}(n^2)$ operation. This neighbor list class overrides the `build()` method with an $\mathcal{O}(n)$ time routine. The fast routine is based on a spatial partitioning algorithm.

² For very large systems the number of subdivisions is limited to conserve memory so the $\mathcal{O}(n)$ scaling is eventually lost. Say we divide the volume in a hundred subvolumes along each axis; we end up with a million subvolumes which is a lot!

The way cutoffs are handled is also somewhat different to the original ASE list. In ASE, the distances for two atoms are compared against the sum of the individual cutoffs + neighbor list skin. This list, however, searches for the neighbors of each atom at a distance of the cutoff of the given atom only, plus skin.

build(*atoms*)

Builds the neighbor list.

The routine requires that the given atomic structure matches the one in the core. This is because the method invokes the Fortran core to do the neighbor search. The method overrides the similar method in the original ASE neighborlist class, which directly operates on the given structure, so this method also takes the atomic structure as an argument. However, in order to keep the core modification routines in the `Pysic` class, this method does not change the core structure. It does raise an error if the structures do not match, though.

The neighbor search is done via the `generate_neighbor_lists()` routine. The routine builds the neighbor list in the core, after which the list is fed back to the `FastNeighborList` object by looping over all atoms and saving the lists of neighbors and offsets.

Parameters:

atoms: ASE Atoms object the structure for which the neighbors are searched

CoreMirror class

CoreMirror is a Python representation of the Fortran core. When running `pysic`, it is intended that a single instance of CoreMirror exists, created automatically when importing `pysic` as the `core` object in `Pysic`.

Whenever changes are made in the Fortran core, they should also be reflected in the CoreMirror. This way one has always easy access to the state of the Fortran core without having to directly access the core and parse the data.

Normally, the user should not touch the CoreMirror directly. It is automatically handled through `Pysic`.

List of Methods

- `atoms_ready()` (meant for internal use)
- `cell_ready()` (meant for internal use)
- `charges_ready()` (meant for internal use)
- `coulomb_summation_ready()` (meant for internal use)
- `get_atoms()` (meant for internal use)
- `neighbor_lists_ready()` (meant for internal use)
- `potentials_ready()` (meant for internal use)
- `set_atomic_momenta()` (meant for internal use)
- `set_atomic_positions()` (meant for internal use)
- `set_atoms()` (meant for internal use)
- `set_cell()` (meant for internal use)
- `set_charges()` (meant for internal use)
- `set_coulomb()` (meant for internal use)
- `set_neighbor_lists()` (meant for internal use)
- `set_potentials()` (meant for internal use)

- `view_fortran()` (for testing)

Full documentation of the **CoreMirror** class

class `pysic.CoreMirror`

A class representing the status of the core.

Whenever data is being passed over to the core for calculation, it should also be saved in the **CoreMirror**. This makes the **CoreMirror** reflect the current status of the core. Then, when something needs to be calculated, the **Pysic** calculator can simply check that it contains the same system as the **CoreMirror** to ensure that the core operates on the correct data.

All data given to **CoreMirror** is saved as deep copies, i.e., not as the objects themselves but objects with exactly the same contents. This way if the original objects are modified, the ones in **CoreMirror** are not. This is the proper way to work, since the Fortran core obviously does not change without pushing the changes in the Python side to the core first.

Since exactly one **CoreMirror** should exist during the simulation, deletion of the instance (which should happen at program termination automatically) will automatically trigger release of memory in the Fortran core as well as termination of the MPI framework.

atoms_ready (*atoms*)

Checks if the positions and momenta of the given atoms match those in the core.

True is returned if the structures match, False otherwise.

Parameters:

atoms: **ASE Atoms** object The atoms to be compared.

cell_ready (*atoms*)

Checks if the given supercell matches that in the core.

True is returned if the structures match, False otherwise.

Parameters:

atoms: **ASE Atoms** object The cell to be compared.

charges_ready (*atoms*)

Checks if the charges of the given atoms match those in the core.

True is returned if the charges match, False otherwise.

Parameters:

atoms: **ASE Atoms** object The atoms to be compared.

coulomb_summation_ready (*coulomb*)

Checks if the given Coulomb summation matches that in the core.

True is returned if the summation algorithms match, False otherwise.

Parameters: **CoulombSummation** the summation algorithm to be compared

get_atoms ()

Returns the **ASE Atoms** structure stored in the **CoreMirror**.

neighbor_lists_ready (*lists*)

Checks if the given neighbor lists match those in the core.

True is returned if the structures match, False otherwise.

Parameters:

atoms: **ASE NeighborList object** The neighbor lists to be compared.

potentials_ready (*pots*)
Checks if the given potentials match those in the core.
True is returned if the potentials match, False otherwise.
Parameters:
atoms: list of **Potential objects** The potentials to be compared.

set_atomic_momenta (*atoms*)
Copies and stores the momenta of atoms in the **ASE Atoms** instance.
Parameters:
atoms: **ASE Atoms object** atomic structure containing the momenta to be saved.

set_atomic_positions (*atoms*)
Copies and stores the positions of atoms in the **ASE Atoms** instance.
Parameters:
atoms: **ASE Atoms object** atomic structure containing the positions to be saved.

set_atoms (*atoms*)
Copies and stores the entire **ASE Atoms** instance.
Parameters:
atoms: **ASE Atoms object** atomic structure to be saved

set_cell (*atoms*)
Copies and stores the supercell in the **ASE Atoms** instance.
Parameters:
atoms: **ASE Atoms object** atomic structure containing the supercell to be saved.

set_charges (*charges*)
Copies and stores the charges of atoms in the **ASE Atoms** instance.
Parameters:
atoms: **ASE Atoms object** atomic structure containing the positions to be saved.

set_coulomb (*coulomb*)
Copies and stores the Coulomb summation algorithm.
Parameters:
coulomb: **CoulombSummation** Coulomb summation algorithm to be saved

set_neighbor_lists (*lists*)
Copies and stores the neighbor lists.
Parameters:
atoms: **ASE NeighborList object** Neighbor lists to be saved.

set_potentials (*potentials*)
Copies and stores **Potential** potentials.
The **Potential** instances are copied as a whole, so any possible **Coordinator** and **BondOrderParameters** objects are also stored.
Parameters:

atoms: list of **Potential** objects Potentials to be saved.

view_fortran()

Print some information on the data allocated in the Fortran core.

This is mainly a debugging utility for directly seeing the status of the core. It can be accessed through:

```
>>> pysic.Pysic.core.view_fortran()
```

The result is a bunch of data dumped to stdout. The function does not return anything.

4.2.2 List of methods

Below is a list of methods in `pysic`.

Potential and bond order factor inquiry

- `list_potentials()`
- `list_valid_potentials()`
- `is_potential()`
- `is_valid_potential()`
- `list_bond_order_factors()`
- `list_valid_bond_order_factors()`
- `is_bond_order_factor()`
- `is_valid_bond_order_factor()`
- `number_of_targets()`
- `number_of_parameters()`
- `names_of_parameters()`
- `index_of_parameter()`
- `descriptions_of_parameters()`
- `description_of_potential()`

Charge relaxation inquiry

- `is_valid_charge_relaxation()`
- `is_charge_relaxation()`

Message Parsing Interface

- `finish_mpi()`
- `get_number_of_cpus()`
- `get_cpu_id()`

4.2.3 Errors defined by the pysic module

The module defines a group of intrinsic errors to describe situations where one tries to use or set up the calculator with erroneous or insufficient information.

exception `pysic.InvalidParametersError` (*message='', params=None*)

An error raised when an invalid set of parameters is about to be created.

Parameters:

message: `string` information describing why the error occurred

params: `BondOrderParameters` the erroneous parameters

exception `pysic.InvalidCoordinatorError` (*message='', coordinator=None*)

An error raised when an invalid coordination calculator is about to be created or used.

Parameters:

message: `string` information describing why the error occurred

coordinator: `Coordinator` the erroneous coordinator

exception `pysic.InvalidPotentialError` (*message='', potential=None*)

An error raised when an invalid potential is about to be created or used.

Parameters:

message: `string` information describing why the error occurred

potential: `Potential` the erroneous potential

exception `pysic.MissingAtomsError` (*message=''*)

An error raised when the core is being updated with per atom information before updating the atoms.

Parameters:

message: `string` information describing why the error occurred

exception `pysic.MissingNeighborsError` (*message=''*)

An error raised when a calculation is initiated without initializing the neighbor lists.

In principle `Pysic` should always take care of handling the neighbors automatically. This error is an indication that there is a loophole in the built-in preparations.

Parameters:

message: `string` information describing why the error occurred

exception `pysic.LockedCoreError` (*message=''*)

An error raised when a `Pysic` tries to access the core which is locked by another calculator.

Parameters:

message: `string` information describing why the error occurred

4.2.4 Functions of the pysic module

The module defines a group of functions to directly access the Fortran core for information on available potentials.

`pysic.list_potentials()`

Same as `list_valid_potentials()`

pysic.list_valid_potentials()

A list of names of potentials currently known by the core.

The method retrieves from the core a list of the names of different potentials currently implemented. Since the fortran core is directly accessed, any updates made in the core source code should get noticed automatically.

pysic.is_potential (*potential_name*)

Same as `is_valid_potential()`

Parameters:

potential_name: string the name of the potential

pysic.is_charge_relaxation (*relaxation_name*)

Same as `is_valid_charge_relaxation()`

Parameters:

relaxation_name: string the name of the relaxation mode

pysic.is_valid_charge_relaxation (*relaxation_name*)

Tells if the given string is the name of a charge relaxation mode.

Parameters:

relaxation_name: string the name of the relaxation mode

pysic.is_valid_potential (*potential_name*)

Tells if the given string is the name of a potential.

Parameters:

potential_name: string the name of the potential

pysic.list_bond_order_factors()

Same as `list_valid_bond_order_factors()`

pysic.list_valid_bond_order_factors()

A list of names of bond order factors currently known by the core.

The method retrieves from the core a list of the names of different bond factors currently implemented. Since the fortran core is directly accessed, any updates made in the core source code should get noticed automatically.

pysic.is_bond_order_factor (*bond_order_name*)

Same as `is_valid_bond_order_factor()`

Parameters:

bond_order_name: string the name of the bond order factor

pysic.is_valid_bond_order_factor (*bond_order_name*)

Tells if the given string is the name of a bond order factor.

Parameters:

bond_order_name: string the name of the bond order factor

pysic.number_of_targets (*potential_name*)

Tells how many targets a potential or bond order factor acts on, i.e., is it pair or many-body.

Parameters:

potential_name: string the name of the potential or bond order factor

pysic.number_of_parameters (*potential_name*, *as_list=False*)

Tells how many parameters a potential, bond order factor, charge relaxation mode or coulomb summation mode incorporates.

A potential has a simple list of parameters and thus the function returns by default a single number. A bond order factor can incorporate parameters for different number of targets (some for single elements, others for pairs, etc.), and so a list of numbers is returned, representing the number of single, pair etc. parameters. If the parameter 'as_list' is given and is True, the result is a list containing one number also for a potential.

Parameters:

potential_name: string the name of the potential or bond order factor

as_list: logical should the result always be a list

`pysic.names_of_parameters` (*potential_name*)

Lists the names of the parameters of a potential, bond order factor, charge relaxation mode or coulomb summation mode.

For a potential, a simple list of names is returned. For a bond order factor, the parameters are categorised according to the number of targets they apply to (single element, pair, etc.). So, for a bond order factor, a list of lists is returned, where the first list contains the single element parameters, the second list the pair parameters etc.

Parameters:

potential_name: string the name of the potential or bond order factor

`pysic.index_of_parameter` (*potential_name, parameter_name*)

Tells the index of a parameter of a potential or bond order factor in the list of parameters the potential uses.

For a potential, the index of the specified parameter is given. For a bond order factor, a list of two integers is given. These give the number of targets (single element, pair etc.) the parameter is associated with and the list index.

Note especially that an index is returned, and these start counting from 0. So for a bond order factor, a parameter for pairs (2 targets) will return 1 as the index for number of targets.

Parameters:

potential_name: string the name of the potential or bond order factor

parameter_name: string the name of the parameter

`pysic.descriptions_of_parameters` (*potential_name*)

Returns a list of strings containing physical names of the parameters of a potential, bond order factor, or charge relaxation mode, e.g., 'spring constant' or 'decay length'.

For a potential, a simple list of descriptions is returned. For a bond order factor, the parameters are categorised according to the number of targets they apply to (single element, pair, etc.). So, for a bond order factor, a list of lists is returned, where the first list contains the single element parameters, the second list the pair parameters etc.

Parameters:

potential_name: string the name of the potential or bond order factor

`pysic.description_of_potential` (*potential_name, parameter_values=None, cutoff=None, elements=None, tags=None, indices=None*)

Prints a brief description of a potential.

If optional arguments are provided, they are incorporated in the description. That is, by default the method describes the general features of a potential, but it can also be used for describing a particular potential with set parameters.

Parameters:

potential_name: string the name of the potential

`pysic.finish_mpi()`

Terminates the MPI framework.

If the Fortran core is compiled in MPI mode, `pysic` will automatically initialize MPI upon being imported. This method terminates the MPI.

`pysic.get_number_of_cpus()`

Gets the number of cpus from the Fortran MPI.

`pysic.get_cpu_id()`

Gets the cpu ID from the Fortran MPI.

4.3 Pysic Utility modules

Modules for providing utility tools and parameters.

4.3.1 Plot

`pysic.utility.plot.plot_abs_force_on_line(index, system, direction=None, length=None, steps=100, start=None, end=None, lims=[-10000000000.0, 10000000000.0])`

Plots the absolute value of the force on a particle as a function of the position.

The method probes the system by moving a single particle on a line and recording the force. A plot is drawn. Also a tuple containing arrays of the distance traveled and the recorded forces is returned.

After the operation is complete, the initial structure is restored.

Parameters:

index: integer index of the particle to be moved

system: ASE Atoms object the structure to be explored

direction: double 3-vector the direction where the atom is moved

length: double the distance moved

steps: integer number of points (taken uniformly on the movement path) for measuring the force

start: double 3-vector (array or list) starting point for the trajectory - if not specified, the position of the particle in 'system' is used

end: double 3-vector (array or list) end point for the trajectory - alternative for direction and length (will override them)

lims: double 2-vector (array or list) lower and upper truncation limits - if a recorded value is smaller than the lower limit or larger than the upper, it is replaced by the corresponding truncation value

`pysic.utility.plot.plot_abs_force_on_plane(index, system, directions, lengths, steps=100, start=None, lims=[-10000000000.0, 10000000000.0])`

Plots the absolute value of force on a particle as a function of the position.

The method probes the system by moving a single particle on a plane and recording the force. A contour plot is drawn. Also a tuple containing arrays of the distances traveled on the plane and the recorded forces is returned.

After the operation is complete, the initial structure is restored.

Parameters:

index: integer index of the particle to be moved

system: ASE Atoms object the structure to be explored

directions: double 2x3-matrix The directions where the atom is moved - i.e., the vectors defining the plane.
If the second vector is not perpendicular to the first, the normal component is automatically used instead.

lengths: double 2-vector the distances moved in the given directions

steps: integer number of points (taken uniformly on the movement plane) for measuring the force

start: double 3-vector (array or list) starting point for the trajectory, i.e., a corner for the plane to be probed - if not specified, the position of the particle in 'system' is used

lims: double 2-vector (array or list) lower and upper truncation limits - if a recorded value is smaller than the lower limit or larger than the upper, it is replaced by the corresponding truncation value

```
pysic.utility.plot.plot_energy_on_line(index, system, direction=None, length=None,
                                       steps=100, start=None, end=None, lims=[-
                                       10000000000.0, 10000000000.0])
```

Plots the energy of the system as a function of the position of a single particle.

The method probes the system by moving a single particle on a line and recording the energy. A plot is drawn. Also a tuple containing arrays of the distance traveled and the recorded energies is returned.

After the operation is complete, the initial structure is restored.

Parameters:

index: integer index of the particle to be moved

system: ASE Atoms object the structure to be explored

direction: double 3-vector the direction where the atom is moved

length: double the distance moved

steps: integer number of points (taken uniformly on the movement path) for measuring the energy

start: double 3-vector (array or list) starting point for the trajectory - if not specified, the position of the particle in 'system' is used

end: double 3-vector (array or list) end point for the trajectory - alternative for direction and length (will override them)

lims: double 2-vector (array or list) lower and upper truncation limits - if a recorded value is smaller than the lower limit or larger than the upper, it is replaced by the corresponding truncation value

```
pysic.utility.plot.plot_energy_on_plane(index, system, directions, lengths, steps=100,
                                       start=None, lims=[-10000000000.0,
                                       10000000000.0])
```

Plots the energy of the system as a function of the position of a particle.

The method probes the system by moving a single particle on a plane and recording the energy. A contour plot is drawn. Also a tuple containing arrays of the distances traveled on the plane and the recorded energies is returned.

After the operation is complete, the initial structure is restored.

Parameters:

index: integer index of the particle to be moved

system: ASE Atoms object the structure to be explored

directions: double 2x3-matrix The directions where the atom is moved - i.e., the vectors defining the plane.
If the second vector is not perpendicular to the first, the normal component is automatically used instead.

lengths: double 2-vector the distances moved in the given directions

steps: integer number of points (taken uniformly on the movement plane) for measuring the energy

start: double 3-vector (array or list) starting point for the trajectory, i.e., a corner for the plane to be probed - if not specified, the position of the particle in 'system' is used

lims: double 2-vector (array or list) lower and upper truncation limits - if a recorded value is smaller than the lower limit or larger than the upper, it is replaced by the corresponding truncation value

```
pysic.utility.plot.plot_force_component_on_plane(index,      system,      directions,
                                                  lengths,   component,  steps=100,
                                                  start=None, lims=[-10000000000.0,
                                                                10000000000.0])
```

Plots the projected component of force on a particle as a function of the position.

The method probes the system by moving a single particle on a plane and recording the force. The component of the force projected on a given vector is recorded. A contour plot is drawn. Also a tuple containing arrays of the distances traveled on the plane and the recorded forces is returned.

After the operation is complete, the initial structure is restored.

Parameters:

index: integer index of the particle to be moved

system: ASE Atoms object the structure to be explored

directions: double 2x3-matrix The directions where the atom is moved - i.e., the vectors defining the plane. If the second vector is not perpendicular to the first, the normal component is automatically used instead.

lengths: double 2-vector the distances moved in the given directions

component: double 3-vector the direction on which the force is projected - e.g., if component is [1,0,0], the x-component is recorded

steps: integer number of points (taken uniformly on the movement plane) for measuring the force

start: double 3-vector (array or list) starting point for the trajectory, i.e., a corner for the plane to be probed - if not specified, the position of the particle in 'system' is used

lims: double 2-vector (array or list) lower and upper truncation limits - if a recorded value is smaller than the lower limit or larger than the upper, it is replaced by the corresponding truncation value

```
pysic.utility.plot.plot_tangent_force_on_line(index,      system,      direction=None,
                                                length=None, steps=100, start=None,
                                                end=None,   lims=[-10000000000.0,
                                                                10000000000.0])
```

Plots the tangential force on a particle as a function of the position.

The method probes the system by moving a single particle on a line and recording the force tangent. A plot is drawn. Also a tuple containing arrays of the distance traveled and the recorded forces is returned.

After the operation is complete, the initial structure is restored.

Parameters:

index: integer index of the particle to be moved

system: ASE Atoms object the structure to be explored

direction: double 3-vector the direction where the atom is moved

length: double the distance moved

steps: integer number of points (taken uniformly on the movement path) for measuring the energy

start: double 3-vector (array or list) starting point for the trajectory - if not specified, the position of the particle in 'system' is used

end: double 3-vector (array or list) end point for the trajectory - alternative for direction and length (will override them)

lims: double 2-vector (array or list) lower and upper truncation limits - if a recorded value is smaller than the lower limit or larger than the upper, it is replaced by the corresponding truncation value

```
pysic.utility.plot.plot_tangent_force_on_plane(index, system, directions, lengths,
                                              steps=100, start=None, lims=[-
                                              10000000000.0, 10000000000.0])
```

Plots the absolute value of the tangent component of force on a particle as a function of the position.

The method probes the system by moving a single particle on a plane and recording the force. The force is projected on the same plane, and the absolute value of the projection is calculated. A contour plot is drawn. Also a tuple containing arrays of the distances traveled on the plane and the recorded forces is returned.

After the operation is complete, the initial structure is restored.

Parameters:

index: integer index of the particle to be moved

system: ASE Atoms object the structure to be explored

directions: double 2x3-matrix The directions where the atom is moved - i.e., the vectors defining the plane. If the second vector is not perpendicular to the first, the normal component is automatically used instead.

lengths: double 2-vector the distances moved in the given directions

steps: integer number of points (taken uniformly on the movement plane) for measuring the force

start: double 3-vector (array or list) starting point for the trajectory, i.e., a corner for the plane to be probed - if not specified, the position of the particle in 'system' is used

lims: double 2-vector (array or list) lower and upper truncation limits - if a recorded value is smaller than the lower limit or larger than the upper, it is replaced by the corresponding truncation value

4.3.2 Convenience

```
pysic.utility.convenience.expand_symbols_table(symbol_list, type=None)
```

Creates a table of symbols for a BondOrderParameters object.

The syntax for defining the targets of bond order factors is precise but somewhat cumbersome due to the large number of permutations one gets when the number of bodies increases. Oftentimes one does not need such fine control over all the parameters since many of them have the same numerical values. Therefore it is convenient to be able to define the targets in a more compact way.

This method generates the detailed target tables from compact syntax. By default, the method takes a list of list and multiplies each list with the others (note the call for a static method):

```
>>> pysic.BondOrderParameters.expand_symbols_table([ 'Si',
...                                                  ['O', 'C'],
...                                                  ['H', 'O'] ])
[['Si', 'O', 'H'],
 ['Si', 'C', 'H'],
 ['Si', 'O', 'O'],
 ['Si', 'C', 'O']]
```

Other custom types of formatting can be defined with the type parameter.

For type 'triplet', the target list is created for triplets A-B-C from an input list of the form:

```
['A', 'B', 'C']
```

Remember that in the symbol table accepted by the BondOrderParameters, one needs to define the B-A and B-C bonds separately and so B appears as the first symbol in the output and the other two appear as second and third (both cases):

```
[['B', 'A', 'C'],  
 ['B', 'C', 'A']]
```

However, for an A-B-A triplet, the A-B bond should only be defined once to prevent double counting. Like the default function, also here several triplets can be defined at once:

```
>>> pysic.BondOrderParameters.expand_symbols_table([ ['H', 'O'],  
...                                                    'Si',  
...                                                    ['O', 'C'] ],  
...                                                    type='triplet')  
[['Si', 'H', 'O'],  
 ['Si', 'O', 'H'],  
 ['Si', 'H', 'C'],  
 ['Si', 'C', 'H'],  
 ['Si', 'O', 'O'],  
 ['Si', 'O', 'C'],  
 ['Si', 'C', 'O']]
```

Parameters:

symbol_list: list of strings list to be expanded to a table type: string specifies a custom way of generating the table

4.3.3 Geometry

class `pysic.utility.geometry.Cell` (*vector1, vector2, vector3, pbc*)

Cell describing the simulation volume of a subvolume.

This class can be used by the user for coordinate manipulation. Note however, that ASE does not use on this class, as it is part of Pysic. The class is merely a tool for examining the geometry.

Parameters:

vector1: list of doubles 3-vector specifying the first vector spanning the cell \mathbf{v}_1

vector2: list of doubles 3-vector specifying the second vector spanning the cell \mathbf{v}_2

vector3: list of doubles 3-vector specifying the third vector spanning the cell \mathbf{v}_3

pbc: list of logicals three logic switches for specifying periodic boundaries - True denotes periodicity

get_absolute_coordinates (*fractional*)

For the given fractional coordinates, returns the absolute coordinates.

The absolute coordinates are the cell vectors multiplied by the fractional coordinates.

Parameters:

fractional: numpy double 3-vector the fractional coordinates

get_distance (*atom1, atom2, offsets=None*)

Calculates the distance between two atoms.

Offsets are multipliers for the cell vectors to be added to the plain separation vector r_1-r_2 between the atoms.

Parameters:

atom1: ASE Atoms object first atom

atom2: ASE Atoms object second atom

offsets: Numpy integer 3-vector the periodic boundary offsets

get_relative_coordinates (*coordinates*)

Returns the coordinates of the given atom in fractional coordinates.

The absolute position of the atom is given by multiplying the cell vectors by the fractional coordinates.

Parameters:

coordinates: numpy double 3-vector the absolute coordinates

get_separation (*atom1*, *atom2*, *offsets=None*)

Returns the separation vector between two atoms, r_1-r_2 .

Offsets are multipliers for the cell vectors to be added to the plain separation vector r_1-r_2 between the atoms.

Parameters:

atom1: ASE Atoms object first atom

atom2: ASE Atoms object second atom

offsets: Numpy integer 3-vector the periodic boundary offsets

get_wrapped_coordinates (*coordinates*)

Wraps the coordinates of the given atom inside the simulation cell.

This method return the equivalent position (with respect to the periodic boundaries) of the atom inside the cell.

For instance, if the cell is spanned by vectors of length 10.0 in directions of x , y , and z , an the coordinates $[-1.0, 12.0, 3.0]$ wrap to $[9.0, 2.0, 3.0]$.

Parameters:

coordinates: numpy double 3-vector the absolute coordinates

4.4 Pysic Fortran module

Pysic Fortran (`pysic_fortran`) is the Python interface of the Fortran core generated using `f2py` from `PyInterface.F90`. This is the only Fortran source file of Pysic that should be wrapped with `f2py`: the rest of the core needs to be directly compiled with a Fortran compiler to `.mod` Fortran modules.

The module is naturally accessible from within Python, but usually there should be no need to directly invoke its functions as `pysic` defines a more refined interface to the Fortran core, mainly through the class `Pysic`. It is assumed that the arguments passed to the functions have proper data types and array dimensions, and that they are called in such an order that the necessary memory allocations have been done within Fortran before data structures are accessed. Methods in `Pysic` do this automatically and are thus much safer to use than directly calling the functions in this module. The Fortran routines are documented here mostly for development purposes.

4.4.1 Modules of the Fortran core of Pysic

`pysic_interface` (`PyInterface.f90`)

`Pysic_interface` is an interface module between the Python and Fortran sides of `pysic`. When `pysic` is compiled, only this file is interfaced to Python via `f2py` while the rest of the Fortran source files are directly compiled to `.mod` Fortran modules. The main reason for this is that F90 derived types are used extensively in the core and these are not yet (2011) supported by `f2py` (although the support for derived types is planned in third generation `f2py`). Because of this, most data is passed from `pysic` to `pysic_interface` (`PyInterface.f90`) as NumPy arrays, and conversions from objects is needed on both sides. This is cumbersome and adds overhead, but it is not an efficiency issue since most of the information is only passed from Python to Fortran once and saved. Even during a molecular dynamics simulation only the forces, coordinates and momenta of atoms need to be communicated through the interface, which is naturally and efficiently handled using just numeric arrays anyway.

Another limitation in current `f2py` is handling of string arrays. To overcome this, string arrays are converted to integer arrays and back using simple mapping functions in `pysic_utility` and `utility` (`Utility.f90`).

Due to the current limitations of `f2py`, no derived types can appear in the module. This severely limits what the module can do, and therefore the module has been by design made to be very light in terms of functionality: No data is stored in the module and almost all routines simply redirect the call to a submodule, most often `pysic_core` (`Core.f90`). In the descriptions of the routines in this documentation, links are provided to the submodule routines that the call is directed to, if the routine is just a redirect of the call.

Full documentation of subroutines in `pysic_interface`

`add_bond_order_factor` (*`n_targets`, `n_params`, `n_split`, `bond_name`, `parameters`,
`param_split`, `cutoff`, `smooth_cut`, `elements`, `orig_elements`,
`group_index`*)

Creates a bond order factor in the core. The memory must have been allocated first using `allocate_potentials`.

Calls `core_add_bond_order_factor()`

Parameters:

`n_targets`: integer *intent(in)* scalar number of targets (interacting bodies)

`n_params`: integer *intent(in)* scalar number of parameters

`n_split`: integer *intent(in)* scalar number of subsets in the list of parameters, should equal `n_targets`

`bond_name`: character(len=*) *intent(in)* scalar bond order factor names

`parameters`: double precision *intent(in)* size(`n_params`) numeric parameters

`param_split`: integer *intent(in)* size(`n_split`) the numbers of parameters for 1-body, 2-body etc.

`cutoff`: double precision *intent(in)* scalar interaction hard cutoff

`smooth_cut`: double precision *intent(in)* scalar interaction soft cutoff

`elements`: integer *intent(in)* size(2, `n_targets`) atomic symbols specifying the elements the interaction acts on

`orig_elements`: integer *intent(in)* size(2, `n_targets`) original atomic symbols specifying the elements the interaction acts on

`group_index`: integer *intent(in)* scalar index denoting the potential to which the factor is connected

add_potential (*n_targets, n_params, pot_name, parameters, cutoff, smooth_cut, elements, tags, indices, orig_elements, orig_tags, orig_indices, pot_index*)
 Creates a potential in the core. The memory must have been allocated first using `allocate_potentials`.
 Calls `core_add_potential()`
 Parameters:

n_targets: integer *intent(in) scalar* number of targets (interacting bodies)
n_params: integer *intent(in) scalar* number of parameters
pot_name: character(len=*) *intent(in) scalar* potential names
parameters: double precision *intent(in) size(n_params)* numeric parameters
cutoff: double precision *intent(in) scalar* interaction hard cutoff
smooth_cut: double precision *intent(in) scalar* interaction soft cutoff
elements: integer *intent(in) size(2, n_targets)* atomic symbols specifying the elements the interaction acts on
tags: integer *intent(in) size(n_targets)* tags specifying the atoms the interaction acts on
indices: integer *intent(in) size(n_targets)* indices specifying the atoms the interaction acts on
orig_elements: integer *intent(in) size(2, n_targets)* original atomic symbols specifying the elements the interaction acts on
orig_tags: integer *intent(in) size(n_targets)* original tags specifying the atoms the interaction acts on
orig_indices: integer *intent(in) size(n_targets)* original indices specifying the atoms the interaction acts on
pot_index: integer *intent(in) scalar* index of the potential

allocate_bond_order_factors (*n_bonds*)
 Allocates memory for storing bond order parameters for describing the atomic interactions. Similar to the `allocate_potentials` routine.
 Calls `core_allocate_bond_order_factors()`
 Parameters:

n_bonds: integer *intent(in) scalar* number of bond order factors

allocate_bond_order_storage (*n_atoms, n_groups, n_factors*)
 Allocates memory for storing bond order factors for describing the atomic interactions. The difference to `allocate_bond_order_factors` is that this method allocates space for arrays used in storing actual calculated bond order factors. The other routine allocates space for storing the parameters used in the calculations.
 Calls `core_allocate_bond_order_storage()`
 Parameters:

n_atoms: integer *intent(in) scalar* number of atoms
n_groups: integer *intent(in) scalar* number of bond order groups
n_factors: integer *intent(in) scalar* number of bond order parameters

allocate_potentials (*n_pots*)
 Allocates memory for storing potentials for describing the atomic interactions. It is more convenient to loop through the potentials and format them in a suitable way in python than in fortran. Therefore

the core is first called through this routine in order to allocate memory for the potentials. Then, each potential is created individually.

Calls `core_allocate_potentials()`

Parameters:

n_pots: integer *intent(in) scalar* number of potentials

calculate_bond_order_factors (*n_atoms, group_index, bond_orders*)

Returns bond order factors of the given group for all atoms. The group index is an identifier for the bond order parameters which are used for calculating one and the same factors. In practice, the Coordinators in pysic are indexed and this indexing is copied in the core. Thus the group index specifies the coordinator / potential.

Calls `core_get_bond_order_factors()`

Parameters:

n_atoms: integer *intent(in) scalar* number of atoms

group_index: integer *intent(in) scalar* index for the bond order factor group

bond_orders: double precision *intent(out) size(n_atoms)* the calculated bond order factors

calculate_bond_order_gradients (*n_atoms, group_index, atom_index, gradients*)

Returns bond order factors gradients of the given group. The gradients of all factors are given with respect to moving the given atom. The group index is an identifier for the bond order parameters which are used for calculating one and the same factors. In practice, the Coordinators in pysic are indexed and this indexing is copied in the core. Thus the group index specifies the coordinator / potential.

Calls `core_get_bond_order_sums()`

and `core_calculate_bond_order_gradients()`

Parameters:

n_atoms: integer *intent(in) scalar* number of atoms

group_index: integer *intent(in) scalar* an index denoting the potential to which the factor is connected

atom_index: integer *intent(in) scalar* index of the atom with respect to which the factors are differentiated

gradients: double precision *intent(out) size(3, n_atoms)* the calculated bond order gradients

calculate_bond_order_gradients_of_factor (*n_atoms, group_index, atom_index, gradients*)

Returns bond order factors gradients of the given group. The gradients of the given factors is given with respect to moving all atoms. The group index is an identifier for the bond order parameters which are used for calculating one and the same factors. In practice, the Coordinators in pysic are indexed and this indexing is copied in the core. Thus the group index specifies the coordinator / potential.

Calls `core_get_bond_order_sums()`

and `core_calculate_bond_order_gradients_of_factor()`

Parameters:

n_atoms: integer *intent(in) scalar* number of atoms

group_index: integer *intent(in) scalar* an index denoting the potential to which the factor is connected

atom_index: integer *intent(in) scalar* index of the atom whose factor is differentiated

gradients: double precision *intent(out) size(3, n_atoms)* the calculated bond order gradients

calculate_electronegativities (*n_atoms, enegs*)
Returns electronegativities of the particles

Calls `core_calculate_electronegativities()`

Parameters:

n_atoms: integer *intent(in) scalar* number of atoms

enegs: double precision *intent(out) size(n_atoms)* array of electronegativities on all atoms

calculate_energy (*n_atoms, energy*)
Returns the total potential energy of the system

Calls `core_calculate_energy()`

Parameters:

n_atoms: integer *intent(in) scalar* number of atoms

energy: double precision *intent(out) scalar* total potential energy

calculate_forces (*n_atoms, forces, stress*)
Returns forces acting on the particles and the stress tensor

Calls `core_calculate_forces()`

Parameters:

n_atoms: integer *intent(in) scalar* number of atoms

forces: double precision *intent(out) size(3, n_atoms)* array of forces on all atoms

stress: double precision *intent(out) size(6)* array containing the components of the stress tensor
(in order *xx, yy, zz, yz, xz, xy*)

create_atoms (*n_atoms, masses, charges, positions, momenta, tags, elements*)
Creates atomic particles. Atoms are handled as custom fortran types `atom` in the core. Currently `f2py` does not support direct creation of types from Python, so instead all the necessary data is passed from Python as arrays and reassembled as types in Fortran. This is not much of an added overhead - the memory allocation itself already makes this a routine one does not wish to call repeatedly. Instead, one should call the routines for updating atoms whenever the actual atoms do not change (e.g., between MD timesteps).

Calls `core_generate_atoms()`

Parameters:

n_atoms: integer *intent(in) scalar* number of atoms

masses: double precision *intent(in) size(n_atoms)* masses of atoms

charges: double precision *intent(in) size(n_atoms)* electric charges of atoms

positions: double precision *intent(in) size(3, n_atoms)* coordinates of atoms

momenta: double precision *intent(in) size(3, n_atoms)* momenta of atoms

tags: integer *intent(in) size(n_atoms)* numeric tags for the atoms

elements: integer *intent(in) size(2, n_atoms)* atomic symbols of the atoms

create_bond_order_factor_list ()

Similarly to the potential lists, also list containing all the bond order factors that may affect an atom are stored in a list.

Calls `core_assign_bond_order_factor_indices ()`

create_cell (vectors, inverse, periodicity)

Creates a supercell for containing the calculation geometry. Also the inverse cell matrix must be given, although it is not checked that the given inverse actually is the true inverse.

Calls `core_create_cell ()`

Parameters:

vectors: double precision intent(in) size(3, 3) A 3x3 matrix containing the vectors spanning the supercell. The first index runs over xyz and the second index runs over the three vectors.

inverse: double precision intent(in) size(3, 3) A 3x3 matrix containing the inverse matrix of the one given in vectors, i.e. $M^{-1} * M = I$ for the two matrices. Since the latter represents a cell of non-zero volume, this inverse must exist. It is not tested that the given matrix actually is the inverse, the user must make sure it is.

periodicity: logical intent(in) size(3) A 3-element vector containing logical tags specifying if the system is periodic in the directions of the three vectors spanning the supercell.

create_neighbor_list (n_nbs, atom_index, neighbors, offsets)

Creates neighbor lists for a single atom telling it which other atoms are in its immediate neighborhood. The neighbor list must be precalculated, this method only stores them in the core. The list must contain an array storing the indices of the neighboring atoms as well as the supercell offsets. The offsets are integer triplets showing how many times must the supercell vectors be added to the position of the neighbor to find the neighboring image in a periodic system. Note that if the system is small, one atom can in principle appear several times in the neighbor list.

Calls `core_create_neighbor_list ()`

Parameters:

n_nbs: integer intent(in) scalar number of neighbors

atom_index: integer intent(in) scalar index of the atom for which the neighbor list is created

neighbors: integer intent(in) size(n_nbs) An array containing the indices of the neighboring atoms

offsets: integer intent(in) size(3, n_nbs) An array containing vectors specifying the offsets of the neighbors in periodic systems.

create_potential_list ()

Creates a list of indices for all atoms showing which potentials act on them. The user may define many potentials to sum up the potential energy of the system. However, if some potentials only act on certain atoms, they will be redundant for the other atoms. The potential lists are lists given to each atom containing the potentials which can act on the atom.

Calls `core_assign_potential_indices ()`

description_of_bond_order_factor (bond_name, description)

Returns a description of the given bond order factor

Calls `get_description_of_bond_order_factor ()`

Parameters:

bond_name: character(len=*) intent(in) scalar name of the bond order factor

description: character(len=500) intent(out) scalar description of the bond order factor

description_of_potential (*pot_name, description*)

Returns a description of the given potential

Calls `get_description_of_potential()`

Parameters:

pot_name: *character(len=*) intent(in) scalar* name of the potential

description: *character(len=500) intent(out) scalar* description of the potential

descriptions_of_parameters_of_bond_order_factor (*bond_name, n_targets, param_notes*)

Lists descriptions for parameters the given bond order factor. Output is an array of integers. This is because `f2py` doesn't currently support string arrays. So, the characters are translated to integers and back in fortran and python. This adds a bit of overhead, but the routine is only invoked on user command so it doesn't matter.

Calls `get_descriptions_of_parameters_of_bond_order_factor()`

Parameters:

bond_name: *character(len=*) intent(in) scalar* name of the bond order factor

n_targets: *integer intent(in) scalar* number of targets

param_notes: *integer intent(out) size(100, 12)* descriptions of the parameters

descriptions_of_parameters_of_potential (*pot_name, param_notes*)

Lists descriptions for parameters the given potential. Output is an array of integers. This is because `f2py` doesn't currently support string arrays. So, the characters are translated to integers and back in fortran and python. This adds a bit of overhead, but the routine is only invoked on user command so it doesn't matter.

Calls `get_descriptions_of_parameters_of_potential()`

Parameters:

pot_name: *character(len=*) intent(in) scalar* name of the potential

param_notes: *integer intent(out) size(100, 12)* descriptions of the parameters

distribute_mpi (*n_atoms*)

Distributes atoms among the processors. In the MPI scheme, atoms are distributed among the cpus for force and energy calculations. This routine initializes the arrays that tell each cpu which atoms it has to calculate interactions for. It can be called before the atoms are created in the core but one has to make sure the number of atoms specified in the last call matches the number of atoms in the core when a calculation is invoked.

Calls `mpi_distribute()`

Parameters:

n_atoms: *integer intent(in) scalar* number of atoms

examine_atoms ()

Prints some information about the atoms allocated in the core. This is mainly for debugging, as the python side should always dictate what is in the core.

Calls `list_atoms()`

examine_bond_order_factors ()

Prints some information about the bond order factors allocated in the core. This is mainly for debugging, as the python side should always dictate what is in the core.

Calls `list_bonds()`

examine_cell()

Prints some information about the supercell allocated in the core. This is mainly for debugging, as the python side should always dictate what is in the core.

Calls `list_cell()`

examine_potentials()

Prints some information about the potential allocated in the core. This is mainly for debugging, as the python side should always dictate what is in the core.

Calls `list_interactions()`

finish_mpi()

Finishes MPI for parallel calculations.

Calls `mpi_finish()`

generate_neighbor_lists(*n_atoms*, *cutoffs*)

calculates and allocates neighbor lists

Parameters:

n_atoms: integer *intent(in)* scalar

cutoffs: double precision *intent(in)* size(*n_atoms*)

get_cell_vectors(*vectors*)

Returns the vectors defining the simulation supercell.

Calls `core_get_cell_vectors()`

Parameters:

vectors: double precision **intent(out)** size(3, 3) A 3x3 matrix containing the vectors spanning the supercell. The first index runs over xyz and the second index runs over the three vectors.

get_cpu_id(*id*)

Returns the MPI cpu id number, which is an integer between 0 and $n_{\text{cpus}} - 1$, where n_{cpus} is the total number of cpus.

Parameters:

id: integer **intent(out)** scalar cpu id number in MPI - 0 in serial mode

get_ewald_energy(*real_cut*, *reciprocal_cut*, *sigma*, *epsilon*, *energy*)

Debugging routine for Ewald

Parameters:

real_cut: double precision *intent(in)* scalar

reciprocal_cut: integer *intent(in)* size(3)

sigma: double precision *intent(in)* scalar

epsilon: double precision *intent(in)* scalar

energy: double precision **intent(out)** scalar

get_mpi_list_of_atoms(*n_atoms*, *cpu_atoms*)

Returns a logical array containing true for every atom that is allocated to this cpu, and false for all other atoms.

Parameters:

n_atoms: integer *intent(in)* scalar number of atoms

cpu_atoms: logical intent(out) size(*n_atoms*) array of logical values showing which atoms are marked to be handled by this cpu

get_neighbor_list_of_atom (*atom_index*, *n_neighbors*, *neighbors*, *offsets*)

Parameters:

atom_index: integer intent(in) scalar

n_neighbors: integer intent(in) scalar

neighbors: integer intent(out) size(*n_neighbors*)

offsets: integer intent(out) size(3, *n_neighbors*)

get_number_of_atoms (*n_atoms*)

Counts the number of atoms in the current core

Calls `core_get_number_of_atoms()`

Parameters:

n_atoms: integer intent(out) scalar number of atoms

get_number_of_cpus (*ncpu*)

Returns the MPI cpu count

Parameters:

ncpu: integer intent(out) scalar the total number of cpus available

get_number_of_neighbors_of_atom (*atom_index*, *n_neighbors*)

Parameters:

atom_index: integer intent(in) scalar

n_neighbors: integer intent(out) scalar

is_bond_order_factor (*string*, *is_ok*)

Tells whether a given keyword defines a bond order factor or not

Calls `is_valid_bond_order_factor()`

Parameters:

string: character(len=*) intent(in) scalar name of a bond order factor

is_ok: logical intent(out) scalar true if string is a name of a bond order factor

is_potential (*string*, *is_ok*)

Tells whether a given keyword defines a potential or not

Calls `is_valid_potential()`

Parameters:

string: character(len=*) intent(in) scalar name of a potential

is_ok: logical intent(out) scalar true if string is a name of a potential

list_valid_bond_order_factors (*n_bonds*, *bond_factors*)

Lists all the keywords which define a bond order factor

Calls `list_bond_order_factors()`

Parameters:

n_bonds: integer intent(in) scalar number of bond order factor types

bond_factors: integer intent(out) size(11, *n_bonds*) names of the bond order factor types

list_valid_potentials (*n_pots, potentials*)

Lists all the keywords which define a potential

Calls `list_potentials()`

Parameters:

n_pots: integer *intent(in) scalar* number of potential types

potentials: integer *intent(out) size(11, n_pots)* names of the potential types

names_of_parameters_of_bond_order_factor (*bond_name, n_targets, param_names*)

Lists the names of parameters the given bond order factor knows. Output is an array of integers. This is because `f2py` doesn't currently support string arrays. So, the characters are translated to integers and back in fortran and python. This adds a bit of overhead, but the routine is only invoked on user command so it doesn't matter.

Calls `get_names_of_parameters_of_bond_order_factor()`

Parameters:

bond_name: character(*len=**) *intent(in) scalar* name of the bond order factor

n_targets: integer *intent(in) scalar* number of targets

param_names: integer *intent(out) size(10, 12)* names of the parameters

names_of_parameters_of_potential (*pot_name, param_names*)

Lists the names of parameters the given potential knows. Output is an array of integers. This is because `f2py` doesn't currently support string arrays. So, the characters are translated to integers and back in fortran and python. This adds a bit of overhead, but the routine is only invoked on user command so it doesn't matter.

Calls `get_names_of_parameters_of_potential()`

Parameters:

pot_name: character(*len=**) *intent(in) scalar* name of the potential

param_names: integer *intent(out) size(10, 12)* names of the parameters

number_of_bond_order_factors (*n_bonds*)

Tells the number of differently named bond order factors the core knows

Calls `get_number_of_bond_order_factors()`

Parameters:

n_bonds: integer *intent(out) scalar* number of bond order factors

number_of_parameters_of_bond_order_factor (*bond_name, n_targets, n_params*)

Tells how many numeric parameters a bond order factor incorporates

Calls `get_number_of_parameters_of_bond_order_factor()`

Parameters:

bond_name: character(*len=**) *intent(in) scalar* name of the bond order factor

n_targets: integer *intent(in) scalar* number of targets

n_params: integer *intent(out) scalar* number of parameters

number_of_parameters_of_potential (*pot_name, n_params*)

Tells how many numeric parameters a potential incorporates

Calls `get_number_of_parameters_of_potential()`

Parameters:

pot_name: *character(len=*) intent(in) scalar* name of the potential

n_params: *integer intent(out) scalar* number of parameters

number_of_potentials (*n_pots*)

Tells the number of differently named potentials the core knows

Calls `get_number_of_potentials()`

Parameters:

n_pots: *integer intent(out) scalar* number of potentials

number_of_targets_of_bond_order_factor (*bond_name, n_target*)

Tells how many targets a bond order factor has, i.e., is it many-body

Calls `get_number_of_targets_of_bond_order_factor()`

Parameters:

bond_name: *character(len=*) intent(in) scalar* name of the bond order factor

n_target: *integer intent(out) scalar* number of targets

number_of_targets_of_potential (*pot_name, n_target*)

Tells how many targets a potential has, i.e., is it a many-body potential

Calls `get_number_of_targets_of_potential()`

Parameters:

pot_name: *character(len=*) intent(in) scalar* name of the potential

n_target: *integer intent(out) scalar* number of targets

release ()

Deallocates all the arrays in the core

Calls `core_release_all_memory()`

set_ewald_parameters (*n_atoms, real_cut, reciprocal_cut, sigma, epsilon, scaler*)

Sets the parameters for Ewald summation in the core.

Parameters:

n_atoms: *integer intent(in) scalar*

real_cut: *double precision intent(in) scalar* the real-space cutoff

reciprocal_cut: *integer intent(in) size(3)* the k-space cutoffs

sigma: *double precision intent(in) scalar* the split parameter

epsilon: *double precision intent(in) scalar* electric constant

scaler: *double precision intent(in) size(n_atoms)* scaling factors for the individual charges

start_bond_order_factors ()

Initializes the bond order factors. A routine is called to generate descriptors for potentials. These descriptors are needed by the python interface in order to directly inquire the core on the types of factors available.

Calls `initialize_bond_order_factor_characterizers()`

start_mpi()

Initializes MPI for parallel calculations.

Calls `mpi_initialize()`

start_potentials()

Initializes the potentials. A routine is called to generate descriptors for potentials. These descriptors are needed by the python interface in order to directly inquire the core on the types of potentials available.

Calls `initialize_potential_characterizers()`

start_rng(seed)

Initialize Mersenne Twister random number generator.

A seed number has to be given. In case we run in MPI mode, the master cpu will broadcast its seed to all other cpus to ensure that the random number sequences match in all the cpus.

Parameters:

seed: integer *intent(in) scalar* a seed for the random number generator

sync_mpi()

Syncs MPI. This just calls `mpi_barrier`, so it makes all cpus wait until everyone is at this particular point in execution.

Calls `mpi_sync()`

update_atom_charges(n_atoms, charges)

Updates the charges of existing atoms. This method does not allocate memory and so the atoms must already exist in the core.

Calls `core_update_atom_charges()`

Parameters:

n_atoms: integer *intent(in) scalar* number of atoms

charges: double precision *intent(in) size(n_atoms)* new charges for the atoms

update_atom_coordinates(n_atoms, positions, momenta)

Updates the positions and velocities of existing atoms. This method does not allocate memory and so the atoms must already exist in the core.

Calls `core_update_atom_coordinates()`

Parameters:

n_atoms: integer *intent(in) scalar* number of atoms

positions: double precision *intent(in) size(3, n_atoms)* new coordinates for the atoms

momenta: double precision *intent(in) size(3, n_atoms)* new momenta for the atoms

pysic_core (Core.f90)

Core, true to its name, is the heart of the Fortran core of Pysic. It contains the data structures defining the simulation geometry and interactions and defines the central routines for calculating the total energy of and the forces acting on the system.

Many of the routines in *pysic_interface (PyInterface.f90)* which *f2py* interfaces to Python are simply calling routines here.

Full documentation of global variables in `pysic_core`**atoms**

`type(atom) pointer size(:)`

an array of `atom` objects representing the system

atoms_created

logical *scalar*

initial value = .false.

logical tag indicating if atom storing arrays have been created

bond_factors

`type(bond_order_parameters) pointer size(:)`

an array of `bond_order_parameters` objects representing bond order factors modifying the potentials

bond_factors_allocated

logical *scalar*

initial value = .false.

logical tag indicating if bond order parameter storing arrays have been allocated

bond_storage_allocated

logical *scalar*

initial value = .false.

logical tag indicating if bond order factor storing arrays have been allocated

cell

`type(supercell) scalar`

a `supercell` object representing the simulation cell

electronegativity_evaluation_index

integer *scalar parameter*

initial value = 3

energy_evaluation_index

integer *scalar parameter*

initial value = 1

evaluate_ewald

logical *scalar*

initial value = .false.

switch for enabling Ewald summation of coulomb interactions

ewald_allocated

logical *scalar*

initial value = .false.

ewald_cutoff

double precision *scalar*

ewald_epsilon

double precision *scalar*

ewald_k_cutoffsinteger *size*(3)**ewald_scaler**double precision *pointer size*(:)**ewald_sigma**double precision *scalar***force_evaluation_index**integer *scalar parameter**initial value* = 2**group_index_save_slot**integer *pointer size*(:)**interactions**type(potential) *pointer size*(:)an array of `potential` objects representing the interactions**n_bond_factors**integer *scalar**initial value* = 0**n_interactions**integer *scalar**initial value* = 0

number of potentials

n_saved_bond_order_factorsinteger *scalar**initial value* = 0

number of saved bond order factors

potentials_allocatedlogical *scalar**initial value* = .false.

logical tag indicating if potential storing arrays have been allocated

saved_bond_order_factorsdouble precision *pointer size*(:, :)

Array for storing calculated bond order factors. Indexing: (atom index, group_index_save_slot(group index))

saved_bond_order_gradientsdouble precision *pointer size*(:, :, :, :)

Array for storing calculated bond order gradients. Indexing: (xyz, atom index, group_index_save_slot(group index), target index)

saved_bond_order_sumsdouble precision *pointer size*(:, :)

Array for storing calculated bond order sums. Indexing: (atom index, group_index_save_slot(group index))

saved_bond_order_virials

double precision *pointer size*(:, :, :)

Array for storing calculated bond order virials. Indexing: (xyz, group_index_save_slot(group index), target index)

use_saved_bond_order_factors

logical *scalar*

initial value = .false.

Logical tag which enables / disables bond order saving. If true, bond order calculation routines try to find the precalculated factors in the saved bond order arrays instead of calculating.

use_saved_bond_order_gradients

integer *pointer size*(:, :)

Array storing the atom index of the bond gradient stored for indices (group index, target index). Since gradients are needed for all factors (N) with respect to moving all atoms (N), storing them all would require an N x N matrix. Therefore only some are stored. This array is used for searching the stroage to see if the needed gradient is there or needs to be calculated.

Full documentation of subroutines in pysic_core

core_add_bond_order_factor (*n_targets*, *n_params*, *n_split*, *bond_name*, *parameters*,
param_split, *cutoff*, *smooth_cut*, *elements*, *orig_elements*,
group_index)

Creates one additional bond_order_factor in the core. The routine assumes that adequate memory has been allocated already using core_allocate_bond_order_factors.

When the bond order parameters in the Python interface are imported to the Fortran core, the target specifiers (elements) are permuted to create all equivalent bond order parameters. That is, if we have parameters for Si-O, both Si-O and O-Si parameters are created. This is because the energy and force calculation loops only deal with atom pairs A-B once (so only A-B or B-A is considered, not both) and if, say, the loop only finds an O-Si pair, it is important to apply the Si-O parameters also on that pair. In some cases, such as with the tersoff factor affecting triplets (A-B-C), the contribution is not symmetric for all the atoms. Therefore it is necessary to also store the original targets of the potential as specified in the Python interface. These are to be given in the 'orig_elements' lists.

called from PyInterface: `add_bond_order_factor()`

Parameters:

n_targets: integer *intent(in)* *scalar* number of targets (interacting bodies)

n_params: integer *intent(in)* *scalar* number of parameters

n_split: integer *intent(in)* *scalar* number of subsets in the list of parameters, should equal n_targets

bond_name: character(len=*) *intent(in)* *scalar* bond order factor names

parameters: double precision *intent(in)* *size*(n_params) numeric parameters

param_split: integer *intent(in)* *size*(n_split) the numbers of parameters for 1-body, 2-body etc.

cutoff: double precision *intent(in)* *scalar* interaction hard cutoff

smooth_cut: double precision *intent(in)* *scalar* interaction soft cutoff

elements: character(len=label_length) *intent(in)* *size*(n_targets) atomic symbols specifying the elements the interaction acts on

orig_elements: *character(len=label_length) intent(in) size(n_targets)* original atomic symbols specifying the elements the interaction acts on

group_index: *integer intent(in) scalar* index denoting the potential to which the factor is connected

core_add_potential (*n_targets, n_params, pot_name, parameters, cutoff, smooth_cut, elements, tags, indices, orig_elements, orig_tags, orig_indices, pot_index*)

Creates one additional potential in the core. The routine assumes that adequate memory has been allocated already using `core_allocate_potentials`.

When the potentials in the Python interface are imported to the Fortran core, the target specifiers (elements, tags, indices) are permuted to create all equivalent potentials. That is, if we have a potential for Si-O, both Si-O and O-Si potentials are created. This is because the energy and force calculation loops only deal with atom pairs A-B once (so only A-B or B-A is considered, not both) and if, say, the loop only finds an O-Si pair, it is important to apply the Si-O interaction also on that pair. In some cases, such as with the bond-bending potential affecting triplets (A-B-C), the interaction is not symmetric for all the atoms. Therefore it is necessary to also store the original targets of the potential as specified in the Python interface. These are to be given in the 'orig_*' lists.

called from PyInterface: `add_potential()`

Parameters:

n_targets: *integer intent(in) scalar* number of targets (interacting bodies)

n_params: *integer intent(in) scalar* number of parameters

pot_name: *character(len=*) intent(in) scalar* potential names

parameters: *double precision intent(in) size(n_params)* numeric parameters

cutoff: *double precision intent(in) scalar* interaction hard cutoff

smooth_cut: *double precision intent(in) scalar* interaction soft cutoff

elements: *character(len=label_length) intent(in) size(n_targets)* atomic symbols specifying the elements the interaction acts on

tags: *integer intent(in) size(n_targets)* tags specifying the atoms the interaction acts on

indices: *integer intent(in) size(n_targets)* indices specifying the atoms the interaction acts on

orig_elements: *character(len=label_length) intent(in) size(n_targets)* original atomic symbols specifying the elements the interaction acts on

orig_tags: *integer intent(in) size(n_targets)* original tags specifying the atoms the interaction acts on

orig_indices: *integer intent(in) size(n_targets)* original indices specifying the atoms the interaction acts on

pot_index: *integer intent(in) scalar* index of the potential

core_allocate_bond_order_factors (*n_bond_factors*)

Allocates pointers for storing bond order factors.

called from PyInterface: `allocate_bond_order_factors()`

Parameters:

n_bond_factors: *integer intent(in) scalar*

core_allocate_bond_order_storage (*n_atoms, n_groups, n_factors*)

Allocates arrays for storing precalculated values of bond order factors and gradients.

called from PyInterface: `allocate_bond_order_factors()`

Parameters:

n_atoms: integer *intent(in)* scalar number of atoms

n_groups: integer *intent(in)* scalar number of bond order groups

n_factors: integer *intent(in)* scalar number of bond order parameters

core_allocate_potentials (*n_pots*)

Allocates pointers for storing potentials.

called from PyInterface: `allocate_potentials()`

Parameters:

n_pots: integer *intent(in)* scalar number of potentials

core_assign_bond_order_factor_indices ()

This routine finds for each atom the potentials for which the atom is an accepted target at the first position. First position here means that for instance in an A-B-C triplet. A is in first position. Being an accepted target means that the atom has the correct element.

called from PyInterface: `create_bond_order_factor_list()`

core_assign_potential_indices ()

This routine finds for each atom the potentials for which the atom is an accepted target at the first position. First position here means that for instance in an A-B-C triplet. A is in first position. Being an accepted target means that the atom has the correct element, index or tag (one that the potential targets).

called from PyInterface: `create_potential_list()`

core_build_neighbor_lists (*n_atoms, cutoffs*)

Parameters:

n_atoms: integer *intent(in)* scalar

cutoffs: double precision *intent(in)* size(*n_atoms*)

core_calculate_bond_order_factors (*n_atoms, group_index, total_bond_orders*)

Calculates the bond order sums of all atoms for the given group.

For a factor such as

$$b_i = f\left(\sum_j c_{ij}\right)$$

The routine calculates

$$\sum_j c_{ij}.$$

The full bond order factor is then obtained by applying the scaling function f . This is done with `core_post_process_bond_order_factors()`.

Parameters:

n_atoms: integer *intent(in)* scalar number of atoms

group_index: integer *intent(in)* scalar an index denoting the potential to which the factor is connected

total_bond_orders: double precision **intent(out)** *size(n_atoms)* the calculated bond order sums

core_calculate_bond_order_gradients (*n_atoms*, *group_index*, *atom_index*,
raw_sums, *total_gradient*, *total_virial*,
for_factor)

Returns the gradients of bond order factors.

For a factor such as

$$b_i = f\left(\sum_j c_{ij}\right)$$

The routine calculates

$$\nabla_\alpha b_i = f'\left(\sum_j c_{ij}\right) \nabla_\alpha \sum_j c_{ij}.$$

By default, the gradients of all factors i are calculated with respect to moving the given atom α . If *for_factor* is `.true.`, the gradients of the bond factor of the given atom are calculated with respect to moving all atoms.

Parameters:

n_atoms: integer *intent(in)* *scalar* number of atoms

group_index: integer *intent(in)* *scalar* an index denoting the potential to which the factor is connected

atom_index: integer *intent(in)* *scalar* index of the atom with respect to which the factors are differentiated (α), or the atoms whose factor is differentiated (i) if *for_factor* is `.true.`

raw_sums: double precision *intent(in)* *size(n_atoms)* precalculated bond order sums, $\sum_j c_{ij}$, in the above example.

total_gradient: double precision **intent(out)** *size(3, n_atoms)* the calculated bond order gradients $\nabla_\alpha b_i$

total_virial: double precision **intent(out)** *size(6)*

for_factor: logical *intent(in)* *scalar optional* a switch for requesting the gradients for a given i instead of a given α

core_calculate_bond_order_gradients_of_factor (*n_atoms*, *group_index*,
atom_index, *raw_sums*,
total_gradient, *total_virial*)

Returns the gradients of one bond order factor with respect to moving all atoms.

This calls `core_calculate_bond_order_gradients()` with *for_factor* = `.true.`

For a factor such as

$$b_i = f\left(\sum_j c_{ij}\right)$$

The routine calculates

$$\nabla_\alpha b_i = f'\left(\sum_j c_{ij}\right) \nabla_\alpha \sum_j c_{ij}.$$

The gradients of the bond factor of the given atom i are calculated with respect to moving all atoms α .

Parameters:

n_atoms: integer *intent(in)* *scalar* number of atoms

group_index: integer *intent(in) scalar* an index denoting the potential to which the factor is connected

atom_index: integer *intent(in) scalar* index of the atom whose factor is differentiated (i)

raw_sums: double precision *intent(in) size(n_atoms)* precalculated bond order sums, $\sum_j c_{ij}$, in the above example.

total_gradient: double precision *intent(out) size(3, n_atoms)* the calculated bond order gradients $\nabla_{\alpha} b_i$

total_virial: double precision *intent(out) size(6)*

core_calculate_electronegativities (n_atoms , $total_enegs$)

Calculates electronegativity forces acting on all atomic charges of the system.

The routine calculates the electronegativities

$$\chi_{\alpha} = -\frac{\partial V}{\partial q_{\alpha}}$$

for all atoms α . This is done according to the the structure and potentials allocated in the core, so the routine does not accept arguments. Instead, the core modifying routines such as `core_generate_atoms()` must be called first to set up the calculation.

called from PyInterface: `calculate_electronegativities()`

Parameters:

n_atoms: integer *intent(in) scalar* number of atoms

totalenegs: double precision *intent(out) size(n_atoms)* an array containing the calculated charge forces for all atoms

core_calculate_energy (n_atoms , $total_energy$)

Calculates the total potential energy of the system.

This is done according to the the structure and potentials allocated in the core, so the routine does not accept arguments. Instead, the core modifying routines such as `core_generate_atoms()` must be called first to set up the calculation.

called from PyInterface: `calculate_energy()`

Parameters:

n_atoms: integer *intent(in) scalar* number of atoms

total_energy: double precision *intent(out) scalar* calculated total potential energy

core_calculate_forces (n_atoms , $total_forces$, $total_stress$)

Calculates forces acting on all atoms of the system.

The routine calculates the potential gradient

$$\mathbf{F}_{\alpha} = -\nabla_{\alpha} V$$

for all atoms α . This is done according to the the structure and potentials allocated in the core, so the routine does not accept arguments. Instead, the core modifying routines such as `core_generate_atoms()` must be called first to set up the calculation.

called from PyInterface: `calculate_forces()`

Parameters:

n_atoms: integer *intent(in) scalar* number of atoms

total_forces: double precision *intent(out)* *size(3, n_atoms)* an array containing the calculated forces for all atoms

total_stress: double precision *intent(out)* *size(6)* as array containing the calculated stress tensor

core_clear_atoms ()

Deallocates the array of atoms in the core, if allocated.

core_clear_bond_order_factors ()

Deallocates pointers for bond order factors (the parameters)

core_clear_bond_order_storage ()

Deallocates pointers for bond order factors (the precalculated factor values).

core_clear_potentials ()

Deallocates pointers for potentials

core_create_cell (*vectors, inverse, periodicity*)

Creates a supercell for containing the calculation geometry.

called from PyInterface: `create_cell` ()

Parameters:

vectors: double precision *intent(in)* *size(3, 3)* A 3x3 matrix containing the vectors spanning the supercell. The first index runs over xyz and the second index runs over the three vectors.

inverse: double precision *intent(in)* *size(3, 3)* A 3x3 matrix containing the inverse matrix of the one given in vectors, i.e. $A * B = I$ for the two matrices. Since the latter represents a cell of non-zero volume, this inverse must exist. It is not tested that the given matrix actually is the inverse, the user must make sure it is.

periodicity: logical *intent(in)* *size(3)* A 3-element vector containing logical tags specifying if the system is periodic in the directions of the three vectors spanning the supercell.

core_create_neighbor_list (*n_nbs, atom_index, neighbors, offsets*)

Assigns a precalculated neighbor list to a single atom of the given index. The neighbor list must be precalculated, this method only stores them in the core. The list must contain an array storing the indices of the neighboring atoms as well as the supercell offsets. The offsets are integer triplets showing how many times must the supercell vectors be added to the position of the neighbor to find the neighboring image in a periodic system. For example, let the supercell be:

```
[[1.0, 0, 0], [0, 1.0, 0], [0, 0, 1.0]],
```

i.e., a unit cube, with periodic boundaries. Now, if we have particles with coordinates:

```
a = [1.5, 0.5, 0.5]
```

```
b = [0.4, 1.6, 3.3]
```

the closest separation vector $\mathbf{r}_b - \mathbf{r}_a$ between the particles is:

```
[-.1, .1, -.2]
```

obtained if we add the vector of periodicity:

```
[1.0, -1.0, -3.0]
```

to the coordinates of particle b. The offset vector (for particle b, when listing neighbors of a) is then:

```
[1, -1, -3]
```

Note that if the system is small, one atom can in principle appear several times in the neighbor list with different offsets.

called from PyInterface: `create_neighbor_list()`

Parameters:

n_nbs: integer *intent(in)* scalar number of neighbors

atom_index: integer *intent(in)* scalar index of the atom for which the neighbor list is created

neighbors: integer *intent(in)* size(*n_nbs*) An array containing the indices of the neighboring atoms

offsets: integer *intent(in)* size(3, *n_nbs*) An array containing vectors specifying the offsets of the neighbors in periodic systems.

core_create_space_partitioning (*max_cutoff*)

Parameters:

max_cutoff: double precision *intent(in)* scalar

core_debug_dump (*forces*)

Write atomic coordinates and other info in a file. This is only for debugging.

Parameters:

forces: double precision *intent(in)* size(:, :)

core_empty_bond_order_gradient_storage (*index*)

Clears bond order factor gradients (the precalculated gradient values) but does not deallocate the arrays. If an index is given, then only that column is emptied.

Parameters:

index: integer *intent(in)* scalar *optional* the column to be emptied

core_empty_bond_order_storage ()

Clears bond order factors (the precalculated factor values) but does not deallocate the arrays.

core_evaluate_local_doublet (*n_atoms*, *atom_doublet*, *index1*, *index2*, *test_index1*, *interaction_indices*, *separations*, *directions*, *distances*, *calculation_type*, *energy*, *forces*, *enegs*, *stress*, *many_bodies_found*)

Parameters:

n_atoms: integer *intent(in)* scalar

atom_doublet: type(atom) *intent(in)* size(2)

index1: integer *intent(in)* scalar

index2: integer *intent(in)* scalar

test_index1: integer *intent(in)* scalar

interaction_indices: integer *intent()* pointer size(:)

separations: double precision *intent(in)* size(3, 1)

directions: double precision *intent(in)* size(3, 1)

distances: double precision *intent(in)* size(1)

calculation_type: integer *intent(in)* scalar

energy: double precision **intent(out)** scalar

forces: double precision **intent(out)** size(3, *n_atoms*)

enegs: double precision **intent(out)** size(*n_atoms*)

stress: double precision **intent(out)** size(6)

many_bodies_found: logical **intent(out)** scalar

core_evaluate_local_quaduplet (*n_atoms, atom_quaduplet, index1, index2, index3, index4, test_index1, test_index2, test_index3, interaction_indices, separations, directions, distances, calculation_type, energy, forces, enegs, stress, many_bodies_found*)

Parameters:

n_atoms: integer **intent(in)** scalar

atom_quaduplet: type(atom) **intent(in)** size(4)

index1: integer **intent(in)** scalar

index2: integer **intent(in)** scalar

index3: integer **intent(in)** scalar

index4: integer **intent(in)** scalar

test_index1: integer **intent(in)** scalar

test_index2: integer **intent(in)** scalar

test_index3: integer **intent(in)** scalar

interaction_indices: integer **intent()** pointer size(:)

separations: double precision **intent(in)** size(3, 3)

directions: double precision **intent(in)** size(3, 3)

distances: double precision **intent(in)** size(3)

calculation_type: integer **intent(in)** scalar

energy: double precision **intent(out)** scalar

forces: double precision **intent(out)** size(3, n_atoms)

enegs: double precision **intent(out)** size(n_atoms)

stress: double precision **intent(out)** size(6)

many_bodies_found: logical **intent(out)** scalar

core_evaluate_local_singlet (*n_atoms, index1, atom_singlet, interaction_indices, calculation_type, energy, forces, enegs*)

Evaluates the local potential affecting a single atom

Parameters:

n_atoms: integer **intent(in)** scalar number of atoms

index1: integer **intent(in)** scalar index of the atom

atom_singlet: type(atom) **intent(in)** scalar the atom that is targeted

interaction_indices: integer **intent()** pointer size(:) the interactions targeting the given atom

calculation_type: integer **intent(in)** scalar specifies if we are evaluating the energy, forces, or electronegativities

energy: double precision **intent(inout)** scalar calculated energy

forces: double precision **intent(inout)** size(3, n_atoms) calculated forces

enegs: double precision **intent(inout)** size(n_atoms) calculated electronegativities

core_evaluate_local_triplet (*n_atoms, atom_triplet, index1, index2, index3, test_index1, test_index2, interaction_indices, separations, directions, distances, calculation_type, energy, forces, enegs, stress, many_bodies_found*)

Parameters:

n_atoms: integer *intent(in)* scalar

atom_triplet: type(atom) *intent(in)* size(3)

index1: integer *intent(in)* scalar

index2: integer *intent(in)* scalar

index3: integer *intent(in)* scalar

test_index1: integer *intent(in)* scalar

test_index2: integer *intent(in)* scalar

interaction_indices: integer *intent()* pointer size(:)

separations: double precision *intent(in)* size(3, 2)

directions: double precision *intent(in)* size(3, 2)

distances: double precision *intent(in)* size(2)

calculation_type: integer *intent(in)* scalar

energy: double precision **intent(out)** scalar

forces: double precision **intent(out)** size(3, n_atoms)

enegs: double precision **intent(out)** size(n_atoms)

stress: double precision **intent(out)** size(6)

many_bodies_found: logical **intent(out)** scalar

core_fill_bond_order_storage (*n_atoms*)

Fills the storage for bond order factors and bond order sums. This is meant to be called in the beginning of force and energy evaluation. The routine calculates all bond order factors (in parallel, if run in MPI) and stores them. Then during the energy or force calculation, it is sufficient to just look up the needed values in the arrays. The routine does not calculate and store bond factor gradients.

Parameters:

n_atoms: integer *intent(in)* scalar number of atoms

core_generate_atoms (*n_atoms, masses, charges, positions, momenta, tags, elements*)

Creates the atomic particles by invoking a subroutine in the geometry module.

called from PyInterface: `create_atoms()`

Parameters:

n_atoms: integer *intent(in)* scalar number of atoms

masses: double precision *intent(in)* size(n_atoms) masses of atoms

charges: double precision *intent(in)* size(n_atoms) electric charges of atoms

positions: double precision *intent(in)* size(3, n_atoms) coordinates of atoms

momenta: double precision *intent(in)* size(3, n_atoms) momenta of atoms

tags: integer *intent(in)* size(n_atoms) numeric tags for the atoms

elements: **character**(len=label_length) **intent(in)** **size**(n_atoms) atomic symbols of the atoms

core_get_bond_order_factor_of_atom (n_atoms, group_index, atom_index, bond_order_factor)

Returns the bond order factors of the given atom for the given group.

Parameters:

n_atoms: **integer intent(in) scalar** number of atoms

group_index: **integer intent(in) scalar** index for the bond order factor group

atom_index: **integer intent(in) scalar** index of the atom whose bond order factor is returned

bond_order_factor: **double precision intent(out) scalar** the calculated bond order factor

core_get_bond_order_factors (n_atoms, group_index, bond_order_factors)

Returns the bond order factors of all atoms for the given group. The routine tries to find the values in the stored precalculated values first if use_saved_bond_order_factors is true, and saves the calculated values if it does not find them.

Parameters:

n_atoms: **integer intent(in) scalar** number of atoms

group_index: **integer intent(in) scalar** index for the bond order factor group

bond_order_factors: **double precision intent(out) size(n_atoms)** the calculated bond order factors

core_get_bond_order_gradients (n_atoms, group_index, atom_index, slot_index, bond_order_gradients, bond_order_virial)

Returns the gradients of the bond order factor of the given atom with respect to moving all atoms, for the given group. The routine tries to find the values in the stored precalculated values first if use_saved_bond_order_factors is true, and saves the calculated values if it does not find them.

The slot index is the index of the atom in the interaction being evaluated (so for a triplet A-B-C, A would have slot 1, B slot 2, and C slot 3). This is only used for storing the values.

Parameters:

n_atoms: **integer intent(in) scalar** number of atoms

group_index: **integer intent(in) scalar** index for the bond order factor group

atom_index: **integer intent(in) scalar** index of the atom whose bond order factor is differentiated

slot_index: **integer intent(in) scalar** index denoting the position of the atom in an interacting group (such as A-B-C triplet)

bond_order_gradients: **double precision intent(out) size(3, n_atoms)** the calculated gradients of the bond order factor

bond_order_virial: **double precision intent(out) size(6)**

core_get_bond_order_sums (n_atoms, group_index, bond_order_sums)

Returns the bond order sums of all atoms for the given group. By ‘bond order sum’, we mean the summation of local terms without per atom scaling. E.g., for $b_i = 1 + \sum c_{ij}$, $\sum c_{ij}$ is the sum. The routine tries to find the values in the stored precalculated values first if use_saved_bond_order_factors is true, and saves the calculated values if it does not find them.

Parameters:

n_atoms: **integer intent(in) scalar** number of atoms

group_index: **integer intent(in) scalar** index for the bond order factor group

bond_order_sums: double precision **intent(out)** *size(n_atoms)* the calculated bond order sums

core_get_cell_vectors (*vectors*)
Returns the vectors defining the supercell stored in the core.
called from PyInterface: `get_cell_vectors()`
Parameters:

vectors: double precision **intent(out)** *size(3, 3)* A 3x3 matrix containing the vectors spanning the supercell. The first index runs over xyz and the second index runs over the three vectors.

core_get_ewald_energy (*real_cut, reciprocal_cut, sigma, epsilon, energy*)
Debug routine for Ewald
Parameters:

real_cut: double precision **intent(in)** *scalar*
reciprocal_cut: integer **intent(in)** *size(3)*
sigma: double precision **intent(in)** *scalar*
epsilon: double precision **intent(in)** *scalar*
energy: double precision **intent(out)** *scalar*

core_get_neighbor_list_of_atom (*atom_index, n_neighbors, neighbors, offsets*)
Parameters:

atom_index: integer **intent(in)** *scalar*
n_neighbors: integer **intent(in)** *scalar*
neighbors: integer **intent(out)** *size(n_neighbors)*
offsets: integer **intent(out)** *size(3, n_neighbors)*

core_get_number_of_atoms (*n_atoms*)
Returns the number of atoms in the array allocated in the core.
called from PyInterface: `get_number_of_atoms()`
Parameters:

n_atoms: integer **intent(out)** *scalar* number of atoms

core_get_number_of_neighbors (*atom_index, n_neighbors*)
Parameters:

atom_index: integer **intent(in)** *scalar*
n_neighbors: integer **intent(out)** *scalar*

core_loop_over_local_interactions (*n_atoms, calculation_type, total_energy, total_forces, totalenegs, total_stress*)
Loops over atoms, atomic pairs, atomic triplets, and atomic quadruplets and calculates the contributions from local potentials to energy, forces, or electronegativities. This routine is called from the routines

- `core_calculate_energy()`
- `core_calculate_forces()`
- `core_calculate_electronegativities()`

Parameters:

n_atoms: integer **intent(in)** *scalar* number of atoms

calculation_type: integer *intent(in) scalar* index to specify if the loop calculates energies, forces, or e-negativities

total_energy: double precision *intent(out) scalar* calculated energy

total_forces: double precision *intent(out) size(3, n_atoms)* calculated forces

total_enegs: double precision *intent(out) size(n_atoms)* calculated electronegativities

total_stress: double precision *intent(out) size(6)* calculated stress

core_post_process_bond_order_factors (*n_atoms*, *group_index*, *raw_sums*, *total_bond_orders*)

Bond-order post processing, i.e., application of per-atom scaling functions.

By post processing, we mean any operations done after calculating the sum of pair- and many-body terms. That is, if a factor is, say,

$$b_i = f\left(\sum_j c_{ij}\right) = 1 + \sum_j c_{ij},$$

the $\sum_j c_{ij}$ would have been calculated already (with `core_calculate_bond_order_factors()`) and the operation $f(x) = 1 + x$ remains to be carried out. The post processing is done per atom regardless of if the bond factor is of a pair or many body type.

Parameters:

n_atoms: integer *intent(in) scalar* number of atoms

group_index: integer *intent(in) scalar* an index denoting the potential to which the factor is connected

raw_sums: double precision *intent(in) size(n_atoms)* precalculated bond order sums, $\sum_j c_{ij}$, in the above example.

total_bond_orders: double precision *intent(out) size(n_atoms)* the calculated bond order factors b_i

core_post_process_bond_order_gradients (*n_atoms*, *group_index*, *raw_sums*, *raw_gradients*, *total_bond_gradients*, *mpi_split*)

Bond-order post processing, i.e., application of per-atom scaling functions. This routine does the scaling for all bond factors with the given bond order sums and gradients of these sums.

By post processing, we mean any operations done after calculating the sum of pair- and many-body terms. That is, if a factor is, say,

$$b_i = f\left(\sum_j c_{ij}\right) = 1 + \sum_j c_{ij},$$

the $\sum_j c_{ij}$ would have been calculated already and the operation $f(x) = 1 + x$ remains to be carried out. The post processing is done per atom regardless of if the bond factor is of a pair or many body type.

For gradients, one needs to evaluate

$$\nabla_\alpha b_i = f'\left(\sum_j c_{ij}\right) \nabla_\alpha \sum_j c_{ij}$$

Parameters:

n_atoms: integer *intent(in) scalar* number of atoms

group_index: integer *intent(in) scalar* an index denoting the potential to which the factor is connected

raw_sums: double precision *intent(in) size(n_atoms)* precalculated bond order sums, $\sum_j c_{ij}$, in the above example

raw_gradients: double precision *intent(in) size(3, n_atoms)* precalculated gradients of bond order sums, $\nabla_\alpha \sum_j c_{ij}$, in the above example

total_bond_gradients: double precision *intent(out) size(3, n_atoms)* the calculated bond order gradients $\nabla_\alpha b_i$

mpi_split: logical *intent(in) scalar optional* A switch for enabling MPI parallelization. By default the routine is sequential since the calculation may be called from within an already parallelized routine.

core_post_process_bond_order_gradients_of_factor (*n_atoms*, *group_index*,
atom_index, *raw_sum*,
raw_gradients, *total_bond_gradients*,
raw_virial, *total_virial*,
mpi_split)

Bond-order post processing, i.e., application of per-atom scaling functions. This routine does the scaling for the bond order factor of the given atom with respect to moving all atoms with the given bond order sum for the factor and the gradients of the sum with respect to moving all atoms.

By post processing, we mean any operations done after calculating the sum of pair- and many-body terms. That is, if a factor is, say,

$$b_i = f\left(\sum_j c_{ij}\right) = 1 + \sum_j c_{ij},$$

the $\sum_j c_{ij}$ would have been calculated already and the operation $f(x) = 1 + x$ remains to be carried out. The post processing is done per atom regardless of if the bond factor is of a pair or many body type.

For gradients, one needs to evaluate

$$\nabla_\alpha b_i = f'\left(\sum_j c_{ij}\right) \nabla_\alpha \sum_j c_{ij}$$

Parameters:

n_atoms: integer *intent(in) scalar* number of atoms

group_index: integer *intent(in) scalar* an index denoting the potential to which the factor is connected

atom_index: integer *intent(in) scalar* the index of the atom whose factor is differentiated (*i*)

raw_sum: double precision *intent(in) scalar* precalculated bond order sum for the given atom, $\sum_j c_{ij}$, in the above example

raw_gradients: double precision *intent(in) size(3, n_atoms)* precalculated gradients of bond order sums, $\nabla_\alpha \sum_j c_{ij}$, in the above example

total_bond_gradients: double precision *intent(out) size(3, n_atoms)* the calculated bond order gradients $\nabla_\alpha b_i$

raw_virial: double precision *intent(in) size(6)*

total_virial: double precision *intent(out) size(6)*

mpi_split: *logical intent(in) scalar optional* A switch for enabling MPI parallelization. By default the routine is sequential since the calculation may be called from within an already parallelized routine.

core_release_all_memory ()

Release all allocated pointer arrays in the core.

core_set_ewald_parameters (*n_atoms, real_cut, reciprocal_cut, sigma, epsilon, scaler*)

Sets the parameters for Ewald summation in the core.

Parameters:

n_atoms: integer *intent(in) scalar*

real_cut: *double precision intent(in) scalar* the real-space cutoff

reciprocal_cut: *integer intent(in) size(3)* the k-space cutoffs

sigma: *double precision intent(in) scalar* the split parameter

epsilon: *double precision intent(in) scalar* electric constant

scaler: *double precision intent(in) size(n_atoms)* scaling factors for the individual charges

core_update_atom_charges (*n_atoms, charges*)

Updates the charges of atomic particles.

called from PyInterface: `update_atom_charges()`

Parameters:

n_atoms: *integer intent(in) scalar* number of atoms

charges: *double precision intent(in) size(n_atoms)* new charges for the atoms

core_update_atom_coordinates (*n_atoms, positions, momenta*)

Updates the positions and momenta of atomic particles.

called from PyInterface: `update_atom_coordinates()`

Parameters:

n_atoms: *integer intent(in) scalar* number of atoms

positions: *double precision intent(in) size(3, n_atoms)* new coordinates for the atoms

momenta: *double precision intent(in) size(3, n_atoms)* new momenta for the atoms

expand_neighbor_storage (*nbors_and_offsets, length, new_length, n_atoms*)

Parameters:

nbors_and_offsets: integer *intent()* pointer *size(:, :, :)*

length: integer *intent(in) scalar*

new_length: integer *intent(in) scalar*

n_atoms: integer *intent(in) scalar*

list_atoms ()

Prints some information on the atoms stored in the core in stdout.

list_bonds ()

Prints some information on the bond order factors stored in the core in stdout.

list_cell ()

Prints some information on the supercell stored in the core in stdout.

```
list_interactions()
```

Prints some information on the potentials stored in the core in stdout.

potentials (Potentials.f90)

Potentials contains the low-level routines for handling interactions. The module defines custom types for both describing the types of potentials and bond order factors (`potential_descriptor`, `bond_order_descriptor`) as well as for storing the parameters of actual interactions in use for the Fortran calculations (`potential`, `bond_order_parameters`). Tools for creating the custom datatypes (`create_potential()`, `create_bond_order_factor()`) are provided.

The types of potentials and bond order factors are defined using the types `potential_descriptor` and `bond_order_descriptor`. These should be created at start-up and remain untouched during simulation. They are used by the Fortran core for checking the types of parameters a potential needs, for instance, but they are also accessible from the Python interface. Especially, upon creation of `Potential` and `BondOrderParameters` instances, one needs to specify the type as a keyword. This keyword is then compared to the list of characterizers in the core to determine the type of the interaction.

The basic routines for calculating the actual forces and energies are also defined in this module (`evaluate_energy()`, `evaluate_forces()`, `evaluate_bond_order_factor()`, `evaluate_bond_order_gradient()`). However, these routines do not calculate the total potential energy of the system, V , or the total forces acting on the particles, $\mathbf{F} = -\nabla_{\alpha} V$. Instead, the routines evaluate the contributions from individual atoms, atom pairs, atom triplets, etc. For instance, let the total energy of the system be

$$V = \sum_p \left(\sum_i v_i^p + \sum_{(i,j)} v_{ij}^p + \sum_{(i,j,k)} v_{ijk}^p \right),$$

where p sums over the different potentials acting on the system and i , (i, j) and (i, j, k) sum over all atoms, pairs and triplet, respectively. Then the energy terms v are obtained from `evaluate_energy()`. In pseudo-code,

$$v_S^p = \text{evaluate_energy}(S, p),$$

where S is a set of atoms. The summation over potentials and atoms is done in `pysic_core (Core.f90)` in `calculate_energy()`. Similarly for forces, the summation is carried out in `calculate_forces()`.

The reason for separating the calculation of individual interaction terms to `potentials (Potentials.f90)` and the overall summation to `pysic_core (Core.f90)` is that only the core knows the current structure and interactions of the system. It is the task of this module to tell the core how all the potentials behave given any local structure, but the overall system information is kept in the core. So during energy evaluation, `pysic_core (Core.f90)` finds all local structures that possibly contribute with an interaction and asks `potentials (Potentials.f90)` to calculate this contribution.

Bond order factors are potential modifiers, not direct interactions themselves. In general, the factors are scalar functions defined per atom, for instance,

$$b_i^p = s_i^p \left(\sum_{(i,j)} c_{ij}^p + \sum_{(i,j,k)} c_{ijk}^p \right)$$

for a three-body factor, where c^p are local contributions (usually representing chemical bonds) and s_i^p is a per atom scaling function. The bond factors multiply the potentials p leading to the total energy

$$V = \sum_p \left(\sum_i b_i^p v_i^p + \sum_{(i,j)} \frac{1}{2} (b_i^p + b_j^p) v_{ij}^p + \sum_{(i,j,k)} \frac{1}{3} (b_i^p + b_j^p + b_k^p) v_{ijk}^p \right).$$

The corresponding total force on atom α is then

$$\mathbf{F}_\alpha = -\nabla_\alpha V = -\sum_p \left(\sum_i ((\nabla_\alpha b_i^p) v_i^p + b_i^p (\nabla_\alpha v_i^p)) + \dots \right).$$

The contributions $\mathbf{f}_\alpha^p = -\nabla_\alpha v^p$, c^p , and $\nabla_\alpha c^p$ are calculated in `evaluate_forces()`, `evaluate_bond_order_factor()`, and `evaluate_bond_order_gradient()`. Application of the scaling functions s_i and s'_i on the sums $\sum_{(i,j)} c_{ij}^p + \sum_{(i,j,k)} c_{ijk}^p$ is done in the routines `post_process_bond_order_factor()` and `post_process_bond_order_gradient()` to produce the actual bond order factors b_i^p and gradients $\nabla_\alpha b_i^p$. These sums, similarly to the energy and force summations, are evaluated with `core_calculate_bond_order_factors()` in *pysic_core (Core.f90)*.

Note when adding potentials or bond order factors in the source code:

The parameters defined in `Potentials.f90` are used for determining the maximum sizes of arrays, numbers of potentials and bond factors, and the internally used indices for them. When adding new potentials or bond factors, make sure to update the relevant numbers. Especially the number of potentials (`n_potential_types`) or number of bond order factors (`n_bond_order_types`) must be increased when more types are defined.

Also note that in *pysic_interface (PyInterface.f90)*, some of these parameters are used for determining array sizes. However, the actual parameters are not used because `f2py` does not read the values from here. Therefore if you change a parameter here, search for its name in *pysic_interface (PyInterface.f90)* to see if the name appears in a comment. That is an indicator that a numeric value must be updated accordingly.

Full documentation of global variables in potentials

bond_descriptors_created

logical scalar

initial value = .false.

logical tag used for managing pointer allocations for bond order factor descriptors

bond_order_descriptors

type(bond_order_descriptor) pointer size(:)

an array for storing descriptors for the different *types* of bond order factors

c_scale_index

integer scalar parameter

initial value = 3

internal index for the coordination scaling potential

coordination_index

integer scalar parameter

initial value = 1

descriptors_created

logical scalar

initial value = .false.

logical tag used for managing pointer allocations for potential descriptors

mono_const_index

integer scalar parameter

initial value = 3

internal index for the constant force potential

mono_none_index

integer *scalar parameter*

initial value = 6

internal index for the constant potential

n_bond_order_types

integer *scalar parameter*

initial value = 6

number of different types of bond order factors known

n_max_params

integer *scalar parameter*

initial value = 12

n_potential_types

integer *scalar parameter*

initial value = 9

number of different types of potentials known

no_name

character(len=label_length) *scalar parameter*

initial value = "xx"

The label for unlabeled atoms. In other words, there are routines that expect atomic symbols as arguments, but if there are no symbols to pass, this should be given to mark an empty entry.

pair_buck_index

integer *scalar parameter*

initial value = 7

internal index for the Buckingham potential

pair_exp_index

integer *scalar parameter*

initial value = 5

pair_lj_index

integer *scalar parameter*

initial value = 1

internal index for the Lennard-Jones potential

pair_power_index

integer *scalar parameter*

initial value = 9

internal index for the power law potential

pair_spring_index

integer *scalar parameter*

initial value = 2

internal index for the spring potential

param_name_length

integer scalar parameter

initial value = 10

param_note_length

integer scalar parameter

initial value = 100

maximum length allowed for the descriptions of parameters

pot_name_length

integer scalar parameter

initial value = 11

maximum length allowed for the names of potentials

pot_note_length

integer scalar parameter

initial value = 500

maximum length allowed for the description of the potential

potential_descriptors

type(potential_descriptor) pointer size(:)

an array for storing descriptors for the different *types* of potentials

power_index

integer scalar parameter

initial value = 5

internal index for the power law bond order factor

quad_dihedral_index

integer scalar parameter

initial value = 8

internal index for the dihedral angle potential

sqrt_scale_index

integer scalar parameter

initial value = 6

internal index for the square root scaling potential

tersoff_index

integer scalar parameter

initial value = 2

internal index for the Tersoff bond order factor

tri_bend_index

integer scalar parameter

initial value = 4

internal index for the bond bending potential

triplet_index*integer scalar parameter**initial value = 4*

internal index for the triplet bond order factor

Full documentation of custom types in potentials**bond_order_descriptor**

Description of a type of a potential. The type contains the name and description of the potential and the parameters it contains. The descriptors contain the information that the inquiry methods in the python interface fetch.

Contained data:

parameter_notes: *character(len=param_note_length) pointer size(:, :)* Descriptions of the parameters. The descriptions should be very short indicators such as ‘spring constant’ or ‘energy coefficient’. For more detailed explanations, the proper documentation should be used.

n_parameters: *integer pointer size(:)* number of parameters for each number of bodies (1-body parameters, 2-body parameters etc.)

includes_post_processing: *logical scalar* a logical tag specifying if there is a scaling function s_i attached to the factor.

description: *character(len=pot_note_length) scalar* A description of the bond order factor. This should contain the mathematical formulation as well as a short verbal explanation.

type_index: *integer scalar* The internal index of the bond order factor. This can also be used for recognizing the factor and must therefore match the name. For instance, if name = ‘neighbors’, type_index = `coordination_index`.

n_targets: *integer scalar* number of targets, i.e., interacting bodies

parameter_names: *character(len=param_name_length) pointer size(:, :)* The names of the parameters of the bond order factor: these are keywords according to which the parameters may be recognized.

name: *character(len=pot_name_length) scalar* The name of the bond order factor: this is a keyword according to which the factor may be recognized.

bond_order_parameters

Defines a particular bond order factor. The factor should correspond to the description of some built-in type and hold actual numeric values for parameters. In addition a real bond order factor must have information on the particles it acts on and the range it operates in. These are created based on the `BondOrderParameters` objects in the Python interface when calculations are invoked.

Contained data:

cutoff: *double precision scalar* The hard cutoff for the bond order factor. If the atoms are farther away from each other than this, they do not contribute to the total bond order factor does not affect them.

includes_post_processing: *logical scalar* a logical switch specifying if there is a scaling function s_i attached to the factor

group_index: *integer scalar* The internal index of the *potential* the bond order factor is modifying.

parameters: *double precision pointer size(:, :)* numerical values for parameters

soft_cutoff: *double precision scalar* The soft cutoff for the bond order factor. If this is smaller than the hard cutoff, the bond contribution is scaled to zero continuously when the interatomic distances grow from the soft to the hard cutoff.

type_index: *integer scalar* The internal index of the bond order factor *type*. This is used for recognizing the factor. Note that the bond order parameters instance does not have a name. If the name is needed, it can be obtained from the `bond_order_descriptor` of the correct index.

n_params: *integer pointer size(:)* array containing the numbers of parameters for different number of targets (1-body parameters, 2-body parameters, etc.)

original_elements: *character(len=2) pointer size(:)* The list of elements (atomic symbols) of the original `BondOrderParameters` in the Python interface from which this factor was created. Whereas the `apply_elements` lists are used for finding all pairs and triplets of atoms which could contribute to the bond order factor, the `original_elements` lists specify the roles of atoms in the factor.

derived_parameters: *double precision pointer size(:, :)* numerical values for parameters calculated based on the parameters specified by the user

apply_elements: *character(len=2) pointer size(:)* A list of elements (atomic symbols) the factor affects. E.g., for Si-O bonds, it would be ('Si','O'). Note that unlike in the Python interface, a single `bond_order_parameters` only has one set of targets, and for multiple target options several `bond_order_parameters` instances are created.

potential

Defines a particular potential. The potential should correspond to the description of some built-in type and hold actual numeric values for parameters. In addition, a real potential must have information on the particles it acts on and the range it operates in. These are to be created based on the `Potential` objects in the Python interface when calculations are invoked.

Contained data:

pot_index: *integer scalar* The internal index of the *actual potential*. This is needed when bond order factors are included so that the factors may be joint with the correct potentials.

smoothened: *logical scalar* logical switch specifying if a smooth cutoff is applied to the potential

filter_elements: *logical scalar* a logical switch specifying whether the potential targets atoms based on the atomic symbols

parameters: *double precision pointer size(:)* numerical values for parameters

cutoff: *double precision scalar* The hard cutoff for the potential. If the atoms are farther away from each other than this, the potential does not affect them.

soft_cutoff: *double precision scalar* The soft cutoff for the potential. If this is smaller than the hard cutoff, the potential is scaled to zero continuously when the interatomic distances grow from the soft to the hard cutoff.

apply_tags: *integer pointer size(:)* A list of atom tags the potential affects. Note that unlike in the Python interface, a single `potential` only has one set of targets, and for multiple target options several `potential` instances are created.

original_indices: *integer pointer size(:)* The list of atom indices of the original `Potential` in the Python interface from which this potential was created. Whereas the `apply_*` lists are used for finding all pairs and triplets of atoms for which the potential could act on, the `original_*` lists specify the roles of atoms in the interaction.

apply_indices: *integer pointer size(:)* A list of atom indices the potential affects. Note that unlike in the Python interface, a single `potential` only has one set of targets, and for multiple target options several `potential` instances are created.

filter_indices: *logical scalar* a logical switch specifying whether the potential targets atoms based on the atom indices

type_index: *integer scalar* The internal index of the potential *type*. This is used for recognizing the potential. Note that the potential instance does not have a name. If the name is needed, it can be obtained from the `potential_descriptor` of the correct index.

original_tags: *integer pointer size(:)* The list of atom tags of the original `Potential` in the Python interface from which this potential was created. Whereas the `apply_*` lists are used for finding all pairs and triplets of atoms for which the potential could act on, the `original_*` lists specify the roles of atoms in the interaction.

original_elements: *character(len=2) pointer size(:)* The list of elements (atomic symbols) of the original `Potential` in the Python interface from which this potential was created. Whereas the `apply_*` lists are used for finding all pairs and triplets of atoms for which the potential could act on, the `original_*` lists specify the roles of atoms in the interaction.

derived_parameters: *double precision pointer size(:)* numerical values for parameters calculated based on the parameters specified by the user

apply_elements: *character(len=2) pointer size(:)* A list of elements (atomic symbols) the potential affects. E.g., for a Si-O potential, it would be ('Si','O'). Note that unlike in the Python interface, a single `potential` only has one set of targets, and for multiple target options several `potential` instances are created.

filter_tags: *logical scalar* a logical switch specifying whether the potential targets atoms based on the atom tags

potential_descriptor

Description of a type of a potential. The type contains the name and description of the potential and the parameters it contains. The descriptors contain the information that the inquiry methods in the python interface fetch.

Contained data:

parameter_notes: *character(len=param_note_length) pointer size(:)* Descriptions of the parameters. The descriptions should be very short indicators such as 'spring constant' or 'energy coefficient'. For more detailed explanations, the proper documentation should be used.

n_parameters: *integer scalar* number of parameters

description: *character(len=pot_note_length) scalar* A description of the potential. This should contain the mathematical formulation as well as a short verbal explanation.

type_index: *integer scalar* The internal index of the potential. This can also be used for recognizing the potential and must therefore match the name. For instance, if name = 'LJ', `type_index = pair_lj_index`.

n_targets: *integer scalar* number of targets, i.e., interacting bodies

parameter_names: *character(len=param_name_length) pointer size(:)* The names of the parameters of the potential: these are keywords according to which the parameters may be recognized.

name: *character(len=pot_name_length) scalar* The name of the potential: this is a keyword according to which the potentials may be recognized.

Full documentation of subroutines in potentials

bond_order_factor_affects_atom (*factor, atom_in, affects, position*)

Tests whether the given bond order factor affects the specific atom.

For bond order factors, the atoms are specified as valid targets by the atomic symbol only.

If position is not given, then the routine returns true if the atom can appear in the bond order factor in any role. If position is given, then true is returned only if the atom is valid for that particular position.

For instance, we may want to calculate the coordination of Cu-O bonds for Cu but not for O.

Parameters:

factor: **type(bond_order_parameters) intent(in) scalar** the `bond_order_parameters`

atom_in: **type(atom) intent(in) scalar** the `atom`

affects: **logical intent(out) scalar** true if the bond order factor is affected by the atom

position: **integer intent(in) scalar optional** specifies the particular role of the atom in the bond order factor

bond_order_factor_is_in_group (*factor, group_index, in_group*)

Tests whether the given bond order factor is a member of a specific group, i.e., if it affects the potential specified by the group index.

Parameters:

factor: **type(bond_order_parameters) intent(in) scalar** the `bond_order_parameters`

group_index: **integer intent(in) scalar** the index for the potential

in_group: **logical intent(out) scalar** true if the factor is a member of the group

calculate_derived_parameters_bond_bending (*n_params, parameters, new_potential*)

Bond bending derived parameters

Parameters:

n_params: **integer intent(in) scalar**

parameters: **double precision intent(in) size(n_params)**

new_potential: **type(potential) intent(inout) scalar** the potential object for which the parameters are calculated

calculate_derived_parameters_charge_exp (*n_params, parameters, new_potential*)

Charge exponential derived parameters

Parameters:

n_params: **integer intent(in) scalar**

parameters: **double precision intent(in) size(n_params)**

new_potential: **type(potential) intent(inout) scalar** the potential object for which the parameters are calculated

calculate_derived_parameters_dihedral (*n_params, parameters, new_potential*)

Dihedral angle derived parameters

Parameters:

n_params: **integer intent(in) scalar**

parameters: double precision *intent(in) size(n_params)*

new_potential: **type(potential) intent(inout) scalar** the potential object for which the parameters are calculated

calculate_ewald_electronegativities (*n_atoms*, *atoms*, *cell*, *real_cutoff*,
reciprocal_cutoff, *gaussian_width*,
electric_constant, *filter*, *scaler*, *include_dipole_correction*, *totalenegs*)

Calculates the electronegativities due to long ranged $\frac{1}{r}$ potentials. These electronegativities are the derivatives of the energies U given by `calculate_ewald_energy()`

$$\chi_{\alpha} = -\frac{\partial U}{\partial q_{\alpha}}$$

Parameters:

n_atoms: **integer intent(in) scalar** number of atoms

atoms: **type(atom) intent(in) size(n_atoms)** list of atoms

cell: **type(supercell) intent(in) scalar** the supercell containing the system

real_cutoff: **double precision intent(in) scalar** Cutoff radius of real-space interactions. Note that the neighbor lists stored in the atoms are used for neighbor finding so the cutoff cannot exceed the cutoff for the neighbor lists. (Or, it can, but the neighbors not in the lists will not be found.)

reciprocal_cutoff: **integer intent(in) size(3)** The number of cells to be included in the reciprocal sum in the directions of the reciprocal cell vectors. For example, if `reciprocal_cutoff = [3, 4, 5]`, the reciprocal sum will be truncated as $\sum_{\mathbf{k} \neq 0} = \sum_{k_1=-3}^3 \sum_{k_2=-4}^4 \sum_{k_3=-5}^5, (k_1, k_2, k_3) \neq (0, 0, 0)$.

gaussian_width: **double precision intent(in) scalar** The σ parameter, i.e., the distribution width of the screening Gaussians. This should not influence the actual value of the energy, but it does influence the convergence of the summation. If σ is large, the real space sum E_s converges slowly and a large real space cutoff is needed. If it is small, the reciprocal term E_l converges slowly and the sum over the reciprocal lattice has to be evaluated over several cell lengths.

electric_constant: **double precision intent(in) scalar** The electric constant, i.e., vacuum permittivity ϵ_0 . In atomic units, it is $\epsilon_0 = 0.00552635 \frac{e^2}{\text{eV}}$, but if one wishes to scale the results to some other unit system (such as reduced units with $\epsilon_0 = 1$), that is possible as well.

filter: **logical intent(in) size(n_atoms)** a list of logical values, one per atom, false for the atoms that should be ignored in the calculation

scaler: **double precision intent(in) size(n_atoms)** a list of numerical values to scale the individual charges of the atoms

include_dipole_correction: **logical intent(in) scalar** if true, a dipole correction term is included

totalenegs: **double precision intent(out) size(n_atoms)** the calculated electronegativities

calculate_ewald_energy (*n_atoms*, *atoms*, *cell*, *real_cutoff*, *reciprocal_cutoff*,
gaussian_width, *electric_constant*, *filter*, *scaler*, *include_dipole_correction*, *total_energy*)

Calculates the energy of $\frac{1}{r}$ potentials through Ewald summation.

If a periodic system contains charges interacting via the $\frac{1}{r}$ Coulomb potential, direct summation of the interactions

$$E = \sum_{(i,j)} \frac{1}{4\pi\epsilon_0} \frac{q_i q_j}{r_{ij}}, \quad (4.17)$$

where the sum is over pairs of charges q_i, q_j (charges of the entire system, not just the simulation cell) and the distance between the charges is $r_{ij} = |\mathbf{r}_j - \mathbf{r}_i|$, does not work in general because the sum (4.17) converges very slowly.³ Therefore truncating the sum may lead to severe errors.

The standard technique for overcoming this problem is the so called Ewald summation method. The idea is to split the long ranged and singular Coulomb potential to a short ranged singular and long ranged smooth parts, and calculate the long ranged part in reciprocal space via Fourier transformations. This is possible since the system is periodic and the same supercell repeats infinitely in all directions. In practice the calculation can be done by adding (and subtracting) Gaussian charge densities over the point charges to screen the potential in real space. That is, the original charge density $\rho(\mathbf{r}) = \sum_i q_i \delta(\mathbf{r} - \mathbf{r}_i)$ is split by

$$\rho(\mathbf{r}) = \rho_s(\mathbf{r}) + \rho_l(\mathbf{r}) \quad (4.18)$$

$$\rho_s(\mathbf{r}) = \sum_i [q_i \delta(\mathbf{r} - \mathbf{r}_i) - q_i G_\sigma(\mathbf{r} - \mathbf{r}_i)] \quad (4.19)$$

$$\rho_l(\mathbf{r}) = \sum_i q_i G_\sigma(\mathbf{r} - \mathbf{r}_i) \quad (4.20)$$

$$G_\sigma(\mathbf{r}) = \frac{1}{(2\pi\sigma^2)^{3/2}} \exp\left(-\frac{|\mathbf{r}|^2}{2\sigma^2}\right) \quad (4.21)$$

Here ρ_l generates a long range interaction since at large distances the Gaussian densities G_σ appear the same as point charges ($\lim_{\sigma/r \rightarrow 0} G_\sigma(\mathbf{r}) = \delta(\mathbf{r})$). Since the charge density is smooth, so will be the potential it creates. The density ρ_s exhibits short ranged interactions for the same reason: At distances longer than the width of the Gaussians the point charges are screened by the Gaussians which exactly cancel them ($\lim_{\sigma/r \rightarrow 0} \delta(\mathbf{r}) - G_\sigma(\mathbf{r}) = 0$).

The short ranged interactions are directly calculated in real space

$$E_s = \frac{1}{4\pi\epsilon_0} \int \frac{\rho_s(\mathbf{r})\rho_s(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} d^3r d^3r' \quad (4.22)$$

$$= \frac{1}{4\pi\epsilon_0} \sum_{(i,j)} \frac{q_i q_j}{r_{ij}} \operatorname{erfc}\left(\frac{r_{ij}}{\sigma\sqrt{2}}\right). \quad (4.23)$$

The complementary error function $\operatorname{erfc}(r) = 1 - \operatorname{erf}(r) = 1 - \frac{2}{\sqrt{\pi}} \int_0^r e^{-t^2/2} dt$ makes the sum converge rapidly as $\frac{r_{ij}}{\sigma} \rightarrow \infty$.

The long ranged interaction can be calculated in reciprocal space by Fourier transformation. The result is

$$E_l = \frac{1}{2V\epsilon_0} \sum_{\mathbf{k} \neq 0} \frac{e^{-\sigma^2 k^2/2}}{k^2} |S(\mathbf{k})|^2 - \frac{1}{4\pi\epsilon_0} \frac{1}{\sqrt{2\pi}\sigma} \sum_i q_i^2 \quad (4.24)$$

$$S(\mathbf{k}) = \sum_i^N q_i e^{i\mathbf{k} \cdot \mathbf{r}_i} \quad (4.25)$$

The first sum in E_l runs over the reciprocal lattice $\mathbf{k} = k_1 \mathbf{b}_1 + k_2 \mathbf{b}_2 + k_3 \mathbf{b}_3$ where \mathbf{b}_i are the vectors spanning the reciprocal cell ($[\mathbf{b}_1 \mathbf{b}_2 \mathbf{b}_3] = ([\mathbf{v}_1 \mathbf{v}_2 \mathbf{v}_3]^{-1})^T$ where \mathbf{v}_i are the real space cell vectors). The latter sum is the self energy of each point charge in the potential of the particular Gaussian that screens the charge, and the sum runs over all charges in the supercell spanning the periodic system. (The self energy must be removed because it is present in the first sum even though when evaluating the potential at the position of a charge due to the other charges, no screening Gaussian function should be placed over the charge itself.) Likewise the sum in the structure factor $S(\mathbf{k})$ runs over all charges in the supercell.

³ In fact, the sum converges only conditionally meaning the result depends on the order of summation. Physically this is not a problem, because one never has infinite lattices.

The total energy is then the sum of the short and long range energies

$$E = E_s + E_l.$$

Parameters:

n_atoms: integer *intent(in)* scalar number of atoms

atoms: type(atom) *intent(in)* size(n_atoms) list of atoms

cell: type(supercell) *intent(in)* scalar the supercell containing the system

real_cutoff: double precision *intent(in)* scalar Cutoff radius of real-space interactions. Note that the neighbor lists stored in the atoms are used for neighbor finding so the cutoff cannot exceed the cutoff for the neighbor lists. (Or, it can, but the neighbors not in the lists will not be found.)

reciprocal_cutoff: integer *intent(in)* size(3) The number of cells to be included in the reciprocal sum in the directions of the reciprocal cell vectors. For example, if `reciprocal_cutoff = [3, 4, 5]`, the reciprocal sum will be truncated as $\sum_{\mathbf{k} \neq 0} = \sum_{k_1=-3}^3 \sum_{k_2=-4}^4 \sum_{k_3=-5}^5, (k_1, k_2, k_3) \neq (0, 0, 0)$.

gaussian_width: double precision *intent(in)* scalar The σ parameter, i.e., the distribution width of the screening Gaussians. This should not influence the actual value of the energy, but it does influence the convergence of the summation. If σ is large, the real space sum E_s converges slowly and a large real space cutoff is needed. If it is small, the reciprocal term E_l converges slowly and the sum over the reciprocal lattice has to be evaluated over several cell lengths.

electric_constant: double precision *intent(in)* scalar The electric constant, i.e., vacuum permittivity ϵ_0 . In atomic units, it is $\epsilon_0 = 0.00552635 \frac{e^2}{\text{eV}}$, but if one wishes to scale the results to some other unit system (such as reduced units with $\epsilon_0 = 1$), that is possible as well.

filter: logical *intent(in)* size(n_atoms) a list of logical values, one per atom, false for the atoms that should be ignored in the calculation

scaler: double precision *intent(in)* size(n_atoms) a list of numerical values to scale the individual charges of the atoms

include_dipole_correction: logical *intent(in)* scalar if true, a dipole correction term is included in the energy

total_energy: double precision *intent(out)* scalar the calculated energy

calculate_ewald_forces (n_atoms, atoms, cell, real_cutoff, reciprocal_cutoff, gaussian_width, electric_constant, filter, scaler, include_dipole_correction, total_forces, total_stress)

Calculates the forces due to long ranged $\frac{1}{r}$ potentials. These forces are the gradients of the energies U given by `calculate_ewald_energy()`

$$\mathbf{F}_\alpha = -\nabla_\alpha U$$

Parameters:

n_atoms: integer *intent(in)* scalar number of atoms

atoms: type(atom) *intent(in)* size(n_atoms) list of atoms

cell: type(supercell) *intent(in)* scalar the supercell containing the system

real_cutoff: double precision *intent(in)* scalar Cutoff radius of real-space interactions. Note that the neighbor lists stored in the atoms are used for neighbor finding so the cutoff cannot exceed the cutoff for the neighbor lists. (Or, it can, but the neighbors not in the lists will not be found.)

reciprocal_cutoff: integer *intent(in) size(3)* The number of cells to be included in the reciprocal sum in the directions of the reciprocal cell vectors. For example, if `reciprocal_cutoff = [3, 4, 5]`, the reciprocal sum will be truncated as $\sum_{\mathbf{k} \neq 0} = \sum_{k_1=-3}^3 \sum_{k_2=-4}^4 \sum_{k_3=-5}^5, (k_1, k_2, k_3) \neq (0, 0, 0)$.

gaussian_width: double precision *intent(in) scalar* The σ parameter, i.e., the distribution width of the screening Gaussians. This should not influence the actual value of the energy, but it does influence the convergence of the summation. If σ is large, the real space sum E_s converges slowly and a large real space cutoff is needed. If it is small, the reciprocal term E_l converges slowly and the sum over the reciprocal lattice has to be evaluated over several cell lengths.

electric_constant: double precision *intent(in) scalar* The electric constant, i.e., vacuum permittivity ϵ_0 . In atomic units, it is $\epsilon_0 = 0.00552635 \frac{e^2}{\text{eV}}$, but if one wishes to scale the results to some other unit system (such as reduced units with $\epsilon_0 = 1$), that is possible as well.

filter: logical *intent(in) size(n_atoms)* a list of logical values, one per atom, false for the atoms that should be ignored in the calculation

scaler: double precision *intent(in) size(n_atoms)* a list of numerical values to scale the individual charges of the atoms

include_dipole_correction: logical *intent(in) scalar* if true, a dipole correction term is included in the energy

total_forces: double precision *intent(out) size(3, n_atoms)* the calculated forces

total_stress: double precision *intent(out) size(6)* the calculated stress

clear_bond_order_factor_characterizers ()

Deallocates all memory associated with bond order factor characterizes.

clear_potential_characterizers ()

Deallocates all memory associated with potential characterizes.

create_bond_order_factor (*n_targets, n_params, n_split, bond_name, parameters, param_split, cutoff, soft_cutoff, elements, orig_elements, group_index, new_bond*)

Returns a `bond_order_parameters`.

The routine takes as arguments all the necessary parameters and returns a bond order parameters type wrapping them in one package.

Parameters:

n_targets: integer *intent(in) scalar* number of targets, i.e., interacting bodies

n_params: integer *intent(in) scalar* array containing the numbers of parameters for different number of targets (1-body parameters, 2-body parameters, etc.)

n_split: integer *intent(in) scalar* number of groupings in the list of parameters, per number of bodies - should equal `n_targets`

bond_name: character(len=*) *intent(in) scalar* name of the bond order factor - a keyword that must match a name of one of the `bond_order_descriptors`

parameters: double precision *intent(in) size(n_params)* numerical values for parameters as a one-dimensional array

param_split: integer *intent(in) size(n_split)* Array containing the numbers of 1-body, 2-body, etc. parameters. The parameters are given as a list, but a bond order factor may have parameters separately for different numbers of targets. This list specifies the number of parameters for each.

cutoff: double precision *intent(in) scalar* The hard cutoff for the bond order factor. If the atoms are farther away from each other than this, they do not contribute to the total bond order factor does not affect them.

soft_cutoff: double precision *intent(in) scalar* The soft cutoff for the bond order factor. If this is smaller than the hard cutoff, the bond contribution is scaled to zero continuously when the interatomic distances grow from the soft to the hard cutoff.

elements: character(len=2) *intent(in) size(n_targets)* a list of elements (atomic symbols) the factor affects

orig_elements: character(len=2) *intent(in) size(n_targets)* the list of elements (atomic symbols) of the original `BondOrderParameters` in the Python interface from which this factor was created

group_index: integer *intent(in) scalar* The internal index of the *potential* the bond order factor is modifying.

new_bond: type(bond_order_parameters) *intent(out) scalar* the created
`bond_order_parameters`

create_bond_order_factor_characterizer_coordination (*index*)

Coordination characterizer initialization

Parameters:

index: integer *intent(in) scalar* index of the bond order factor

create_bond_order_factor_characterizer_power (*index*)

Power decay characterizer initialization

Parameters:

index: integer *intent(in) scalar* index of the bond order factor

create_bond_order_factor_characterizer_scaler_1 (*index*)

Scaler characterizer initialization

Parameters:

index: integer *intent(in) scalar* index of the bond order factor

create_bond_order_factor_characterizer_scaler_sqrt (*index*)

Square root scaler characterizer initialization

Parameters:

index: integer *intent(in) scalar* index of the bond order factor

create_bond_order_factor_characterizer_tersoff (*index*)

Tersoff characterizer initialization

Parameters:

index: integer *intent(in) scalar* index of the bond order factor

create_bond_order_factor_characterizer_triplet (*index*)

Triplet characterizer initialization

Parameters:

index: integer *intent(in) scalar* index of the bond order factor

create_potential (*n_targets, n_params, pot_name, parameters, cutoff, soft_cutoff, elements, tags, indices, orig_elements, orig_tags, orig_indices, pot_index, new_potential*)

Returns a `potential`.

The routine takes as arguments all the necessary parameters and returns a potential type wrapping them in one package.

Parameters:

n_targets: *integer intent(in) scalar* number of targets, i.e., interacting bodies

n_params: *integer intent(in) scalar* number of parameters

pot_name: *character(len=*) intent(in) scalar* name of the potential - a keyword that must match a name of one of the `potential_descriptors`

parameters: *double precision intent(in) size(n_params)* array of numerical values for the parameters

cutoff: *double precision intent(in) scalar* the hard cutoff for the potential

soft_cutoff: *double precision intent(in) scalar* the soft cutoff for the potential

elements: *character(len=2) intent(in) size(n_targets)* the elements (atomic symbols) the potential acts on

tags: *integer intent(in) size(n_targets)* the atom tags the potential acts on

indices: *integer intent(in) size(n_targets)* the atom indices the potential acts on

orig_elements: *character(len=2) intent(in) size(n_targets)* The elements (atomic symbols) in the `Potential` used for generating the potential. This is needed to specify the roles of the atoms in the interaction.

orig_tags: *integer intent(in) size(n_targets)* The atom tags in the `Potential` used for generating the potential. This is needed to specify the roles of the atoms in the interaction.

orig_indices: *integer intent(in) size(n_targets)* The atom indices in the `Potential` used for generating the potential. This is needed to specify the roles of the atoms in the interaction.

pot_index: *integer intent(in) scalar* the internal index of the potential

new_potential: *type(potential) intent(out) scalar* the created `potential`

create_potential_characterizer_LJ (*index*)

LJ characterizer initialization

Parameters:

index: *integer intent(in) scalar* index of the potential

create_potential_characterizer_bond_bending (*index*)

bond-bending characterizer initialization

Parameters:

index: *integer intent(in) scalar* index of the potential

create_potential_characterizer_buckingham (*index*)

Buckingham characterizer initialization

Parameters:

index: *integer intent(in) scalar* index of the potential

create_potential_characterizer_charge_exp (*index*)

charge exponential characterizer initialization

Parameters:

index: integer *intent(in) scalar* index of the potential

create_potential_characterizer_constant_force (*index*)

constant F characterizer initialization

Parameters:

index: integer *intent(in) scalar* index of the potential

create_potential_characterizer_constant_potential (*index*)

constant potential characterizer initialization

Parameters:

index: integer *intent(in) scalar* index of the potential

create_potential_characterizer_dihedral (*index*)

dihedral angle characterizer initialization

Parameters:

index: integer *intent(in) scalar* index of the potential

create_potential_characterizer_power (*index*)

Power law characterizer initialization

Parameters:

index: integer *intent(in) scalar* index of the potential

create_potential_characterizer_spring (*index*)

spring characterizer initialization

Parameters:

index: integer *intent(in) scalar* index of the potential

evaluate_bond_order_factor (*n_targets*, *separations*, *distances*, *bond_params*, *factor*,
atoms)

Returns a bond order factor term.

By a bond order factor term, we mean the contribution from specific atoms, c_{ijk} , appearing in the factor

$$b_i = f\left(\sum_{jk} c_{ijk}\right)$$

This routine evaluates the term c_{ij} or c_{ijk} for the given atoms ij or ijk according to the given parameters.

Parameters:

n_targets: integer *intent(in) scalar* number of targets

separations: double precision *intent(in) size(3, n_targets-1)* atom-atom separation vectors \mathbf{r}_{12} , \mathbf{r}_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(n_targets-1)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

bond_params: type(bond_order_parameters) *intent(in) size(n_targets-1)* a
`bond_order_parameters` containing the parameters

factor: double precision *intent(out) size(n_targets)* the calculated bond order term c

atoms: *type(atom) intent(in) size(n_targets)* a list of the actual `atom` objects for which the term is calculated

evaluate_bond_order_factor_coordination (*separations, distances, bond_params, factor*)

Coordination bond order factor

Parameters:

separations: double precision *intent(in) size(3, 1)* atom-atom separation vectors \mathbf{r}_{12} , \mathbf{r}_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(1)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

bond_params: *type(bond_order_parameters) intent(in) size(1)* a `bond_order_parameters` containing the parameters

factor: double precision *intent(out) size(2)* the calculated bond order term c

evaluate_bond_order_factor_power (*separations, distances, bond_params, factor*)

Power bond order factor

Parameters:

separations: double precision *intent(in) size(3, 1)* atom-atom separation vectors \mathbf{r}_{12} , \mathbf{r}_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(1)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

bond_params: *type(bond_order_parameters) intent(in) size(1)* a `bond_order_parameters` containing the parameters

factor: double precision *intent(out) size(2)* the calculated bond order term c

evaluate_bond_order_factor_tersoff (*separations, distances, bond_params, factor, atoms*)

Parameters:

separations: double precision *intent(in) size(3, 2)* atom-atom separation vectors \mathbf{r}_{12} , \mathbf{r}_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(2)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

bond_params: *type(bond_order_parameters) intent(in) size(2)* a `bond_order_parameters` containing the parameters

factor: double precision *intent(out) size(3)* the calculated bond order term c

atoms: *type(atom) intent(in) size(3)* a list of the actual `atom` objects for which the term is calculated

evaluate_bond_order_factor_triplet (*separations, distances, bond_params, factor, atoms*)

Triplet bond factor

Parameters:

separations: double precision *intent(in) size(3, 2)* atom-atom separation vectors \mathbf{r}_{12} , \mathbf{r}_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(2)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

bond_params: type(bond_order_parameters) *intent(in) size(2)* a `bond_order_parameters` containing the parameters

factor: double precision *intent(out) size(3)* the calculated bond order term c

atoms: type(atom) *intent(in) size(3)* a list of the actual `atom` objects for which the term is calculated

evaluate_bond_order_gradient (*n_targets, separations, distances, bond_params, gradient, atoms*)

Returns the gradients of bond order terms with respect to moving an atom.

By a bond order factor term, we mean the contribution from specific atoms, c_{ijk} , appearing in the factor

$$b_i = f\left(\sum_{jk} c_{ijk}\right)$$

This routine evaluates the gradient term $\nabla_{\alpha} c_{ij}$ or $\nabla_{\alpha} c_{ijk}$ for the given atoms ij or ijk according to the given parameters.

The returned array has three dimensions: gradient(coordinates, atom whose factor is differentiated, atom with respect to which we differentiate) So for example, for a three body term atom1-atom2-atom3, gradient(1,2,3) contains the x-coordinate (1), of the factor for atom2 (2), with respect to moving atom3 (3).

Parameters:

n_targets: integer *intent(in) scalar* number of targets

separations: double precision *intent(in) size(3, n_targets-1)* atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(n_targets-1)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

bond_params: type(bond_order_parameters) *intent(in) size(n_targets-1)* a `bond_order_parameters` containing the parameters

gradient: double precision *intent(out) size(3, n_targets, n_targets)* the calculated bond order term $\nabla_{\alpha} c$

atoms: type(atom) *intent(in) size(n_targets)* a list of the actual `atom` objects for which the term is calculated

evaluate_bond_order_gradient_coordination (*separations, distances, bond_params, gradient*)

Coordination bond order factor gradient

Parameters:

separations: double precision *intent(in) size(3, 1)* atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(1)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

bond_params: type(bond_order_parameters) *intent(in) size(1)* a `bond_order_parameters` containing the parameters

gradient: double precision *intent(out) size(3, 2, 2)* the calculated bond order term c

evaluate_bond_order_gradient_power (*separations, distances, bond_params, gradient*)

Power bond order factor gradient

Parameters:

separations: double precision *intent(in) size(3, 1)* atom-atom separation vectors \mathbf{r}_{12} , \mathbf{r}_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(1)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

bond_params: type(bond_order_parameters) *intent(in) size(1)* a `bond_order_parameters` containing the parameters

gradient: double precision *intent(out) size(3, 2, 2)* the calculated bond order term c

evaluate_bond_order_gradient_tersoff (*separations, distances, bond_params, gradient, atoms*)

Coordination bond order factor gradient

Parameters:

separations: double precision *intent(in) size(3, 2)* atom-atom separation vectors \mathbf{r}_{12} , \mathbf{r}_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(2)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

bond_params: type(bond_order_parameters) *intent(in) size(2)* a `bond_order_parameters` containing the parameters

gradient: double precision *intent(out) size(3, 3, 3)* the calculated bond order term c

atoms: type(atom) *intent(in) size(3)* a list of the actual `atom` objects for which the term is calculated

evaluate_bond_order_gradient_triplet (*separations, distances, bond_params, gradient, atoms*)

Coordination bond order factor gradient

Parameters:

separations: double precision *intent(in) size(3, 2)* atom-atom separation vectors \mathbf{r}_{12} , \mathbf{r}_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(2)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

bond_params: type(bond_order_parameters) *intent(in) size(2)* a `bond_order_parameters` containing the parameters

gradient: double precision *intent(out) size(3, 3, 3)* the calculated bond order term c

atoms: type(atom) *intent(in) size(3)* a list of the actual `atom` objects for which the term is calculated

evaluate_electronegativity (*n_targets, separations, distances, interaction, eneg, atoms*)

If a potential, say, U_{ijk} depends on the charges of atoms q_i it will not only create a force, but also a difference in chemical potential μ_i for the atomic partial charges. Similarly to `evaluate_forces()`, this function evaluates the chemical ‘force’ on the atomic charges

$$\chi_{\alpha,ijk} = -\mu_{\alpha,ijk} = -\frac{\partial U_{ijk}}{\partial q_{\alpha}}$$

To be consist the forces returned by `evaluate_electronegativity()` must be derivatives of the energies returned by `evaluate_energy()`.

Parameters:

n_targets: integer *intent(in) scalar* number of targets

separations: double precision *intent(in) size(3, n_targets-1)* atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(n_targets-1)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: type(potential) *intent(in) scalar* a `potential` containing the parameters

eneg: double precision *intent(out) size(n_targets)* the calculated electronegativity component $\chi_{\alpha,ijk}$

atoms: type(atom) *intent(in) size(n_targets)* a list of the actual `atom` objects for which the term is calculated

evaluate_electronegativity_charge_exp (*separations, distances, interaction, eneg, atoms*)

Charge exp electronegativity

Parameters:

separations: double precision *intent(in) size(3, 1)* atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(1)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: type(potential) *intent(in) scalar* a `potential` containing the parameters

eneg: double precision *intent(out) size(2)* the calculated electronegativity component $\chi_{\alpha,ijk}$

atoms: type(atom) *intent(in) size(2)* a list of the actual `atom` objects for which the term is calculated

evaluate_energy (*n_targets, separations, distances, interaction, energy, atoms*)

Evaluates the potential energy due to an interaction between the given atoms. In other words, if the total potential energy is

$$E = \sum_{ijk} v_{ijk}$$

this routine evaluates v_{ijk} for the given atoms i, j, and k.

To be consist the forces returned by `evaluate_forces()` must be gradients of the energies returned by `evaluate_energy()`.

Parameters:

n_targets: integer *intent(in) scalar* number of targets

separations: double precision *intent(in) size(3, n_targets-1)* atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(n_targets-1)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: type(potential) *intent(in) scalar* a `bond_order_parameters` containing the parameters

energy: double precision *intent(out) scalar* the calculated energy v_{ijk}

atoms: `type(atom) intent(in) size(n_targets) optional` a list of the actual `atom` objects for which the term is calculated

evaluate_energy_LJ (*separations, distances, interaction, energy*)

LJ energy

Parameters:

separations: `double precision intent(in) size(3, 1)` atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: `double precision intent(in) size(1)` atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: `type(potential) intent(in) scalar` a `bond_order_parameters` containing the parameters

energy: `double precision intent(out) scalar` the calculated energy v_{ijk}

evaluate_energy_bond_bending (*separations, distances, interaction, energy, atoms*)

Bond bending energy

Parameters:

separations: `double precision intent(in) size(3, 2)` atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: `double precision intent(in) size(2)` atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: `type(potential) intent(in) scalar` a `bond_order_parameters` containing the parameters

energy: `double precision intent(out) scalar` the calculated energy v_{ijk}

atoms: `type(atom) intent(in) size(3)` a list of the actual `atom` objects for which the term is calculated

evaluate_energy_buckingham (*separations, distances, interaction, energy*)

Buckingham energy

Parameters:

separations: `double precision intent(in) size(3, 1)` atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: `double precision intent(in) size(1)` atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: `type(potential) intent(in) scalar` a `bond_order_parameters` containing the parameters

energy: `double precision intent(out) scalar` the calculated energy v_{ijk}

evaluate_energy_charge_exp (*separations, distances, interaction, energy, atoms*)

Charge exp energy

Parameters:

separations: `double precision intent(in) size(3, 1)` atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: `double precision intent(in) size(1)` atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: `type(potential) intent(in) scalar` a `bond_order_parameters` containing the parameters

energy: `double precision intent(out) scalar` the calculated energy v_{ijk}

atoms: `type(atom) intent(in) size(2)` a list of the actual `atom` objects for which the term is calculated

evaluate_energy_constant_force (*interaction, energy, atoms*)

constant force energy

Parameters:

interaction: `type(potential) intent(in) scalar` a `bond_order_parameters` containing the parameters

energy: `double precision intent(out) scalar` the calculated energy v_{ijk}

atoms: `type(atom) intent(in) size(1)` a list of the actual `atom` objects for which the term is calculated

evaluate_energy_constant_potential (*interaction, energy*)

Constant potential energy

Parameters:

interaction: `type(potential) intent(in) scalar` a `bond_order_parameters` containing the parameters

energy: `double precision intent(out) scalar` the calculated energy v_{ijk}

evaluate_energy_dihedral (*separations, distances, interaction, energy, atoms*)

Dihedral angle energy

Parameters:

separations: `double precision intent(in) size(3, 3)` atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: `double precision intent(in) size(3)` atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: `type(potential) intent(in) scalar` a `bond_order_parameters` containing the parameters

energy: `double precision intent(out) scalar` the calculated energy v_{ijk}

atoms: `type(atom) intent(in) size(4)` a list of the actual `atom` objects for which the term is calculated

evaluate_energy_power (*separations, distances, interaction, energy*)

Power energy

Parameters:

separations: `double precision intent(in) size(3, 1)` atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: `double precision intent(in) size(1)` atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: `type(potential) intent(in) scalar` a `bond_order_parameters` containing the parameters

energy: `double precision intent(out) scalar` the calculated energy v_{ijk}

evaluate_energy_spring (*separations, distances, interaction, energy*)

spring energy

Parameters:

separations: double precision *intent(in) size(3, 1)* atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(1)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: type(potential) *intent(in) scalar* a `bond_order_parameters` containing the parameters

energy: double precision *intent(out) scalar* the calculated energy v_{ijk}

evaluate_force_LJ (*separations, distances, interaction, force*)

LJ force

Parameters:

separations: double precision *intent(in) size(3, 1)* atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(1)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: type(potential) *intent(in) scalar* a `potential` containing the parameters

force: double precision *intent(out) size(3, 2)* the calculated force component $f_{\alpha,ijk}$

evaluate_force_bond_bending (*separations, distances, interaction, force, atoms*)

Bond bending force

Parameters:

separations: double precision *intent(in) size(3, 2)* atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(2)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: type(potential) *intent(in) scalar* a `potential` containing the parameters

force: double precision *intent(out) size(3, 3)* the calculated force component $f_{\alpha,ijk}$

atoms: type(atom) *intent(in) size(3)* a list of the actual `atom` objects for which the term is calculated

evaluate_force_buckingham (*separations, distances, interaction, force*)

Buckingham force

Parameters:

separations: double precision *intent(in) size(3, 1)* atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(1)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: type(potential) *intent(in) scalar* a `potential` containing the parameters

force: double precision *intent(out) size(3, 2)* the calculated force component $f_{\alpha,ijk}$

evaluate_force_charge_exp (*separations, distances, interaction, force, atoms*)

charge exp force

Parameters:

separations: double precision *intent(in) size(3, 1)* atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(1)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: type(potential) *intent(in) scalar* a `potential` containing the parameters

force: double precision *intent(out) size(3, 2)* the calculated force component $f_{\alpha,ijk}$

atoms: type(atom) *intent(in) size(2)* a list of the actual `atom` objects for which the term is calculated

evaluate_force_constant_force (*interaction, force*)

constant force

Parameters:

interaction: type(potential) *intent(in) scalar* a `potential` containing the parameters

force: double precision *intent(out) size(3, 1)* the calculated force component $f_{\alpha,ijk}$

evaluate_force_constant_potential (*interaction, force*)

constant force

Parameters:

interaction: type(potential) *intent(in) scalar* a `potential` containing the parameters

force: double precision *intent(out) size(3, 1)* the calculated force component $f_{\alpha,ijk}$

evaluate_force_dihedral (*separations, distances, interaction, force, atoms*)

Dihedral angle force

Parameters:

separations: double precision *intent(in) size(3, 3)* atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(3)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: type(potential) *intent(in) scalar* a `potential` containing the parameters

force: double precision *intent(out) size(3, 4)* the calculated force component $f_{\alpha,ijk}$

atoms: type(atom) *intent(in) size(4)* a list of the actual `atom` objects for which the term is calculated

evaluate_force_power (*separations, distances, interaction, force*)

Power force

Parameters:

separations: double precision *intent(in) size(3, 1)* atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: double precision *intent(in) size(1)* atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: type(potential) *intent(in) scalar* a `potential` containing the parameters

force: double precision intent(out) size(3, 2) the calculated force component $f_{\alpha,ijk}$

evaluate_force_spring (separations, distances, interaction, force)

spring force

Parameters:

separations: double precision intent(in) size(3, 1) atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: double precision intent(in) size(1) atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: type(potential) intent(in) scalar a `potential` containing the parameters

force: double precision intent(out) size(3, 2) the calculated force component $f_{\alpha,ijk}$

evaluate_forces (n_targets, separations, distances, interaction, force, atoms)

Evaluates the forces due to an interaction between the given atoms. In other words, if the total force on atom α is

$$\mathbf{F}_{\alpha} = \sum_{ijk} -\nabla_{\alpha} v_{ijk} = \sum \mathbf{f}_{\alpha,ijk},$$

this routine evaluates $f_{\alpha,ijk}$ for $\alpha = (i, j, k)$ for the given atoms i , j , and k .

To be consist the forces returned by `evaluate_forces()` must be gradients of the energies returned by `evaluate_energy()`.

Parameters:

n_targets: integer intent(in) scalar number of targets

separations: double precision intent(in) size(3, n_targets-1) atom-atom separation vectors r_{12} , r_{23} etc. for the atoms 123...

distances: double precision intent(in) size(n_targets-1) atom-atom distances r_{12} , r_{23} etc. for the atoms 123..., i.e., the norms of the separation vectors.

interaction: type(potential) intent(in) scalar a `potential` containing the parameters

force: double precision intent(out) size(3, n_targets) the calculated force component $f_{\alpha,ijk}$

atoms: type(atom) intent(in) size(n_targets) a list of the actual `atom` objects for which the term is calculated

get_bond_descriptor (bond_name, descriptor)

Returns the `bond_order_descriptor` of a given name.

Parameters:

bond_name: character(len=*) intent(in) scalar name of the bond order factor

descriptor: type(bond_order_descriptor) intent(out) scalar the `bond_order_descriptor` matching

get_description_of_bond_order_factor (bond_name, description)

Returns the description of a bond order factor.

Parameters:

bond_name: character(len=*) intent(in) scalar name of the bond order factor

description: character(len=pot_note_length) intent(out) scalar description of the bond order factor

get_description_of_potential (*pot_name, description*)
Returns the description of a potential.

Parameters:

pot_name: *character(len=*) intent(in) scalar* name of the potential

description: *character(len=pot_note_length) intent(out) scalar* description of the potential

get_descriptions_of_parameters_of_bond_order_factor (*bond_name, n_targets, param_notes*)

Returns the descriptions of the parameters of a bond order factor as a list of strings.

Parameters:

bond_name: *character(len=*) intent(in) scalar* name of the bond order factor

n_targets: *integer intent(in) scalar* number of targets

param_notes: *character(len=param_note_length) intent() pointer size(:)* descriptions of the parameters

get_descriptions_of_parameters_of_potential (*pot_name, param_notes*)

Returns the descriptions of the parameters of a potential as a list of strings.

Parameters:

pot_name: *character(len=*) intent(in) scalar* name of the potential

param_notes: *character(len=param_note_length) intent() pointer size(:)* descriptions of the parameters

get_descriptor (*pot_name, descriptor*)

Returns the *potential_descriptor* of a given name.

Parameters:

pot_name: *character(len=*) intent(in) scalar* name of the potential

descriptor: *type(potential_descriptor) intent(out) scalar* the *potential_descriptor* matching

get_index_of_bond_order_factor (*bond_name, index*)

Returns the index of a *bond_order_descriptor* in the internal list of bond order factor types *bond_order_descriptors*.

Parameters:

bond_name: *character(len=*) intent(in) scalar* name of the bond order factor - a keyword

index: *integer intent(out) scalar* index of the potential in the internal array

get_index_of_parameter_of_bond_order_factor (*bond_name, param_name, index, n_targets*)

Returns the index of a parameter of a bond order factor in the internal list of parameters. Since bond order factors can have parameters for different number of targets, also the number of targets of this parameter is returned.

Parameters:

bond_name: *character(len=*) intent(in) scalar* name of the bond order factor

param_name: *character(len=*) intent(in) scalar* name of the parameter

index: *integer intent(out) scalar* index of the parameter

n_targets: integer intent(out) *scalar* number of targets of the parameter

get_index_of_parameter_of_potential (*pot_name, param_name, index*)
Returns the index of a parameter of a potential in the internal list of parameters.

Parameters:

pot_name: character(len=*) *intent(in) scalar* name of the potential

param_name: character(len=*) *intent(in) scalar* name of the parameter

index: integer intent(out) *scalar* the index of the parameter

get_index_of_potential (*pot_name, index*)
Returns the index of a `potential_descriptor` in the internal list of potential types `potential_descriptors`.

Parameters:

pot_name: character(len=*) *intent(in) scalar* name of the potential - a keyword

index: integer intent(out) *scalar* index of the potential in the internal array

get_names_of_parameters_of_bond_order_factor (*bond_name, n_targets, param_names*)

Returns the names of parameters of a bond order factor as a list of strings.

Parameters:

bond_name: character(len=*) *intent(in) scalar* name of the bond order factor

n_targets: integer *intent(in) scalar* number of targets

param_names: character(len=param_name_length) *intent()* *pointer size(:)* names of the parameters

get_names_of_parameters_of_potential (*pot_name, param_names*)
Returns the names of parameters of a potential as a list of strings.

Parameters:

pot_name: character(len=*) *intent(in) scalar* name of the potential

param_names: character(len=param_name_length) *intent()* *pointer size(:)* names of the parameters

get_number_of_bond_order_factors (*n_bond*)
Returns the number of `bond_order_descriptor` known.

Parameters:

n_bond: integer intent(out) *scalar* number of bond order factor types

get_number_of_parameters_of_bond_order_factor (*bond_name, n_targets, n_params*)

Returns the number of parameters of a bond order factor as a list of strings, each element showing the number of parameters for that number of bodies.

Parameters:

bond_name: character(len=*) *intent(in) scalar* name of the bond order factor

n_targets: integer *intent(in) scalar* number of targets

n_params: integer intent(out) *scalar* number of parameters

get_number_of_parameters_of_potential (*pot_name*, *n_params*)

Returns the number of parameters of a potential.

Parameters:

pot_name: *character(len=*) intent(in) scalar* name of the potential

n_params: *integer intent(out) scalar* number of parameters

get_number_of_potentials (*n_pots*)

Return the number of *potential_descriptor* known.

Parameters:

n_pots: *integer intent(out) scalar* number of potential types

get_number_of_targets_of_bond_order_factor (*bond_name*, *n_target*)

Returns the number of targets (i.e., bodies) of a bond order factor.

Parameters:

bond_name: *character(len=*) intent(in) scalar* name of the bond order factor

n_target: *integer intent(out) scalar* number of targets

get_number_of_targets_of_bond_order_factor_index (*bond_index*, *n_target*)

Returns the number of targets (i.e., bodies) of a bond order factor specified by its index.

Parameters:

bond_index: *integer intent(in) scalar* index of the bond order factor

n_target: *integer intent(out) scalar* number of targets

get_number_of_targets_of_potential (*pot_name*, *n_target*)

Returns the number of targets (i.e., bodies) of a potential.

Parameters:

pot_name: *character(len=*) intent(in) scalar* name of the potential

n_target: *integer intent(out) scalar* number of targets

get_number_of_targets_of_potential_index (*pot_index*, *n_target*)

Returns the number of targets (i.e., bodies) of a potential specified by its index.

Parameters:

pot_index: *integer intent(in) scalar* index of the potential

n_target: *integer intent(out) scalar* numner of targets

initialize_bond_order_factor_characterizers ()

Creates bond order factor characterizers.

This routine is meant to be run once, as pysic is imported, to create the characterizers for bond order factors. Once created, they are accessible by both the python and fortran sides of pysic as a tool for describing the general structure of bond order factor objects.

initialize_potential_characterizers ()

Creates potential characterizers.

This routine is meant to be run once, as pysic is imported, to create the characterizers for potentials. Once created, they are accessible by both the python and fortran sides of pysic as a tool for describing the general structure of potential objects.

is_valid_bond_order_factor (*string, is_valid*)

Returns true if the given keyword is the name of a bond order factor and false otherwise.

Parameters:

string: **character(len=*) intent(in) scalar** name of a bond order factor

is_valid: **logical intent(out) scalar** true if string is a name of a bond order factor

is_valid_potential (*string, is_valid*)

Returns true if the given keyword is the name of a potential and false otherwise.

Parameters:

string: **character(len=*) intent(in) scalar** name of a potential

is_valid: **logical intent(out) scalar** true if string is a name of a potential

list_bond_order_factors (*n_bonds, bonds*)

Returns the names of `bond_order_descriptor` objects.

Parameters:

n_bonds: **integer intent(in) scalar** number of bond order factor types

bonds: **character(len=pot_name_length) intent(out) size(n_bonds)** names of the bond order factor types

list_potentials (*n_pots, pots*)

Returns the names of `potential_descriptor` objects.

Parameters:

n_pots: **integer intent(in) scalar** number of potential types

pots: **character(len=pot_name_length) intent(out) size(n_pots)** names of the potential types

post_process_bond_order_factor (*raw_sum, bond_params, factor_out*)

Bond-order post processing, i.e., application of per-atom scaling functions.

By post processing, we mean any operations done after calculating the sum of pair- and many-body terms. That is, if a factor is, say,

$$b_i = f\left(\sum_j c_{ij}\right) = 1 + \sum_j c_{ij},$$

the $\sum_j c_{ij}$ would have been calculated already and the operation $f(x) = 1 + x$ remains to be carried out. The post processing is done per atom regardless of if the bond factor is of a pair or many body type.

This routine applies the scaling function on the given bond order sum according to the given parameters.

Parameters:

raw_sum: **double precision intent(in) scalar** the precalculated bond order sum, $\sum_j c_{ij}$ in the above example

bond_params: **type(bond_order_parameters) intent(in) scalar** a `bond_order_parameters` specifying the parameters

factor_out: **double precision intent(out) scalar** the calculated bond order factor b_i

post_process_bond_order_factor_scaler_1 (*raw_sum, bond_params, factor_out*)

Scaler post process factor

Parameters:

raw_sum: double precision intent(in) scalar the precalculated bond order sum, $\sum_j c_{ij}$ in the above example

bond_params: type(bond_order_parameters) intent(in) scalar a `bond_order_parameters` specifying the parameters

factor_out: double precision intent(out) scalar the calculated bond order factor b_i

post_process_bond_order_factor_scaler_sqrt (*raw_sum*, *bond_params*, *factor_out*)

Square root scaler post process factor

Parameters:

raw_sum: double precision intent(in) scalar the precalculated bond order sum, $\sum_j c_{ij}$ in the above example

bond_params: type(bond_order_parameters) intent(in) scalar a `bond_order_parameters` specifying the parameters

factor_out: double precision intent(out) scalar the calculated bond order factor b_i

post_process_bond_order_factor_tersoff (*raw_sum*, *bond_params*, *factor_out*)

Tersoff post process factor

Parameters:

raw_sum: double precision intent(in) scalar the precalculated bond order sum, $\sum_j c_{ij}$ in the above example

bond_params: type(bond_order_parameters) intent(in) scalar a `bond_order_parameters` specifying the parameters

factor_out: double precision intent(out) scalar the calculated bond order factor b_i

post_process_bond_order_gradient (*raw_sum*, *raw_gradient*, *bond_params*, *factor_out*)

Bond-order post processing, i.e., application of per-atom scaling functions.

By post processing, we mean any operations done after calculating the sum of pair- and many-body terms. That is, if a factor is, say,

$$b_i = f\left(\sum_j c_{ij}\right) = 1 + \sum_j c_{ij},$$

the $\sum_j c_{ij}$ would have been calculated already and the operation $f(x) = 1 + x$ remains to be carried out. The post processing is done per atom regardless of if the bond factor is of a pair or many body type.

For gradients, one needs to evaluate

$$\nabla_\alpha b_i = f'\left(\sum_j c_{ij}\right) \nabla_\alpha \sum_j c_{ij}$$

This routine applies the scaling function on the given bond order sum and gradient according to the given parameters.

Parameters:

raw_sum: double precision intent(in) scalar the precalculated bond order sum, $\sum_j c_{ij}$ in the above example

raw_gradient: double precision intent(in) size(3) the precalculated bond order gradient sum, $\nabla_\alpha \sum_j c_{ij}$ in the above example

bond_params: `type(bond_order_parameters) intent(in) scalar` a
`bond_order_parameters` specifying the parameters

factor_out: `double precision intent(out) size(3)` the calculated bond order factor $\nabla_{\alpha} b_i$

post_process_bond_order_gradient_scaler_1 (`raw_sum,` `raw_gradient,`
`bond_params,` `factor_out`)

Scaler post process gradient

Parameters:

raw_sum: `double precision intent(in) scalar` the precalculated bond order sum, $\sum_j c_i j$ in the
above example

raw_gradient: `double precision intent(in) size(3)` the precalculated bond order gradient sum,
 $\nabla_{\alpha} \sum_j c_i j$ in the above example

bond_params: `type(bond_order_parameters) intent(in) scalar` a
`bond_order_parameters` specifying the parameters

factor_out: `double precision intent(out) size(3)` the calculated bond order factor b_i

post_process_bond_order_gradient_scaler_sqrt (`raw_sum,` `raw_gradient,`
`bond_params,` `factor_out`)

Square root scaler post process gradient

Parameters:

raw_sum: `double precision intent(in) scalar` the precalculated bond order sum, $\sum_j c_i j$ in the
above example

raw_gradient: `double precision intent(in) size(3)` the precalculated bond order gradient sum,
 $\nabla_{\alpha} \sum_j c_i j$ in the above example

bond_params: `type(bond_order_parameters) intent(in) scalar` a
`bond_order_parameters` specifying the parameters

factor_out: `double precision intent(out) size(3)` the calculated bond order factor b_i

post_process_bond_order_gradient_tersoff (`raw_sum,` `raw_gradient,`
`bond_params,` `factor_out`)

Tersoff post process gradient

Parameters:

raw_sum: `double precision intent(in) scalar` the precalculated bond order sum, $\sum_j c_i j$ in the
above example

raw_gradient: `double precision intent(in) size(3)` the precalculated bond order gradient sum,
 $\nabla_{\alpha} \sum_j c_i j$ in the above example

bond_params: `type(bond_order_parameters) intent(in) scalar` a
`bond_order_parameters` specifying the parameters

factor_out: `double precision intent(out) size(3)` the calculated bond order factor b_i

potential_affects_atom (`interaction,` `atom_in,` `affects,` `position`)

Tests whether the given potential affects the specific atom.

For potentials, the atoms are specified as valid targets by the atomic symbol, index, or tag.

If position is not given, then the routine returns true if the atom can appear in the potential in any
role. If position is given, then true is returned only if the atom is valid for that particular position.

For instance, in a 3-body potential A-B-C, the potential May be specified so that only certain elements are valid for positions A and C while some other elements are valid for B. In a water molecule, for instance, we could have an H-O-H bond bending potential, but no H-H-O potentials.

Parameters:

interaction: *type(potential) intent(in) scalar* the `potential`

atom_in: *type(atom) intent(in) scalar* the `atom`

affects: *logical intent(out) scalar* true if the potential affects the atom

position: *integer intent(in) scalar optional* specifies the particular role of the atom in the interaction

smoothening_derivative (*r, hard_cut, soft_cut, factor*)

Derivative of the function for smoothening potential and bond order cutoffs. In principle any “nice” function which goes from 1 to 0 in a finite interval could be used. Here, we choose

$$f(r) = \frac{1}{2} \left(1 + \cos \pi \frac{r - r_{\text{soft}}}{r_{\text{hard}} - r_{\text{soft}}} \right)$$

for $r \in [r_{\text{soft}}, r_{\text{hard}}]$. The derivative is then

$$f'(r) = \frac{\pi}{2(r_{\text{soft}} - r_{\text{hard}})} \sin \pi \frac{r - r_{\text{soft}}}{r_{\text{hard}} - r_{\text{soft}}}.$$

This routine takes as arguments r , r_{soft} , and r_{hard} , and returns the value of the derivative of the smoothening function.

Parameters:

r: *double precision intent(in) scalar* distance r

hard_cut: *double precision intent(in) scalar* the hard cutoff r_{hard}

soft_cut: *double precision intent(in) scalar* the soft cutoff r_{soft}

factor: *double precision intent(out) scalar* the calculated derivative of the smoothening factor

smoothening_factor (*r, hard_cut, soft_cut, factor*)

Function for smoothening potential and bond order cutoffs. In principle any “nice” function which goes from 1 to 0 in a finite interval could be used. Here, we choose

$$f(r) = \frac{1}{2} \left(1 + \cos \pi \frac{r - r_{\text{soft}}}{r_{\text{hard}} - r_{\text{soft}}} \right)$$

for $r \in [r_{\text{soft}}, r_{\text{hard}}]$.

This routine takes as arguments r , r_{soft} , and r_{hard} , and returns the value of the smoothening function.

Parameters:

r: *double precision intent(in) scalar* distance r

hard_cut: *double precision intent(in) scalar* the hard cutoff r_{hard}

soft_cut: *double precision intent(in) scalar* the soft cutoff r_{soft}

factor: *double precision intent(out) scalar* the calculated smoothening factor

smoothening_gradient (*unit_vector, r, hard_cut, soft_cut, gradient*)

Gradient of the function for smoothening potential and bond order cutoffs. In principle any “nice” function which goes from 1 to 0 in a finite interval could be used. Here, we choose

$$f(r) = \frac{1}{2} \left(1 + \cos \pi \frac{r - r_{\text{soft}}}{r_{\text{hard}} - r_{\text{soft}}} \right)$$

for $r \in [r_{\text{soft}}, r_{\text{hard}}]$. The derivative is then

$$f'(r) = \frac{\pi}{2(r_{\text{soft}} - r_{\text{hard}})} \sin \pi \frac{r - r_{\text{soft}}}{r_{\text{hard}} - r_{\text{soft}}}.$$

and the gradient with respect to r

$$\nabla f(r) = f'(r) \nabla r = f'(r) \hat{r}$$

where \hat{r} is the unit vector in the direction of \mathbf{r} .

This routine takes as arguments \hat{r} , r , r_{soft} , and r_{hard} , and returns the value of the gradient of the smoothening function.

Parameters:

unit_vector: double precision *intent(in) size(3)* the vector \hat{r}

r: double precision *intent(in) scalar* distance r

hard_cut: double precision *intent(in) scalar* the hard cutoff r_{hard}

soft_cut: double precision *intent(in) scalar* the soft cutoff r_{soft}

gradient: double precision *intent(out) size(3)* the calculated derivative of the smoothening factor

geometry (Geometry.f90)

A module for handling the geometric structure of the system.

Full documentation of global variables in geometry

label_length

integer *scalar parameter*

initial value = 2

the number of characters available for denoting chemical symbols

Full documentation of custom types in geometry

atom

Defines an atomic particle.

Contained data:

neighbor_list: type(neighbor_list) *scalar* the list of neighbors for the atom

index: integer *scalar* index of the atom

n_pots: integer *scalar* number of potentials that may affect the atom

tags: integer *scalar* integer tag

potential_indices: integer *pointer size(:)* the indices of the potentials for which this atom is a valid target at first position (see `potential_affects_atom()`)

potentials_listed: logical *scalar* logical tag for checking if the potentials affecting the atom have been listed in potential_indices

bond_indices: *integer pointer size(:)* the indices of the bond order factors for which this atom is a valid target at first position (see `bond_order_factor_affects_atom()`)

element: *character(len=label_length) scalar* the chemical symbol of the atom

charge: *double precision scalar* charge of the atom

subcell_indices: *integer size(3)* indices of the subcell containing the atom, used for fast neighbor searching (see `subcell`)

mass: *double precision scalar* mass of the atom

n_bonds: *integer scalar* number of bond order factors that may affect the atom

bond_order_factors_listed: *logical scalar* logical tag for checking if the bond order factors affecting the atom have been listed in `bond_indices`

position: *double precision size(3)* coordinates of the atom

momentum: *double precision size(3)* momentum of the atom

neighbor_list

Defines a list of neighbors for a single atom. The list contains the indices of the neighboring atoms as well as the periodic boundary condition (PBC) offsets.

The offsets are integer triplets showing how many times must the supercell vectors be added to the position of the neighbor to find the neighboring image in a periodic system. For example, let the supercell be:

```
[[1.0, 0, 0], [0, 1.0, 0], [0, 0, 1.0]],
```

i.e., a unit cube, with periodic boundaries. Now, if we have particles with coordinates:

```
a = [1.5, 0.5, 0.5]
b = [0.4, 1.6, 3.3]
```

the closest separation vector $\mathbf{r}_b - \mathbf{r}_a$ between the particles is:

```
[-.1, .1, -.2]
```

obtained if we add the vector of periodicity:

```
[1.0, -1.0, -3.0]
```

to the coordinates of particle b. The offset vector (for particle b, when listing neighbors of a) is then:

```
[1, -1, -3]
```

Note that if the system is small, one atom can in principle appear several times in the neighbor list with different offsets.

Contained data:

neighbors: *integer pointer size(:)* indices of the neighboring atoms

max_length: *integer scalar* The allocated length of the neighbor lists. To avoid deallocating and reallocating memory, extra space is reserved for the neighbors in case the number of neighbors increases during simulation (due to atoms moving).

pbc_offsets: *integer pointer size(:, :)* offsets for periodic boundaries for each neighbor

n_neighbors: *integer scalar* the number of neighbors in the lists

subcell

Subvolume, which is a part of the supercell containing the simulation.

The subcells are used in partitioning of the simulation space in subvolumes. This divisioning of the simulation cell is needed for quickly finding the neighbors of atoms (see also `pysic.FastNeighborList`). The fast neighbor search is based on dividing the system, locating the subcell in which each atom is located, and then searching for neighbors for each atom by only checking the adjacent subcells. For small subvolumes (short cutoffs) this method is much faster than a brute force algorithm that checks all atom pairs. It also scales $\mathcal{O}(n)$.

Contained data:

neighbors: **integer** *size*(3, -1:1, -1:1, -1:1) indices of the 3 x 3 x 3 neighboring subcells (note that the neighboring subcell 0,0,0 is the cell itself)

vector_lengths: **double precision** *size*(3) lengths of the vectors spanning the subcell

offsets: **integer** *size*(3, -1:1, -1:1, -1:1) integer offsets of the neighboring subcells - if a neighboring subcell is beyond a periodic border, the offset records the fact

max_atoms: **integer scalar** the maximum number of atoms the cell can contain in the currently allocated memory space

vectors: **double precision** *size*(3, 3) the vectors spanning the subcell

atoms: **integer pointer** *size*(:) indices of the atoms in this subcell

n_atoms: **integer scalar** the number of atoms contained by the subcell

indices: **integer** *size*(3) integer coordinates of the subcell in the subcell divisioning of the supercell

include: **logical** *size*(-1:1, -1:1, -1:1) A logical array noting if the neighboring subcells should be included in the neighbor search. Usually all neighbors are included, but in a non-periodic system, there is only a limited number of cells and once the system border is reached, this tag will be set to `.false.` to notify that there is no neighbor to be found.

supercell

Supercell containing the simulation.

The supercell is spanned by three vectors $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ stored as a 3×3 matrix in format

$$\mathbf{M} = \begin{bmatrix} v_{1,x} & v_{1,y} & v_{1,z} \\ v_{2,x} & v_{2,y} & v_{2,z} \\ v_{3,x} & v_{3,y} & v_{3,z} \end{bmatrix}.$$

Also the inverse cell matrix is kept for transformations between the absolute and fractional coordinates.

Contained data:

vector_lengths: **double precision** *size*(3) the lengths of the cell spanning vectors (stored to avoid calculating the vector norms over and over)

max_subcell_atom_count: **integer scalar** the maximum number of atoms any of the subcells has

n_splits: **integer** *size*(3) the number of subcells there are in the subdivisioning of the cell, in the directions of the spanning vectors

inverse_cell: **double precision** *size*(3, 3) the inverse of the cell matrix \mathbf{M}^{-1}

subcells: **type(subcell) pointer** *size*(:, :, :) an array of `subcell` subvolumes which partition the supercell

vectors: **double precision** *size*(3, 3) vectors spanning the supercell containing the system as a matrix \mathbf{M}

volume: double precision *scalar* volume of the cell

periodic: logical *size(3)* logical switch determining if periodic boundary conditions are applied in the directions of the three cell spanning vectors

reciprocal_cell: double precision *size(3, 3)* the reciprocal cell as a matrix, $\mathbf{M}_R = 2\pi(\mathbf{M}^{-1})^T$. That is, if \mathbf{b}_i are the reciprocal lattice vectors and \mathbf{a}_j the real space lattice vectors, then $\mathbf{b}_i \mathbf{a}_j = 2\pi\delta_{ij}$.

Full documentation of subroutines in geometry

absolute_coordinates (*relative, cell, position*)

Transforms from fractional to absolute coordinates.

Absolute coordinates are the coordinates in the normal *xyz* base,

$$\mathbf{r} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}.$$

Fractional coordinates are the coordinates in the base spanned by the vectors defining the supercell, $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$,

$$\mathbf{r} = \tilde{x}\mathbf{v}_1 + \tilde{y}\mathbf{v}_2 + \tilde{z}\mathbf{v}_3.$$

Notably, for positions inside the supercell, the fractional coordinates fall between 0 and 1.

Transformation between the two bases is given by the cell matrix

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \mathbf{M} \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \end{bmatrix}$$

Parameters:

relative: double precision *intent(in) size(3)* the fractional coordinates

cell: type(supercell) *intent(in) scalar* the supercell

position: double precision *intent(out) size(3)* the absolute coordinates

assign_bond_order_factor_indices (*n_bonds, atom_in, indices*)

Save the indices of bond order factors affecting an atom.

In bond order factor evaluation, it is important to loop over bond parameters quickly. As the evaluation of factors goes over atoms, atom pairs etc., it is useful to first filter the parameters by the first atom participating in the factor. Therefore, the atoms can be given a list of bond order parameters for which they are a suitable target as a 'first participant' (in a triplet A-B-C, A is the first participant).

Parameters:

n_bonds: integer *intent(in) scalar* number of bond order factors

atom_in: type(atom) *intent(inout) scalar* the atom for which the bond order factors are assigned

indices: integer *intent(in) size(n_bonds)* the indices of the bond order factors

assign_neighbor_list (*n_nbs, nbor_list, neighbors, offsets*)

Creates a neighbor list for one atom.

The neighbor list will contain an array of the indices of the neighboring atoms as well as periodicity offsets, as explained in [neighbor_list](#)

The routine takes the neighbor_list object to be created as an argument. If the list is empty, it is initialized. If the list already contains information, the list is emptied and refilled. If the previous list

has room to contain the new list (as in, it has enough allocated memory), no memory reallocation is done (since it will be slow if done repeatedly). Only if the new list is too long to fit in the reserved memory, the pointers are deallocated and reallocated.

Parameters:

n_nbs: integer *intent(in) scalar* number of neighbors

nbor_list: type(neighbor_list) *intent(inout) scalar* The list of neighbors to be created.

neighbors: integer *intent(in) size(n_nbs)* array containing the indices of the neighboring atoms

offsets: integer *intent(in) size(3, n_nbs)* periodicity offsets

assign_potential_indices (*n_pots, atom_in, indices*)

Save the indices of potentials affecting an atom.

In force and energy evaluation, it is important to loop over potentials quickly. As the evaluation of energies goes over atoms, atom pairs etc., it is useful to first filter the potentials by the first atom participating in the interaction. Therefore, the atoms can be given a list of potentials for which they are a suitable target as a ‘first participant’ (in a triplet A-B-C, A is the first participant).

Parameters:

n_pots: integer *intent(in) scalar* number of potentials

atom_in: type(atom) *intent(inout) scalar* the atom for which the potentials are assigned

indices: integer *intent(in) size(n_pots)* the indices of the potentials

divide_cell (*cell, splits*)

Split the cell in subcells according to the given number of divisions.

The argument ‘splits’ should be a list of three integers determining how many times the cell is split. For instance, if splits = [3,3,5], the cell is divided in $3*3*5 = 45$ subcells: 3 cells along the first two cell vectors and 5 along the third.

The Cell itself is not changed, but an array ‘subcells’ is created, containing the subcells which are Cell instances themselves. These cells will contain additional data arrays ‘neighbors’ and ‘offsets’. These are 3-dimensional arrays with each dimension running from -1 to 1. The neighbors array contains references to the neighboring subcell Cell instances. The offsets contain coordinate offsets with respect to the periodic boundaries. In other words, if a subcell is at the border of the original Cell, it will have neighbors at the other side of the cell due to periodic boundary conditions. But from the point of view of the subcell, the neighboring cell is not on the other side of the master cell, but a periodic image of that cell. Therefore, any coordinates in the the subcell to which the neighbors array refers to must in fact be shifted by a vector of the master cell. The offsets list contains the multipliers for the cell vectors to make these shifts.

Example in 2D for simplicity: split = [3,4] creates subcells:

```
(0,3) (1,3) (2,3)
(0,2) (1,2) (2,2)
(0,1) (1,1) (2,1)
(0,0) (1,0) (2,0)
```

subcell (0,3) will have the neighbors:: (2,0) (0,0) (1,0) (2,3) (0,3) (1,3) (2,2) (0,2) (1,2)

and offsets:: [-1,1] [0,1] [0,1] [-1,0] [0,0] [0,0] [-1,0] [0,0] [0,0]

Note that the central ‘neighbor’ is the cell itself.

If a boundary is not periodic, extra subcells with indices 0 and split+1 are created to pad the simulation cell. These will contain the atoms that are outside the simulation cell.

Parameters:

cell: type(supercell) **intent(inout)** *scalar*

splits: integer *intent(in)* *size(3)*

expand_subcell_atom_capacity (*atoms_list, old_size, new_size*)

Parameters:

atoms_list: integer *intent()* *pointer* *size(:)*

old_size: integer *intent(in)* *scalar*

new_size: integer *intent(in)* *scalar*

find_subcell_for_atom (*cell, at*)

Parameters:

cell: type(supercell) **intent(inout)** *scalar*

at: type(atom) **intent(inout)** *scalar*

generate_atoms (*n_atoms, masses, charges, positions, momenta, tags, elements, atoms*)

Creates atoms to construct the system to be simulated.

Parameters:

n_atoms: integer *intent(in)* *scalar* number of atoms

masses: double precision *intent(in)* *size(n_atoms)* array of masses for the atoms

charges: double precision *intent(in)* *size(n_atoms)* array of charges for the atoms

positions: double precision *intent(in)* *size(3, n_atoms)* array of coordinates for the atoms

momenta: double precision *intent(in)* *size(3, n_atoms)* array of momenta for the atoms

tags: integer *intent(in)* *size(n_atoms)* array of integer tags for the atoms

elements: character(len=label_length) *intent(in)* *size(n_atoms)* array of chemical symbols for the atoms

atoms: type(atom) *intent()* *pointer* *size(:)* array of the atom objects created

generate_supercell (*vectors, inverse, periodicity, cell*)

Creates the supercell containing the simulation geometry.

The supercell is spanned by three vectors $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ stored as a 3×3 matrix in format

$$\mathbf{M} = \begin{bmatrix} v_{1,x} & v_{1,y} & v_{1,z} \\ v_{2,x} & v_{2,y} & v_{2,z} \\ v_{3,x} & v_{3,y} & v_{3,z} \end{bmatrix}.$$

Also the inverse cell matrix \mathbf{M}^{-1} must be given for transformations between the absolute and fractional coordinates. However, it is not checked that the given matrix and inverse truly fulfill $\mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$ - it is the responsibility of the caller to give the true inverse.

Also the periodicity of the system in the directions of the cell vectors need to be given.

Parameters:

vectors: double precision *intent(in)* *size(3, 3)* the cell spanning matrix \mathbf{M}

inverse: double precision *intent(in)* *size(3, 3)* the inverse cell \mathbf{M}

periodicity: logical *intent(in)* *size(3)* logical switch, true if the boundaries are periodic

cell: type(supercell) **intent(out)** *scalar* the created cell object

get_optimal_splitting (*cell, max_cut, splits*)

Parameters:

cell: `type(supercell) intent(in) scalar`

max_cut: `double precision intent(in) scalar`

splits: `integer intent(out) size(3)`

relative_coordinates (*position, cell, relative*)

Transforms from absolute to fractional coordinates.

Absolute coordinates are the coordinates in the normal xyz base,

$$\mathbf{r} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}.$$

Fractional coordinates are the coordinates in the base spanned by the vectors defining the supercell, $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$,

$$\mathbf{r} = \tilde{x}\mathbf{v}_1 + \tilde{y}\mathbf{v}_2 + \tilde{z}\mathbf{v}_3.$$

Notably, for positions inside the supercell, the fractional coordinates fall between 0 and 1.

Transformation between the two bases is given by the inverse cell matrix

$$\begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \end{bmatrix} = \mathbf{M}^{-1} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Parameters:

position: `double precision intent(in) size(3)` the absolute coordinates

cell: `type(supercell) intent(in) scalar` the supercell

relative: `double precision intent(out) size(3)` the fractional coordinates

separation_vector (*r1, r2, offset, cell, separation*)

Calculates the minimum separation vector between two atoms, $\mathbf{r}_2 - \mathbf{r}_1$, including possible periodicity.

Parameters:

r1: `double precision intent(in) size(3)` coordinates of atom 1, \mathbf{r}_1

r2: `double precision intent(in) size(3)` coordinates of atom 2, \mathbf{r}_2

offset: `integer intent(in) size(3)` periodicity offset (see [neighbor_list](#))

cell: `type(supercell) intent(in) scalar` supercell spanning the system

separation: `double precision intent(out) size(3)` the calculated separation vector, $\mathbf{r}_2 - \mathbf{r}_1$

update_atomic_charges (*n_atoms, charges, atoms*)

Updates the charges of the given atoms. Other properties are not altered.

Parameters:

n_atoms: `integer intent(in) scalar` number of atoms

charges: `double precision intent(in) size(n_atoms)` new charges for the atoms

atoms: `type(atom) intent(out) pointer size(:)` the atoms to be edited

update_atomic_positions (*n_atoms, positions, momenta, atoms*)

Updates the positions and momenta of the given atoms. Other properties are not altered.

This is meant to be used during dynamic simulations or geometry optimization where the atoms are only moved around, not changed in other ways.

Parameters:

n_atoms: integer *intent(in) scalar* number of atoms

positions: double precision *intent(in) size(3, n_atoms)* new coordinates for the atoms

momenta: double precision *intent(in) size(3, n_atoms)* new momenta for the atoms

atoms: type(atom) *intent()* *pointer size(:)* the atoms to be edited

wrapped_coordinates (*position, cell, wrapped, offset*)

Wraps a general coordinate inside the supercell if the system is periodic.

In a periodic system, every particle has periodic images at intervals defined by the cell vectors $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$. That is, for a particle at \mathbf{r} , there are periodic images at

$$\mathbf{R} = \mathbf{r} + a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2 + a_3 \mathbf{v}_3$$

for all $a_1, a_2, a_3 \in \mathbf{Z}$. These are equivalent positions in the sense that if a particle is situated at any of one of them, the set of images is the same. Exactly one of the images is inside the cell - this routine gives the coordinates of that particular image.

If the system is periodic in only some directions, the wrapping is done only along those directions.

Parameters:

position: double precision *intent(in) size(3)* the absolute coordinates

cell: type(supercell) *intent(in) scalar* the supercell

wrapped: double precision *intent(out) size(3)* the wrapped absolute coordinates

offset: integer *intent(out) size(3) optional* wrapping offset, i.e., the number of times the cell vectors are added to the absolute coordinates in order to obtain the wrapped coordinates

Full documentation of functions in geometry

pick (*index1, index2, offset*)

A utility function for sorting the atoms.

The function return `true` if `index1 < index2` and `false` otherwise. If `index1 == index2`, the comparison is made through the separation vector. The vector is examined element at a time, and if a positive number is found, `true` is returned, if a negative one, `false`. For values of zero, the next element is examined.

The purpose for this function is to sort the atoms to prevent double counting when summing over pairs. In principle, a sum over pairs (i, j) can be done with $\frac{1}{2} \sum_{i \neq j}$, but this leads to evaluation of all elements twice (both (i, j) and (j, i) are considered separately). It is more efficient to evaluate $\sum_{i < j}$, where only one of (i, j) and (j, i) fulfill the condition.

A special case arises if interactions are so long ranged that an atom can see its own periodic images. Then, one will need to sum terms for atom pairs where both atoms have the same index $\sum_{\text{images}} \sum_{i, j}$ if they are in different periodic copies of the actual simulation cell. In order to still pick only one of the pairs (i, i') and (i', i) , we compare the offset vectors. If atom i' is in the neighboring cell of i in the first cell vector direction, it has an offset of $[1, 0, 0]$ and vice versa i has an

offset of $[-1, 0, 0]$ from i' . Instead of the index, the sorting $i' < i$ is then done by comparing these offset vectors, element by element.

Parameters:

index1: integer *intent(in) scalar* index of first atom

index2: integer *intent(in) scalar* index of second atom

offset: integer *intent(in) size(3)* pbc offset vector from atom1 to atom2

utility (Utility.f90)

A module containing utility functions and constants.

The module is a collection of standalone helper tools, not physically relevant functionality.

Full documentation of global variables in utility

pi

double precision *scalar parameter*

initial value = 4.d0 * datan(1.d0)

the constant π calculated as $\pi = 4 \arctan 1$

Full documentation of subroutines in utility

int2str (*length, ints, string_out*)

Transforms an integer array to string through a codec:

```
1 - a
2 - b
...
101 - A
102 - B
...
-1 - 1
-2 - 2
...
```

Unrecognized numbers are treated as white spaces.

The function is used for communicating string arrays between Python and Fortran over f2py.

Parameters:

length: integer *intent(in) scalar* string length

ints: integer *intent(in) size(length)* the integers

string_out: character(len=length) *intent(out) scalar* the string

pad_string (*str_in, str_length, str_out*)

Adds spaces after the given string to create a string of a certain length. If the given string is longer than the specified length, it is truncated.

This is used to ensure strings are of a certain length, since character arrays in Fortran may be forced to a certain length.

Parameters:

str_in: *character(len=*) intent(in) scalar* the original string

str_length: *integer intent(in) scalar* the required string length

str_out: *character(len=str_length) intent(out) scalar* the padded string

str2int (*length, string, ints*)

Transforms a string to an integer array through a codec:

```
1 - a
2 - b
...
101 - A
102 - B
...
-1 - 1
-2 - 2
...
```

Unrecognized characters are mapped to 0.

The function is used for communicating string arrays between Python and Fortran over f2py.

Parameters:

length: *integer intent(in) scalar* string length

string: *character(len=length) intent(in) scalar* the string

ints: *integer intent(out) size(length)* the integers

mpi (MPI.f90)

Module for Message Parsing Interface parallelization utilities.

This module handles the initialization of the MPI environment and assigns the cpus their indices. Parallelization is done by distributing atoms on the processors and the routine for doing this randomly is provided. Also tools for monitoring the loads of all the cpus and redistributing them are also implemented.

Full documentation of global variables in mpi

all_atoms

integer scalar

the total number of atoms

all_loads

double precision allocatable size(:)

list of the loads of all cpus

atom_buffer

integer allocatable size(:)

list used for passing atom indices during load balancing

cpu_id

integer scalar

identification number for the cpu, from 0 to $n_{\text{cpus}} - 1$

is_my_atom

logical *allocatable size(:)*

logical array, true for the indices of the atoms that are distributed to this cpu

load_length

integer *scalar*

the number of times loads have been recorded

loadout

integer *scalar*

initial value = 2352

an integer of the output channel for loads

loads_mask

logical *allocatable size(:)*

logical array used in load rebalancing, true for cpus whose loads have not yet been balanced

mpi_atoms_allocated

logical *scalar*

initial value = .false.

logical switch for denoting that the mpi allocatable arrays have been allocated

mpistat

integer *scalar*

mpi return value

mpistatus

integer *size(mpi_status_size)*

array for storing the mpi status return values

my_atoms

integer *scalar*

the number of atoms distributed to this cpu

my_load

double precision *scalar*

storage for the load of this particular cpu

n_cpus

integer *scalar*

number of cpus, n_{cpus}

stopwatch

double precision *scalar*

cpu time storage

track_loads

logical *scalar parameter*

initial value = .false.

logical switch, if true, the loads of cpus are written to a file during run

Full documentation of subroutines in mpi

balance_loads ()

Load balancing.

The loads are gathered from all cpus and sorted. Then load (atoms) is passed from the most loaded cpus to the least loaded ones.

close_loadmonitor ()

Closes the output for writing workload data

initialize_load (reallocate)

Initializes the load monitoring arrays.

Parameters:

reallocate: logical intent(in) scalar Logical switch for reallocating the arrays. If true, the related arrays are allocated. Otherwise only the load counters are set to zero.

mpi_distribute (n_atoms)

distributes atoms among processors

Parameters:

n_atoms: integer intent(in) scalar number of atoms

mpi_finish ()

closes the mpi framework

mpi_initialize ()

initializes the mpi framework

mpi_master_bcast_int (sync_int)

the master cpu broadcasts an integer value to all other cpus

Parameters:

sync_int: integer intent(inout) scalar the broadcast integer

mpi_stack (list, items, depth, length, width)

stacks the "lists" from all cpus together according to the lengths given in "items" and gathers the complete list to cpu 0. For example:

cpu 0	cpu 1	cpu 0
abc....	12.....	abc12..
de.....	3456...	-> de3456.
fghij..	78.....	fghij78

The stacking is done for the second array index: list(1,:,1). The stacking works so that first every cpu $2n+1$ sends its data to cpu $2n$, then $2*(2n+1)$ sends data to $2*2n$, and so on, until the final cpu 2^m sends its data to cpu 0:

```

cpu
0 1 2 3 4 5 6 7 8 9 10
|-/  |-/  |-/  |-/  |-/  |
|---/  |---/  |---/
|-----/  |
|-----/
x

```

Parameters:

list: INTEGER intent() size(:, :, :) 3d arrays containing lists to be stacked

items: **INTEGER** *intent()* **size(:)** the numbers of items to be stacked in each list

depth: **INTEGER** *intent()* **scalar** dimensionality of the stacked objects (size of list(:,1,1))

length: **INTEGER** *intent()* **scalar** the number of lists (size of list(1,1,:))

width: **INTEGER** *intent()* **scalar** max size of lists (size of list(1,:,1))

mpi_sync ()

syncs the cpus by calling mpi_barrier

mpi_wall_clock (clock)

returns the global time through mpi_wtime

Parameters:

clock: **double precision** *intent(out)* **scalar** the measured time

open_loadmonitor ()

Opens the output for writing workload data to a file called “mpi_load.out”

record_load (amount)

Saves the given load.

Parameters:

amount: **double precision** *intent(in)* **scalar** the load to be stored

start_timer ()

records the wall clock time to `stopwatch`

timer (stamp)

reads the elapsed wall clock time since the previous starting of the timer (saved in `stopwatch`) and then restarts the timer.

Parameters:

stamp: **double precision** *intent(inout)* **scalar** the elapsed real time

write_loadmonitor ()

Routine for writing force calculation workload analysis data to a file called “mpi_load.out”

quaternions (Quaternions.f90)

A module for basic quaternion operations and 3D spatial rotations using quaternion representation. Quaternions are a 4-component analogue to complex numbers.

$$\mathbf{q} = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k} = [w, x, y, z]$$

where the three imaginary components obey $\mathbf{i}\mathbf{j}\mathbf{k} = \mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -1$. This leads to a structure similar to that of complex numbers, except that the quaternion product defined according to the above rules is non-commutative $\mathbf{q}_1\mathbf{q}_2 \neq \mathbf{q}_2\mathbf{q}_1$.

It turns out that unit quaternions $||\mathbf{q}|| = \sqrt{w^2 + x^2 + y^2 + z^2} = 1$ represent the space of 3D rotations so that the rotation by angle α around unit axis $\mathbf{u} = [x, y, z]$ is represented by the quaternion

$$\mathbf{q} = [\cos \frac{\alpha}{2}, x \sin \frac{\alpha}{2}, y \sin \frac{\alpha}{2}, z \sin \frac{\alpha}{2}]$$

and joining rotations is represented as a quaternion product (rotating first by \mathbf{q}_1 , then by \mathbf{q}_2 yields the combined rotation of $\mathbf{q}_{12} = \mathbf{q}_2\mathbf{q}_1$).

Full documentation of global variables in quaternions

norm_tolerance

double precision *scalar parameter*

initial value = 1.0d-8

the threshold value for the norm for treating vectors as zero vectors

Full documentation of custom types in quaternions

qtrn

The quaternion type. It only contains four real components, but the main advantage for defining it as a custom type is the possibility to write routines and operators for quaternion algebra.

Contained data:

y: double precision scalar an “imaginary” component of the quaternion

x: double precision scalar an “imaginary” component of the quaternion

z: double precision scalar an “imaginary” component of the quaternion

w: double precision scalar the “real” component of the quaternion

Full documentation of subroutines in quaternions

norm_quaternion (qq)

norms the given quaternion

Parameters:

qq: TYPE(qtrn) intent() scalar quaternion to be normed to unity

Full documentation of functions in quaternions

cross (v, u)

Normal cross product of vectors $\mathbf{v} \times \mathbf{u}$ (Note: for 3-vectors only!)

Parameters:

v: double precision intent() size(3) vector

u: double precision intent() size(3) vector

dot (v, u)

Normal dot product of vectors $\mathbf{v} \cdot \mathbf{u}$ (Note: for 3-vectors only!)

Parameters:

v: double precision intent() size(3) vector

u: double precision intent() size(3) vector

q2angle (q)

Returns the angle of rotation described by the UNIT quaternion **q**. Note that the unity of **q** is not checked (as it would be time consuming to calculate the norm all the time if we know the quaternions used have unit length).

Parameters:

q: TYPE(qtrn) intent() scalar a quaternion representation of rotation

q2axis (*q*)

Returns the axis of rotation described by the UNIT quaternion **q**. Note that the unity of **q** is not checked (as it would be time consuming to calculate the norm all the time if we know the quaternions used have unit length).

Parameters:

q: TYPE(qtrn) intent() scalar a quaternion representation of rotation

q2matrix (*q*)

Returns the rotation matrix described by the UNIT quaternion **q**. Note that the unity of **q** is not checked (as it would be time consuming to calculate the norm all the time if we know the quaternions used have unit length).

Parameters:

q: TYPE(qtrn) intent() scalar a quaternion representation of rotation

qconj (*q*)

Returns the quaternion conjugate of **q**: $\mathbf{q}^* = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k} \rightarrow w - x\mathbf{i} - y\mathbf{j} - z\mathbf{k}$

Parameters:

q: TYPE(qtrn) intent() scalar a quaternion

qdiv (*q, r*)

Returns the quaternion **q** divided by scalar *r* component-wise

Parameters:

q: TYPE(qtrn) intent() scalar a quaternion

r: double precision intent() scalar a real scalar

qinv (*q*)

Returns the quaternion inverse of **q**: $\mathbf{q}^* / \|\mathbf{q}\|$

Parameters:

q: TYPE(qtrn) intent() scalar a quaternion

qminus (*q, r*)

Returns the quaternion **q** subtracted by scalar *r* component-wise

Parameters:

q: TYPE(qtrn) intent() scalar a quaternion

r: double precision intent() scalar a real scalar

qnorm (*q*)

Returns the quaternion norm of **q**: $\|\mathbf{q}\| = \sqrt{w^2 + x^2 + y^2 + z^2}$

Parameters:

q: TYPE(qtrn) intent() scalar a quaternion

qplus (*q, r*)

Returns the quaternion **q** added by scalar *r* component-wise

Parameters:

q: TYPE(qtrn) intent() scalar a quaternion

r: double precision intent() scalar a real scalar

qprod (*q1, q2*)

Returns the quaternion product $\mathbf{q}_1 \mathbf{q}_2$. Note that the product is non-commutative: $\mathbf{q}_1 \mathbf{q}_2 \neq \mathbf{q}_2 \mathbf{q}_1$.

Parameters:

q1: **TYPE(qtrn) intent() scalar** a quaternion

q2: **TYPE(qtrn) intent() scalar** a quaternion

qtimes (*r, q*)

Returns the quaternion \mathbf{q} multiplied by scalar r component-wise.

Parameters:

r: **double precision intent() scalar** a real scalar

q: **TYPE(qtrn) intent() scalar** a quaternion

qtimesB (*q, r*)

Returns the quaternion \mathbf{q} multiplied by scalar r component-wise.

Parameters:

q: **TYPE(qtrn) intent() scalar** a quaternion

r: **double precision intent() scalar** a real scalar

rot2q (*a, u*)

Returns the quaternion representing a rotation around axis \mathbf{u} by angle α .

Parameters:

a: **double precision intent() scalar** angle in radians

u: **double precision intent() size(3)** 3D vector, defining an axis of rotation

rotate_a (*vec, da*)

Returns the vector rotated according to the vector \mathbf{d} . The axis of rotation is given by the direction of \mathbf{d} and the angle by $\|\mathbf{d}\|$.

Parameters:

vec: **double precision intent() size(3)** vector to be rotated

da: **double precision intent() size(3)** rotation vector (e.g., angular velocity \times time ωt)

rotate_au (*vec, a, u*)

Returns the vector rotated according to the axis \mathbf{u} and angle α .

Parameters:

vec: **double precision intent() size(3)** vector to be rotated

a: **double precision intent() scalar** angle of rotation

u: **double precision intent() size(3)** axis of rotation

rotate_q (*vec, q*)

Returns the 3D vector rotated according to the UNIT quaternion \mathbf{q} . Note that the unity of \mathbf{q} is not checked (as it would be time consuming to calculate the norm all the time if we know the quaternions used have unit length).

Parameters:

vec: **double precision intent() size(3)** vector to be rotated

q: **TYPE(qtrn) intent() scalar** a quaternion representation of rotation

vec2q (*v*)

Returns the quaternion representing a rotation around axis **v** by angle $\|\mathbf{v}\|$. If **v** = 0, the quaternion **q** = [1000] will be returned.

Parameters:

v: **double precision intent()** *size(3)* 3D vector, defining both the angle and axis of rotation

vnorm (*v*)

Norm of a vector, $\|\mathbf{v}\|$

Parameters:

v: **double precision intent()** *size(3)* vector

mt95 (Mersenne.f90)

The module mt95 contains a random number generator. Currently ‘Mersenne twister’ is used, but it could in principle be replaced with any generator. This is an external module not written as part of the Pysic project.

The generator is initialized with the routine `genrand_init()` and random real numbers in [0,1] and [0,1) are extracted with `genrand_real1()` and `genrand_real2()`, respectively.

Routines of the mt95 module

genrand_init (*seed*)

genrand_real1 (*random*)

genrand_real2 (*random*)

DEVELOPMENT OF PYSIC

5.1 Version history

This is a list of the main updates in the different versions of Pysic.

5.1.1 Version 0.4.3

- Major restructuring of the Python source code
- Added calculation of the stress tensor with the method `pysic.calculator.Pysic.get_stress()`.
- Bug fix: Fixed an issue with core initialization where changing the size of the supercell would lead to a conflict in neighbor list updating (the list update was tried before the cell update but failed due to the cell having been changed).
- Bug fix: Fixed an issue with the parallel neighbor list building algorithm which did not properly broadcast the calculated lists to all cpus.

5.1.2 Version 0.4.2

- Restructured the interaction evaluation loops in the Fortran core (*potentials (Potentials.f90)*)
- Added support for 4-body potentials
- Added the *Dihedral angle potential*
- Added the *Buckingham potential*
- Added the *Power decay potential*
- Added the *Power decay bond order factor*
- Added the *Square root scaling function*
- Bug fix: fixed a memory issue in Ewald summation `CoulombSummation`
- Bug fix: fixed an issue with periodic boundaries in `FastNeighborList`
- Bug fix: fixed an issue with special parameter values in Tersoff bond order factor evaluation
- Bug fix: fixed an issue where the cutoff of a bond order factor could overwrite a longer cutoff a potential
- Bug fix: fixed an indexing error in evaluation of 3-body interaction which gave to incorrect forces
- Bug fix: fixed and indexing error in neighbor offsets in `FastNeighborList`

5.1.3 Version 0.4.1

- Implemented an order $\mathcal{O}(n)$ neighbor finding algorithm in Fortran (see `pysic.FastNeighborList`)

5.1.4 Version 0.4

- Implemented the Ewald summation of $\frac{1}{r}$ potentials (see `pysic.CoulombSummation`)
- The framework allows for the addition of other summation methods later on, but for now only standard Ewald is available

5.1.5 Version 0.3

- Implemented framework for charge relaxation (see `pysic.ChargeRelaxation`)
- Implemented the *Damped dynamics* charge relaxation algorithm.
- Implemented the *Charge dependent exponential potential* potential.

5.1.6 Version 0.2

- Implemented bond order factors (see `pysic.Coordinator` and `pysic.BondOrderParameters`) for scaling of potential energy according to local bond structure.
- Implemented a more robust method for tracking the status of the Fortran core (see `pysic.CoreMirror`). This makes it less likely that wrong results are produced due to the changes in the user interface not propagating to the core.

5.1.7 Version 0.1

- First publicly available version
- Python interface
 - `pysic`
 - `pysic.calculator.Pysic`
 - `pysic.Potential`
 - `pysic_utility`
- Framework for handling pair- and three-body potentials
- ASE compatibility
 - `pysic.calculator.Pysic.get_forces()`
 - `pysic.calculator.Pysic.get_potential_energy()`

5.2 Roadmap

This is a list of the planned major features to be implemented in Pysic.

5.2.1 Version 0.5

- Implementation of the COMB potential

5.2.2 Version 0.6

- Implementation of the ReaxFF potential

5.2.3 Other planned features

- Extensive library of potentials, including tabulated potentials
- Option for other charge relaxation algorithms
- Option for other long range interaction summation algorithms
- Convenience tools for generating parameters for Ewald summation
- Convenience tools for generating chemical symbol lists for Potentials
- Analysis tools for atomic structures
- Analysis tools for potential parameterisation and training

You may also look at the [issue tracker](#) which includes planned features.

5.3 Development

Currently, the main developer and maintainer of the code is [Teemu Hynninen](#) (Tampere University of Technology and Aalto University, Helsinki).

Please notify me of any bugs you find by mailing to teemu.hynninen@tut.fi or contact me on Twitter, [@thynnine](#). There is also an [issue tracker](#) set up at github, where intended features and bugs are listed.

5.3.1 Fixed bugs

- Very serious MPI bug in the `build()` method of `pysic.FastNeighborList`: The final lists were not broadcast correctly to all cpus leading to some cpus not having the full neighbor lists. Due to the way interactions are evaluated, this usually allowed the simulations to start normally, but at some point the error would manifest as incorrectly calculated forces. Moreover, the erroneous behavior was not reproducible due to the way the result of the incorrect MPI broadcast depended on the relative speed of the cpus during list building. (4.6.2012, 0.4.3)
- MPI bug in Ewald energy and electronegativity calculation (`pysic.CoulombSummation`): Some energy components were evaluated on all cpus and summed together. (3.6.2012, 0.4.3)
- Core update issue: Changing the size of the supercell lead to conflict in neighbor list update in `pysic.FastNeighborList` since the list update was tried before the cell was updated in the core. This was no problem for the ASE neighbor list, but the new list building method requires that the Fortran core matches the atomic system in Python. (31.5.2012)
- Performance issue in Ewald force calculation (`pysic.CoulombSummation.calculate_ewald_forces()`): The force calculation routine saved in memory a huge array of structure factor gradients which were only used once and not calculated in parallel. This was a huge waste of memory and a performance bottleneck. (30.5.2012, 0.4.2)

- Index error in FastNeighborList offsets (`pysic.FastNeighborList`). Also, the offsets passed to Python were in Fortran array order, i.e., transposed. (29.5.2012, 0.4.2)
- Issue with bond order cutoff overwriting longer potential cutoffs: When neighbors are determined, a list of maximum cutoffs (cutoff for each atom) are needed (`pysic.calculator.Pysic.get_individual_cutoffs()`). In the determination, the cutoffs of potentials and bond order factors, as well as the Ewald real space cutoff are considered. There was an issue where the bond order factor if present was always used as the maximum even if the other were longer. (24.5.2012, 0.4.2)
- Index error in the 3-body evaluation loop leading to wrong forces. (24.5.2012, 0.4.2)
- Issues with how wrapping was done in the FastNeighborList search (`pysic.FastNeighborList`): The neighbor finding used only wrapped atomic coordinates to locate neighbors. This could lead to atoms crossing a periodic boundary and thus abruptly changing their wrapped position which was not seen by the neighbor offset until the list was refreshed. Changed this so that the list works in terms of absolute coordinate offsets, which do not change discontinuously. This is how the ASE list also works, and the proper way to handle the periodicity in the neighbor list. (18.05.2012, 0.4.2)
- Errorneous values for the Tersoff bond order factor for specific parameter values (`pysic.Potential, Tersoff bond order factor`): There were special cases of parameters that allowed for 0^0 (or $0**0$) and similar ill-defined expressions in the differentiation of the bond order factor, even if the gradient did exist. Added handling for some such cases. Note though that there are still parameter values that must not be used. For instance, if $\eta < 0$, if the sum of local bond terms is zero, one ends up evaluating $\frac{1}{0}$, which is not defined either. (18.05.2012, 0.4.2)

5.3.2 Known issues

PYTHON MODULE INDEX

p

- `pysic`, [61](#)
- `pysic.utility.convenience`, [67](#)
- `pysic.utility.geometry`, [68](#)
- `pysic.utility.plot`, [64](#)
- `pysic_fortran`, [69](#)
- `pysic_fortran.pysic_interface`, [69](#)

INDEX

A

`absolute_coordinates()` (built-in function), 131
`accepts_parameters()` (`psyc.BondOrderParameters` method), 48
`accepts_target_list()` (`psyc.BondOrderParameters` method), 48
`accepts_target_list()` (`psyc.Potential` method), 31
`add_bond_order_factor()` (built-in function), 70
`add_bond_order_parameters()` (`psyc.Coordinator` method), 41
`add_indices()` (`psyc.Potential` method), 31
`add_potential()` (built-in function), 70
`add_potential()` (`psyc.calculator.Pysic` method), 20
`add_symbols()` (`psyc.BondOrderParameters` method), 49
`add_symbols()` (`psyc.Potential` method), 32
`add_tags()` (`psyc.Potential` method), 32
`all_atoms` (built-in variable), 137
`all_loads` (built-in variable), 137
`allocate_bond_order_factors()` (built-in function), 71
`allocate_bond_order_storage()` (built-in function), 71
`allocate_potentials()` (built-in function), 71
`assign_bond_order_factor_indices()` (built-in function), 131
`assign_neighbor_list()` (built-in function), 131
`assign_potential_indices()` (built-in function), 132
`atom` (built-in variable), 128
`atom_buffer` (built-in variable), 137
`atoms` (built-in variable), 81
`atoms_created` (built-in variable), 81
`atoms_ready()` (`psyc.CoreMirror` method), 58

B

`balance_loads()` (built-in function), 139
`bond_descriptors_created` (built-in variable), 98
`bond_factors` (built-in variable), 81
`bond_factors_allocated` (built-in variable), 81
`bond_order_descriptor` (built-in variable), 101
`bond_order_descriptors` (built-in variable), 98
`bond_order_factor_affects_atom()` (built-in function), 104
`bond_order_factor_is_in_group()` (built-in function), 104

`bond_order_parameters` (built-in variable), 101
`bond_storage_allocated` (built-in variable), 81
`BondOrderParameters` (class in `psyc`), 48
`build()` (`psyc.FastNeighborList` method), 57

C

`c_scale_index` (built-in variable), 98
`calculate_bond_order_factors()` (built-in function), 72
`calculate_bond_order_factors()` (`psyc.Coordinator` method), 41
`calculate_bond_order_gradients()` (built-in function), 72
`calculate_bond_order_gradients_of_factor()` (built-in function), 72
`calculate_derived_parameters_bond_bending()` (built-in function), 104
`calculate_derived_parameters_charge_exp()` (built-in function), 104
`calculate_derived_parameters_dihedral()` (built-in function), 104
`calculate_electronegativities()` (built-in function), 73
`calculate_electronegativities()` (`psyc.calculator.Pysic` method), 20
`calculate_energy()` (built-in function), 73
`calculate_energy()` (`psyc.calculator.Pysic` method), 20
`calculate_ewald_electronegativities()` (built-in function), 105
`calculate_ewald_energy()` (built-in function), 105
`calculate_ewald_forces()` (built-in function), 107
`calculate_forces()` (built-in function), 73
`calculate_forces()` (`psyc.calculator.Pysic` method), 21
`calculate_stress()` (`psyc.calculator.Pysic` method), 21
`calculation_required()` (`psyc.calculator.Pysic` method), 21
`cell` (built-in variable), 81
`Cell` (class in `psyc.utility.geometry`), 68
`cell_ready()` (`psyc.CoreMirror` method), 58
`charge_relaxation()` (`psyc.ChargeRelaxation` method), 53
`ChargeRelaxation` (class in `psyc`), 53
`charges_ready()` (`psyc.CoreMirror` method), 58
`clear_bond_order_factor_characterizers()` (built-in function), 108
`clear_potential_characterizers()` (built-in function), 108

- `close_loadmonitor()` (built-in function), 139
- `coordination_index` (built-in variable), 98
- `Coordinator` (class in `pysic`), 41
- `core` (`pysic.calculator.Pysic` attribute), 21
- `core_add_bond_order_factor()` (built-in function), 83
- `core_add_potential()` (built-in function), 84
- `core_allocate_bond_order_factors()` (built-in function), 84
- `core_allocate_bond_order_storage()` (built-in function), 84
- `core_allocate_potentials()` (built-in function), 85
- `core_assign_bond_order_factor_indices()` (built-in function), 85
- `core_assign_potential_indices()` (built-in function), 85
- `core_build_neighbor_lists()` (built-in function), 85
- `core_calculate_bond_order_factors()` (built-in function), 85
- `core_calculate_bond_order_gradients()` (built-in function), 86
- `core_calculate_bond_order_gradients_of_factor()` (built-in function), 86
- `core_calculate_electronegativities()` (built-in function), 87
- `core_calculate_energy()` (built-in function), 87
- `core_calculate_forces()` (built-in function), 87
- `core_clear_atoms()` (built-in function), 88
- `core_clear_bond_order_factors()` (built-in function), 88
- `core_clear_bond_order_storage()` (built-in function), 88
- `core_clear_potentials()` (built-in function), 88
- `core_create_cell()` (built-in function), 88
- `core_create_neighbor_list()` (built-in function), 88
- `core_create_space_partitioning()` (built-in function), 89
- `core_debug_dump()` (built-in function), 89
- `core_empty_bond_order_gradient_storage()` (built-in function), 89
- `core_empty_bond_order_storage()` (built-in function), 89
- `core_evaluate_local_doublet()` (built-in function), 89
- `core_evaluate_local_quadruplet()` (built-in function), 90
- `core_evaluate_local_singlet()` (built-in function), 90
- `core_evaluate_local_triplet()` (built-in function), 90
- `core_fill_bond_order_storage()` (built-in function), 91
- `core_generate_atoms()` (built-in function), 91
- `core_get_bond_order_factor_of_atom()` (built-in function), 92
- `core_get_bond_order_factors()` (built-in function), 92
- `core_get_bond_order_gradients()` (built-in function), 92
- `core_get_bond_order_sums()` (built-in function), 92
- `core_get_cell_vectors()` (built-in function), 93
- `core_get_ewald_energy()` (built-in function), 93
- `core_get_neighbor_list_of_atom()` (built-in function), 93
- `core_get_number_of_atoms()` (built-in function), 93
- `core_get_number_of_neighbors()` (built-in function), 93
- `core_initialization_is_forced()` (`pysic.calculator.Pysic` method), 21
- `core_loop_over_local_interactions()` (built-in function), 93
- `core_post_process_bond_order_factors()` (built-in function), 94
- `core_post_process_bond_order_gradients()` (built-in function), 94
- `core_post_process_bond_order_gradients_of_factor()` (built-in function), 95
- `core_release_all_memory()` (built-in function), 96
- `core_set_ewald_parameters()` (built-in function), 96
- `core_update_atom_charges()` (built-in function), 96
- `core_update_atom_coordinates()` (built-in function), 96
- `CoreMirror` (class in `pysic`), 58
- `coulomb_summation_ready()` (`pysic.CoreMirror` method), 58
- `CoulombSummation` (class in `pysic`), 37
- `cpu_id` (built-in variable), 137
- `create_atoms()` (built-in function), 73
- `create_bond_order_factor()` (built-in function), 108
- `create_bond_order_factor_characterizer_coordination()` (built-in function), 109
- `create_bond_order_factor_characterizer_power()` (built-in function), 109
- `create_bond_order_factor_characterizer_scaler_1()` (built-in function), 109
- `create_bond_order_factor_characterizer_scaler_sqrt()` (built-in function), 109
- `create_bond_order_factor_characterizer_tersoff()` (built-in function), 109
- `create_bond_order_factor_characterizer_triplet()` (built-in function), 109
- `create_bond_order_factor_list()` (built-in function), 73
- `create_cell()` (built-in function), 74
- `create_neighbor_list()` (built-in function), 74
- `create_neighbor_lists()` (`pysic.calculator.Pysic` method), 21
- `create_potential()` (built-in function), 109
- `create_potential_characterizer_bond_bending()` (built-in function), 110
- `create_potential_characterizer_buckingham()` (built-in function), 110
- `create_potential_characterizer_charge_exp()` (built-in function), 110
- `create_potential_characterizer_constant_force()` (built-in function), 111
- `create_potential_characterizer_constant_potential()` (built-in function), 111
- `create_potential_characterizer_dihedral()` (built-in function), 111
- `create_potential_characterizer_LJ()` (built-in function), 110
- `create_potential_characterizer_power()` (built-in function), 111
- `create_potential_characterizer_spring()` (built-in function), 111

tion), 111
 create_potential_list() (built-in function), 74
 cross() (built-in function), 141

D

describe() (pysic.Potential method), 32
 description_of_bond_order_factor() (built-in function), 74
 description_of_potential() (built-in function), 74
 description_of_potential() (in module pysic), 63
 descriptions_of_parameters() (in module pysic), 63
 descriptions_of_parameters_of_bond_order_factor() (built-in function), 75
 descriptions_of_parameters_of_potential() (built-in function), 75
 descriptors_created (built-in variable), 98
 distribute_mpi() (built-in function), 75
 divide_cell() (built-in function), 132
 dot() (built-in function), 141

E

electronegativity_evaluation_index (built-in variable), 81
 energy_evaluation_index (built-in variable), 81
 evaluate_bond_order_factor() (built-in function), 111
 evaluate_bond_order_factor_coordination() (built-in function), 112
 evaluate_bond_order_factor_power() (built-in function), 112
 evaluate_bond_order_factor_tersoff() (built-in function), 112
 evaluate_bond_order_factor_triplet() (built-in function), 112
 evaluate_bond_order_gradient() (built-in function), 113
 evaluate_bond_order_gradient_coordination() (built-in function), 113
 evaluate_bond_order_gradient_power() (built-in function), 113
 evaluate_bond_order_gradient_tersoff() (built-in function), 114
 evaluate_bond_order_gradient_triplet() (built-in function), 114
 evaluate_electronegativity() (built-in function), 114
 evaluate_electronegativity_charge_exp() (built-in function), 115
 evaluate_energy() (built-in function), 115
 evaluate_energy_bond_bending() (built-in function), 116
 evaluate_energy_buckingham() (built-in function), 116
 evaluate_energy_charge_exp() (built-in function), 116
 evaluate_energy_constant_force() (built-in function), 117
 evaluate_energy_constant_potential() (built-in function), 117
 evaluate_energy_dihedral() (built-in function), 117
 evaluate_energy_LJ() (built-in function), 116
 evaluate_energy_power() (built-in function), 117

evaluate_energy_spring() (built-in function), 117
 evaluate_ewald (built-in variable), 81
 evaluate_force_bond_bending() (built-in function), 118
 evaluate_force_buckingham() (built-in function), 118
 evaluate_force_charge_exp() (built-in function), 118
 evaluate_force_constant_force() (built-in function), 119
 evaluate_force_constant_potential() (built-in function), 119
 evaluate_force_dihedral() (built-in function), 119
 evaluate_force_LJ() (built-in function), 118
 evaluate_force_power() (built-in function), 119
 evaluate_force_spring() (built-in function), 120
 evaluate_forces() (built-in function), 120
 ewald_allocated (built-in variable), 81
 ewald_cutoff (built-in variable), 81
 ewald_epsilon (built-in variable), 81
 ewald_k_cutoffs (built-in variable), 81
 ewald_scaler (built-in variable), 82
 ewald_sigma (built-in variable), 82
 examine_atoms() (built-in function), 75
 examine_bond_order_factors() (built-in function), 75
 examine_cell() (built-in function), 76
 examine_potentials() (built-in function), 76
 expand_neighbor_storage() (built-in function), 96
 expand_subcell_atom_capacity() (built-in function), 133
 expand_symbols_table() (in module pysic.utility.convenience), 67

F

FastNeighborList (class in pysic), 56
 find_subcell_for_atom() (built-in function), 133
 finish_mpi() (built-in function), 76
 finish_mpi() (in module pysic), 63
 force_core_initialization() (pysic.calculator.Pysic method), 21
 force_evaluation_index (built-in variable), 82

G

generate_atoms() (built-in function), 133
 generate_neighbor_lists() (built-in function), 76
 generate_supercell() (built-in function), 133
 genrand_init() (built-in function), 144
 genrand_real1() (built-in function), 144
 genrand_real2() (built-in function), 144
 get_absolute_coordinates() (pysic.utility.geometry.Cell method), 68
 get_atoms() (pysic.calculator.Pysic method), 22
 get_atoms() (pysic.ChargeRelaxation method), 53
 get_atoms() (pysic.CoreMirror method), 58
 get_bond_descriptor() (built-in function), 120
 get_bond_order_factors() (pysic.Coordinator method), 41
 get_bond_order_gradients() (pysic.Coordinator method), 41

get_bond_order_gradients_of_factor() (pysic.Coordinator method), 41
 get_bond_order_parameters() (pysic.Coordinator method), 41
 get_bond_order_type() (pysic.BondOrderParameters method), 49
 get_calculator() (pysic.ChargeRelaxation method), 53
 get_cell_vectors() (built-in function), 76
 get_charge_relaxation() (pysic.calculator.Pysic method), 22
 get_coordinator() (pysic.Potential method), 32
 get_coulomb_summation() (pysic.calculator.Pysic method), 22
 get_cpu_id() (built-in function), 76
 get_cpu_id() (in module pysic), 64
 get_cutoff() (pysic.BondOrderParameters method), 49
 get_cutoff() (pysic.Potential method), 32
 get_cutoff_margin() (pysic.BondOrderParameters method), 49
 get_cutoff_margin() (pysic.Potential method), 32
 get_description_of_bond_order_factor() (built-in function), 120
 get_description_of_potential() (built-in function), 120
 get_descriptions_of_parameters_of_bond_order_factor() (built-in function), 121
 get_descriptions_of_parameters_of_potential() (built-in function), 121
 get_descriptor() (built-in function), 121
 get_different_indices() (pysic.Potential method), 32
 get_different_symbols() (pysic.BondOrderParameters method), 49
 get_different_symbols() (pysic.Potential method), 32
 get_different_tags() (pysic.Potential method), 32
 get_distance() (pysic.utility.geometry.Cell method), 68
 get_electronegativities() (pysic.calculator.Pysic method), 22
 get_electronegativity_differences() (pysic.calculator.Pysic method), 22
 get_ewald_energy() (built-in function), 76
 get_forces() (pysic.calculator.Pysic method), 22
 get_group_index() (pysic.Coordinator method), 41
 get_index_of_bond_order_factor() (built-in function), 121
 get_index_of_parameter_of_bond_order_factor() (built-in function), 121
 get_index_of_parameter_of_potential() (built-in function), 122
 get_index_of_potential() (built-in function), 122
 get_indices() (pysic.Potential method), 32
 get_individual_cutoffs() (pysic.calculator.Pysic method), 22
 get_mpi_list_of_atoms() (built-in function), 76
 get_names_of_parameters_of_bond_order_factor() (built-in function), 122
 get_names_of_parameters_of_potential() (built-in function), 122
 get_neighbor_list_of_atom() (built-in function), 77
 get_neighbor_lists() (pysic.calculator.Pysic method), 22
 get_number_of_atoms() (built-in function), 77
 get_number_of_bond_order_factors() (built-in function), 122
 get_number_of_cpus() (built-in function), 77
 get_number_of_cpus() (in module pysic), 64
 get_number_of_neighbors_of_atom() (built-in function), 77
 get_number_of_parameters() (pysic.BondOrderParameters method), 49
 get_number_of_parameters_of_bond_order_factor() (built-in function), 122
 get_number_of_parameters_of_potential() (built-in function), 122
 get_number_of_potentials() (built-in function), 123
 get_number_of_targets() (pysic.BondOrderParameters method), 49
 get_number_of_targets() (pysic.Potential method), 32
 get_number_of_targets_of_bond_order_factor() (built-in function), 123
 get_number_of_targets_of_bond_order_factor_index() (built-in function), 123
 get_number_of_targets_of_potential() (built-in function), 123
 get_number_of_targets_of_potential_index() (built-in function), 123
 get_numerical_bond_order_gradient() (pysic.calculator.Pysic method), 22
 get_numerical_electronegativity() (pysic.calculator.Pysic method), 22
 get_numerical_energy_gradient() (pysic.calculator.Pysic method), 22
 get_optimal_splitting() (built-in function), 133
 get_parameter_names() (pysic.BondOrderParameters method), 49
 get_parameter_names() (pysic.Potential method), 32
 get_parameter_value() (pysic.BondOrderParameters method), 49
 get_parameter_value() (pysic.Potential method), 32
 get_parameter_values() (pysic.BondOrderParameters method), 49
 get_parameter_values() (pysic.Potential method), 32
 get_parameters() (pysic.ChargeRelaxation method), 53
 get_parameters() (pysic.CoulombSummation method), 37
 get_parameters_as_list() (pysic.BondOrderParameters method), 49
 get_potential_energy() (pysic.calculator.Pysic method), 22
 get_potential_type() (pysic.Potential method), 32
 get_potentials() (pysic.calculator.Pysic method), 23
 get_realspace_cutoff() (pysic.CoulombSummation

method), 37
 get_relative_coordinates() (pysic.utility.geometry.Cell method), 69
 get_relaxation() (pysic.ChargeRelaxation method), 54
 get_scaling_factors() (pysic.CoulombSummation method), 37
 get_separation() (pysic.utility.geometry.Cell method), 69
 get_soft_cutoff() (pysic.BondOrderParameters method), 49
 get_soft_cutoff() (pysic.Potential method), 32
 get_stress() (pysic.calculator.Pysic method), 23
 get_summation() (pysic.CoulombSummation method), 37
 get_symbols() (pysic.BondOrderParameters method), 49
 get_symbols() (pysic.Potential method), 32
 get_tags() (pysic.Potential method), 33
 get_wrapped_coordinates() (pysic.utility.geometry.Cell method), 69
 group_index_save_slot (built-in variable), 82

I

index_of_parameter() (in module pysic), 63
 initialize_bond_order_factor_characterizers() (built-in function), 123
 initialize_fortran_core() (pysic.calculator.Pysic method), 23
 initialize_load() (built-in function), 139
 initialize_parameters() (pysic.ChargeRelaxation method), 54
 initialize_parameters() (pysic.CoulombSummation method), 37
 initialize_potential_characterizers() (built-in function), 123
 int2str() (built-in function), 136
 interactions (built-in variable), 82
 InvalidCoordinatorError, 61
 InvalidParametersError, 61
 InvalidPotentialError, 61
 is_bond_order_factor() (built-in function), 77
 is_bond_order_factor() (in module pysic), 62
 is_charge_relaxation() (in module pysic), 62
 is_my_atom (built-in variable), 137
 is_potential() (built-in function), 77
 is_potential() (in module pysic), 62
 is_valid_bond_order_factor() (built-in function), 123
 is_valid_bond_order_factor() (in module pysic), 62
 is_valid_charge_relaxation() (in module pysic), 62
 is_valid_potential() (built-in function), 124
 is_valid_potential() (in module pysic), 62

L

label_length (built-in variable), 128
 list_atoms() (built-in function), 96
 list_bond_order_factors() (built-in function), 124

list_bond_order_factors() (in module pysic), 62
 list_bonds() (built-in function), 96
 list_cell() (built-in function), 96
 list_interactions() (built-in function), 96
 list_potentials() (built-in function), 124
 list_potentials() (in module pysic), 61
 list_valid_bond_order_factors() (built-in function), 77
 list_valid_bond_order_factors() (in module pysic), 62
 list_valid_potentials() (built-in function), 77
 list_valid_potentials() (in module pysic), 61
 load_length (built-in variable), 138
 loadout (built-in variable), 138
 loads_mask (built-in variable), 138
 LockedCoreError, 61

M

MissingAtomsError, 61
 MissingNeighborsError, 61
 mono_const_index (built-in variable), 98
 mono_none_index (built-in variable), 99
 mpi_atoms_allocated (built-in variable), 138
 mpi_distribute() (built-in function), 139
 mpi_finish() (built-in function), 139
 mpi_initialize() (built-in function), 139
 mpi_master_bcast_int() (built-in function), 139
 mpi_stack() (built-in function), 139
 mpi_sync() (built-in function), 140
 mpi_wall_clock() (built-in function), 140
 mpistat (built-in variable), 138
 mpistatus (built-in variable), 138
 my_atoms (built-in variable), 138
 my_load (built-in variable), 138

N

n_bond_factors (built-in variable), 82
 n_bond_order_types (built-in variable), 99
 n_cpus (built-in variable), 138
 n_interactions (built-in variable), 82
 n_max_params (built-in variable), 99
 n_potential_types (built-in variable), 99
 n_saved_bond_order_factors (built-in variable), 82
 names_of_parameters() (in module pysic), 63
 names_of_parameters_of_bond_order_factor() (built-in function), 78
 names_of_parameters_of_potential() (built-in function), 78
 neighbor_list (built-in variable), 129
 neighbor_lists_expanded() (pysic.calculator.Pysic method), 23
 neighbor_lists_ready() (pysic.CoreMirror method), 58
 no_name (built-in variable), 99
 norm_quaternion() (built-in function), 141
 norm_tolerance (built-in variable), 141
 number_of_bond_order_factors() (built-in function), 78

number_of_parameters() (in module pysic), 62
 number_of_parameters_of_bond_order_factor() (built-in function), 78
 number_of_parameters_of_potential() (built-in function), 78
 number_of_potentials() (built-in function), 79
 number_of_targets() (in module pysic), 62
 number_of_targets_of_bond_order_factor() (built-in function), 79
 number_of_targets_of_potential() (built-in function), 79

O

open_loadmonitor() (built-in function), 140

P

pad_string() (built-in function), 136
 pair_buck_index (built-in variable), 99
 pair_exp_index (built-in variable), 99
 pair_lj_index (built-in variable), 99
 pair_power_index (built-in variable), 99
 pair_spring_index (built-in variable), 99
 param_name_length (built-in variable), 99
 param_note_length (built-in variable), 100
 pi (built-in variable), 136
 pick() (built-in function), 135
 plot_abs_force_on_line() (in module pysic.utility.plot), 64
 plot_abs_force_on_plane() (in module pysic.utility.plot), 64
 plot_energy_on_line() (in module pysic.utility.plot), 65
 plot_energy_on_plane() (in module pysic.utility.plot), 65
 plot_force_component_on_plane() (in module pysic.utility.plot), 66
 plot_tangent_force_on_line() (in module pysic.utility.plot), 66
 plot_tangent_force_on_plane() (in module pysic.utility.plot), 67
 post_process_bond_order_factor() (built-in function), 124
 post_process_bond_order_factor_scaler_1() (built-in function), 124
 post_process_bond_order_factor_scaler_sqrt() (built-in function), 125
 post_process_bond_order_factor_tersoff() (built-in function), 125
 post_process_bond_order_gradient() (built-in function), 125
 post_process_bond_order_gradient_scaler_1() (built-in function), 126
 post_process_bond_order_gradient_scaler_sqrt() (built-in function), 126
 post_process_bond_order_gradient_tersoff() (built-in function), 126
 pot_name_length (built-in variable), 100

pot_note_length (built-in variable), 100
 potential (built-in variable), 102
 Potential (class in pysic), 31
 potential_affects_atom() (built-in function), 126
 potential_descriptor (built-in variable), 103
 potential_descriptors (built-in variable), 100
 potentials_allocated (built-in variable), 82
 potentials_ready() (pysic.CoreMirror method), 59
 power_index (built-in variable), 100
 Pysic (class in pysic.calculator), 20
 pysic (module), 61
 pysic.utility.convenience (module), 67
 pysic.utility.geometry (module), 68
 pysic.utility.plot (module), 64
 pysic_fortran (module), 69
 pysic_fortran.pysic_interface (module), 69

Q

q2angle() (built-in function), 141
 q2axis() (built-in function), 142
 q2matrix() (built-in function), 142
 qconj() (built-in function), 142
 qdiv() (built-in function), 142
 qinv() (built-in function), 142
 qminus() (built-in function), 142
 qnorm() (built-in function), 142
 qplus() (built-in function), 142
 qprod() (built-in function), 142
 qtimes() (built-in function), 143
 qtimesB() (built-in function), 143
 qtrn (built-in variable), 141
 quad_dihedral_index (built-in variable), 100

R

record_load() (built-in function), 140
 relative_coordinates() (built-in function), 134
 relaxation_modes (pysic.ChargeRelaxation attribute), 54
 relaxation_parameter_descriptions (pysic.ChargeRelaxation attribute), 54
 relaxation_parameters (pysic.ChargeRelaxation attribute), 54
 release() (built-in function), 79
 rot2q() (built-in function), 143
 rotate_a() (built-in function), 143
 rotate_au() (built-in function), 143
 rotate_q() (built-in function), 143

S

saved_bond_order_factors (built-in variable), 82
 saved_bond_order_gradients (built-in variable), 82
 saved_bond_order_sums (built-in variable), 82
 saved_bond_order_virials (built-in variable), 82
 separation_vector() (built-in function), 134

set_atomic_momenta() (pysic.CoreMirror method), 59
 set_atomic_positions() (pysic.CoreMirror method), 59
 set_atoms() (pysic.calculator.Pysic method), 23
 set_atoms() (pysic.ChargeRelaxation method), 54
 set_atoms() (pysic.CoreMirror method), 59
 set_bond_order_parameters() (pysic.Coordinator method), 41
 set_calculator() (pysic.ChargeRelaxation method), 54
 set_cell() (pysic.CoreMirror method), 59
 set_charge_relaxation() (pysic.calculator.Pysic method), 24
 set_charges() (pysic.CoreMirror method), 59
 set_coordinator() (pysic.Potential method), 33
 set_core() (pysic.calculator.Pysic method), 24
 set_coulomb() (pysic.CoreMirror method), 59
 set_coulomb_summation() (pysic.calculator.Pysic method), 24
 set_cutoff() (pysic.BondOrderParameters method), 49
 set_cutoff() (pysic.Potential method), 33
 set_cutoff_margin() (pysic.BondOrderParameters method), 49
 set_cutoff_margin() (pysic.Potential method), 33
 set_cutoffs() (pysic.calculator.Pysic method), 24
 set_ewald_parameters() (built-in function), 79
 set_group_index() (pysic.Coordinator method), 42
 set_indices() (pysic.Potential method), 33
 set_neighbor_lists() (pysic.CoreMirror method), 59
 set_parameter_value() (pysic.BondOrderParameters method), 50
 set_parameter_value() (pysic.ChargeRelaxation method), 54
 set_parameter_value() (pysic.CoulombSummation method), 37
 set_parameter_value() (pysic.Potential method), 33
 set_parameter_values() (pysic.BondOrderParameters method), 50
 set_parameter_values() (pysic.ChargeRelaxation method), 55
 set_parameter_values() (pysic.CoulombSummation method), 37
 set_parameter_values() (pysic.Potential method), 33
 set_parameters() (pysic.BondOrderParameters method), 50
 set_parameters() (pysic.ChargeRelaxation method), 55
 set_parameters() (pysic.CoulombSummation method), 37
 set_parameters() (pysic.Potential method), 33
 set_potentials() (pysic.calculator.Pysic method), 24
 set_potentials() (pysic.CoreMirror method), 59
 set_relaxation() (pysic.ChargeRelaxation method), 55
 set_scaling_factors() (pysic.CoulombSummation method), 37
 set_soft_cutoff() (pysic.BondOrderParameters method), 50
 set_soft_cutoff() (pysic.Potential method), 33
 set_summation() (pysic.CoulombSummation method), 38
 set_symbols() (pysic.BondOrderParameters method), 50
 set_symbols() (pysic.Potential method), 34
 set_tags() (pysic.Potential method), 34
 smoothening_derivative() (built-in function), 127
 smoothening_factor() (built-in function), 127
 smoothening_gradient() (built-in function), 127
 sqrt_scale_index (built-in variable), 100
 start_bond_order_factors() (built-in function), 79
 start_mpi() (built-in function), 79
 start_potentials() (built-in function), 80
 start_rng() (built-in function), 80
 start_timer() (built-in function), 140
 stopwatch (built-in variable), 138
 str2int() (built-in function), 137
 subcell (built-in variable), 129
 summation_modes (pysic.CoulombSummation attribute), 38
 summation_parameter_descriptions (pysic.CoulombSummation attribute), 38
 summation_parameters (pysic.CoulombSummation attribute), 38
 supercell (built-in variable), 130
 sync_mpi() (built-in function), 80

T

tersoff_index (built-in variable), 100
 timer() (built-in function), 140
 track_loads (built-in variable), 138
 tri_bend_index (built-in variable), 100
 triplet_index (built-in variable), 100

U

update_atom_charges() (built-in function), 80
 update_atom_coordinates() (built-in function), 80
 update_atomic_charges() (built-in function), 134
 update_atomic_positions() (built-in function), 134
 update_core_charges() (pysic.calculator.Pysic method), 24
 update_core_coordinates() (pysic.calculator.Pysic method), 24
 update_core_coulomb() (pysic.calculator.Pysic method), 25
 update_core_neighbor_lists() (pysic.calculator.Pysic method), 25
 update_core_potential_lists() (pysic.calculator.Pysic method), 25
 update_core_potentials() (pysic.calculator.Pysic method), 25
 update_core_supercell() (pysic.calculator.Pysic method), 25
 use_saved_bond_order_factors (built-in variable), 83
 use_saved_bond_order_gradients (built-in variable), 83

V

`vec2q()` (built-in function), [143](#)

`view_fortran()` (`pysic.CoreMirror` method), [60](#)

`vnorm()` (built-in function), [144](#)

W

`wrapped_coordinates()` (built-in function), [135](#)

`write_loadmonitor()` (built-in function), [140](#)