pyscal Documentation

Release 2.7.0

Sarath Menon, Grisell Díaz Leines, Jutta Rogal

Aug 11, 2020

Contents

1 Highlights **3**

2 Getting started **5**
2.1 Installation 5

3 Download **9**
3.1 Downloads 9

4 Examples **11**
4.1 Examples 11

5 Methods implemented in pyscal **35**
5.1 Methods implemented in pyscal 35

6 News and updates **43**
6.1 News and updates 43

7 pyscal reference **45**
7.1 pyscal reference 45

8 Support, contributing and extending **67**
8.1 Support, contributing and extending 67
8.2 Help and support 68
8.3 Common issues 68

9 Credits **69**
9.1 Citing the code 69
9.2 Developers 69
9.3 Contributers 69
9.4 Acknowledgements 69

10 License **71**
10.1 `pyscal` License 71

Python Module Index **75**

Index **77**

pyscal is a python module for the calculation of local atomic structural environments including Steinhardt's bond orientational order parameters[1] during post-processing of atomistic simulation data. The core functionality of pyscal is written in C++ with python wrappers using pybind11 which allows for fast calculations and easy extensions in python.

Steinhardt's order parameters are widely used for the identification of crystal structures[3]. They are also used to distinguish if an atom is in a solid or liquid environment[4]. pyscal is inspired by the BondOrderAnalysis code, but has since incorporated many additional features and modifications. The pyscal module includes the following functionalities:

[1] Steinhardt, P. J., Nelson, D. R., & Ronchetti, M. (1983). Physical Review B, 28.
[3] Mickel, W., Kapfer, S. C., Schröder-Turk, G. E., & Mecke, K. (2013). The Journal of Chemical Physics, 138.
[4] Auer, S., & Frenkel, D. (2005). Advances in Polymer Science, 173.

CHAPTER 1

Highlights

- calculation of Steinhardt's order parameters and their averaged version[2].
- links with the Voro++ code, for the calculation of Steinhardt parameters weighted using the face areas of Voronoi polyhedra[3].
- classification of atoms as solid or liquid[4].
- clustering of particles based on a user defined property.
- methods for calculating radial distribution functions, Voronoi volumes of particles, number of vertices and face area of Voronoi polyhedra, and coordination numbers.
- calculation of angular parameters to identify diamond structure[5].

[2] Lechner, W., & Dellago, C. (2008). The Journal of Chemical Physics, 129.
[5] Uttormark, M. J., Thompson, M. O., Clancy, P. (1993). Physical Review B, 47.

CHAPTER 2

Getting started

2.1 Installation

2.1.1 Trying pyscal

You can try some examples provided with pyscal using Binder without installing the package. Please use this link to try the package.

2.1.2 Supported operating systems

pyscal can be installed on both Linux and Mac OS based systems. For Windows systems, we recommend using Bash on Windows. Please check this tutorial on how to set up Bash on Windows.

2.1.3 Installation using conda

pyscal can be installed directly using Conda from the conda-forge channel by the following statement-

```
conda install -c conda-forge pyscal
```

This is the recommended way to install if you have an Anaconda distribution.

The above command installs the latest release version of pyscal.

pyscal is no longer maintained for Python 2.

2.1.4 Quick installation

pyscal can be installed using the following steps-

- Download an archive of the pyscal library from here.
- Extract the downloaded version. From the extracted folder, run, `python setup.py install --user`

Tip: Pyscal can be installed system-wide using `python setup.py install`.

2.1.5 Installation from the repository

pyscal can be built from the repository by-

```
git clone https://github.com/srmnitc/pyscal.git
cd pyscal
python setup.py install --user
```

2.1.6 Using a conda environment

pyscal can also be installed in a conda environment, making it easier to manage dependencies. A python3 Conda environment can be created by,

```
conda create -n myenv python=3
```

Once created, the environment can be activated using,

```
conda activate myenv
```

In case Cmake and C++11 are not available, these can be installed using,

```
(myenv) conda install -c anaconda gcc
(myenv) conda install -c anaconda cmake
```

Now the pyscal repository can be cloned and the module can be installed. Python dependencies are installed automatically.

```
(myenv) git clone https://github.com/srmnitc/pyscal.git
(myenv) cd pyscal
(myenv) python setup.py install
```

Tip: A good guide on managing Conda environments is available here.

2.1.7 Dependencies

Dependencies for the C++ part

- Cmake
- C++ 11

Dependencies for the python part

- numpy

Optional dependencies

- pytest
- matplotlib

2.1.8 Tests

In order to see if the installation worked, the following commands can be tried-

```
import pyscal.core as pc
pc.test()
```

The above code does some minimal tests and gives a value of `True` if pyscal was installed successfully. However, pyscal also contains automated tests which use the pytest python library, which can be installed by `pip install pytest`. The tests can be run by executing the command `pytest tests/` from the main code directory.

It is good idea to run the tests to check if everything is installed properly.

CHAPTER 3

Download

3.1 Downloads

The source code is available in latest stable or release versions. We recommend using the latest stable version for all updated features.

3.1.1 Source code

- latest stable version of pyscal (tar.gz)
- release version (zip)

3.1.2 Documentation

- PDF version
- Epub version

3.1.3 Publication

- Publication
- citation

CHAPTER 4

Examples

4.1 Examples

Different examples illustrating the functionality of this module is provided in this section. Interactive versions of these examples can be launched here.

Examples are designed to run on jupyter notebooks. A good tutorial on jupyter notebooks can be found here. All files used in the examples can be found in `examples/` folder of the main repository. A copy of the repository can be downloaded from here.

4.1.1 Using pyscal

Getting started with pyscal

This example illustrates basic functionality of pyscal python library by setting up a system and the atoms.

```
import pyscal.core as pc
import numpy as np
```

The `System` class

`System` is the basic class of pyscal and is required to be setup in

order to perform any calculations. It can be set up as-

```
sys = pc.System()
```

`sys` is a *`System`* object. But at this point, it is completely empty. We have to provide the system with the following information-

- the simulation box dimensions

- the positions of individual atoms.

Let us try to set up a small system, which is the bcc unitcell of lattice constant 1. The simulation box dimensions of such a unit cell would be [[0.0, 1.0], [0.0, 1.0], [0.0, 1.0]] where the first set correspond to the x axis, second to y axis and so on. The unitcell has 2 atoms and their positions are [0,0,0] and [0.5, 0.5, 0.5].

```
sys.box = [[0.0, 1.0], [0.0, 1.0], [0.0, 1.0]]
```

We can easily check if everything worked by getting the box dimensions

```
sys.box
```

```
[[0.0, 1.0], [0.0, 1.0], [0.0, 1.0]]
```

The `Atom` class

The next part is assigning the atoms. This can be done using the *`Atom`* class. Here, we will only look at the basic properties of *`Atom`* class. For a more detailed description, check the documentation. Now let us create two atoms.

```
atom1 = pc.Atom()
atom2 = pc.Atom()
```

Now two empty atom objects are created. The basic poperties of an atom are its positions and id. There are various other properties which can be set here. A detailed description can be found here.

```
atom1.pos = [0., 0., 0.]
atom1.id = 0
atom2.pos = [0.5, 0.5, 0.5]
atom2.id = 1
```

Alternatively, atom objects can also be set up as

```
atom1 = pc.Atom(pos=[0., 0., 0.], id=0)
atom2 = pc.Atom(pos=[0.5, 0.5, 0.5], id=1)
```

We can check the details of the atom by querying it

```
atom1.pos
```

```
[0.0, 0.0, 0.0]
```

Combining `System` and `Atom`

Now that we have created the atoms, we can assign them to the system. We can also assign the same box we created before.

```
sys = pc.System()
sys.atoms = [atom1, atom2]
sys.box = [[0.0, 1.0], [0.0, 1.0], [0.0, 1.0]]
```

That sets up the system completely. It has both of it's constituents - atoms and the simulation box. We can check if everything works correctly.

```
sys.atoms
```

```
[<pyscal.catom.Atom at 0x7fb343025830>, <pyscal.catom.Atom at 0x7fb343025b30>]
```

This returns all the atoms of the system. Alternatively a single atom can be accessed by,

```
atom = sys.get_atom(1)
```

The above call will fetch the atom at position 1 in the list of all atoms in the system.

Once you have all the atoms, you can modify any one and add it back to the list of all atoms in the system. The following statement will set the type of the first atom to 2.

```
atom = sys.atoms[0]
atom.type = 2
```

Lets verify if it was done properly

```
atom.type
```

```
2
```

Now we can push the atom back to the system with the new type

```
sys.set_atom(atom)
```

Warning: Due to Atom being a completely C++ class, it is necessary to use `get_atom()` and `set_atom()` to access individual atoms and set them back into the system object after modification. A list of all atoms however can be accessed directly by `atoms`

Reading in an input file

We are all set! The `System` is ready for calculations. However, in most realistic simulation situations, we have many atoms and it can be difficult to set each of them individually. In this situation we can read in input file directly. An example input file containing 500 atoms in a simulation box can be read in automatically. The file we use for this example is a file of the lammps-dump format. pyscal can also read in POSCAR files. In principle, pyscal only needs the atom positions and simulation box size, so you can write a python function to process the input file, extract the details and pass to pyscal.

```
sys = pc.System()
sys.read_inputfile('conf.dump')
```

Once again, lets check if the box dimensions are read in correctly

```
sys.box
```

```
[[-7.66608, 11.1901], [-7.66915, 11.1931], [-7.74357, 11.2676]]
```

Now we can get all atoms that belong to this system

```
len(sys.atoms)
```

```
500
```

We can see that all the atoms are read in correctly and there are 500 atoms in total. Once again, individual atom properties can be accessed as before.

```
sys.atoms[0].pos
```

```
[-5.66782, -6.06781, -6.58151]
```

Thats it! Now we are ready for some calculations. You can find more in the examples section of the documentation.

4.1.2 Finding nearest neighbors

Find neighbors of an atom and calculating coordination numbers

In this example, we will read in a configuration from an MD simulation and then calculate the coordination number distribution. This example assumes that you read the basic example.

```
import pyscal.core as pc
import numpy as np
import matplotlib.pyplot as plt
```

Read in a file

The first step is setting up a system. We can create atoms and simulation box using the `crystal_structures module`. Let us start by importing the module.

```
import pyscal.crystal_structures as pcs
```

```
atoms, box = pcs.make_crystal('bcc', lattice_constant= 4.00, repetitions=[6,6,6])
```

The above function creates an bcc crystal of 6x6x6 unit cells with a lattice constant of 4.00 along with a simulation box that encloses the particles. We can then create a *`System`* and assign the atoms and box to it.

```
sys = pc.System()
sys.atoms = atoms
sys.box = box
```

Calculating neighbors

We start by calculating the neighbors of each atom in the system. There are two ways to do this, using a `cutoff` method and using a `voronoi` polyhedra method. We will try with both of them. First we try with cutoff system - which has three sub options. We will check each of them in detail.

Cutoff method

Cutoff method takes cutoff distance value and finds all atoms within the cutoff distance of the host atom.

```
sys.find_neighbors(method='cutoff', cutoff=4.1)
```

Now lets get all the atoms.

```
atoms = sys.atoms
```

let us try accessing the coordination number of an atom

```
atoms[0].coordination
```

```
14
```

As we would expect for a bcc type lattice, we see that the atom has 14 neighbors (8 in the first shell and 6 in the second). Lets try a more interesting example by reading in a bcc system with thermal vibrations. Thermal vibrations lead to distortion in atomic positions, and hence there will be a distribution of coordination numbers.

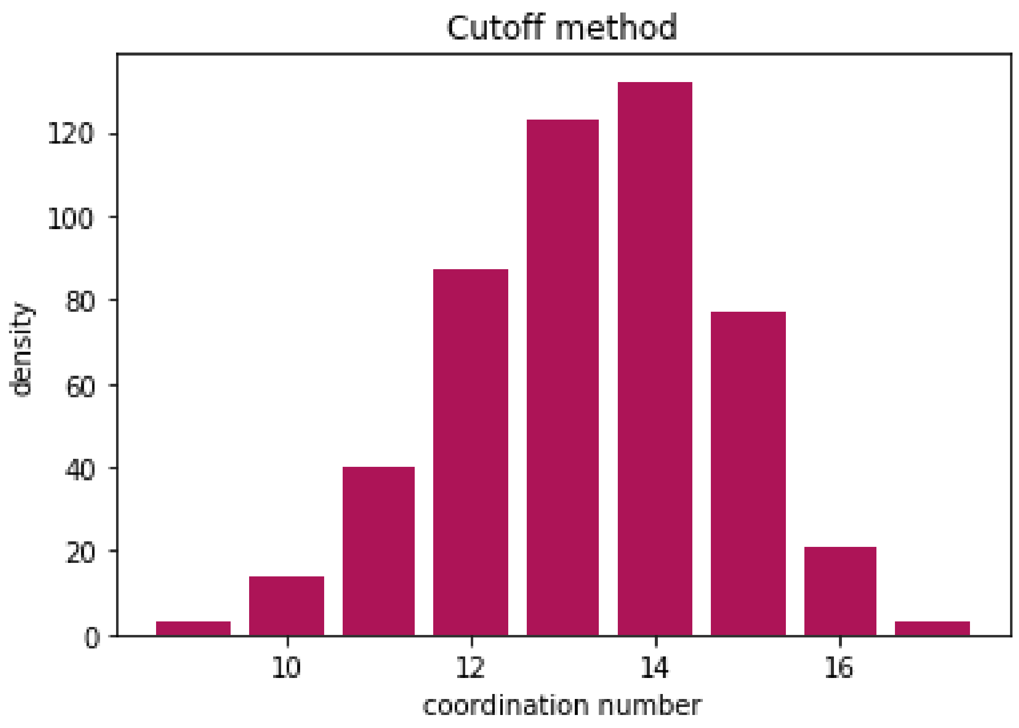
```
sys = pc.System()
sys.read_inputfile('conf.dump')
sys.find_neighbors(method='cutoff', cutoff=3.6)
atoms = sys.atoms
```

We can loop over all atoms and create a histogram of the results

```
coord = [atom.coordination for atom in atoms]
```

Now lets plot and see the results

```
nos, counts = np.unique(coord, return_counts=True)
plt.bar(nos, counts, color="#AD1457")
plt.ylabel("density")
plt.xlabel("coordination number")
plt.title("Cutoff method")
```



Adaptive cutoff methods

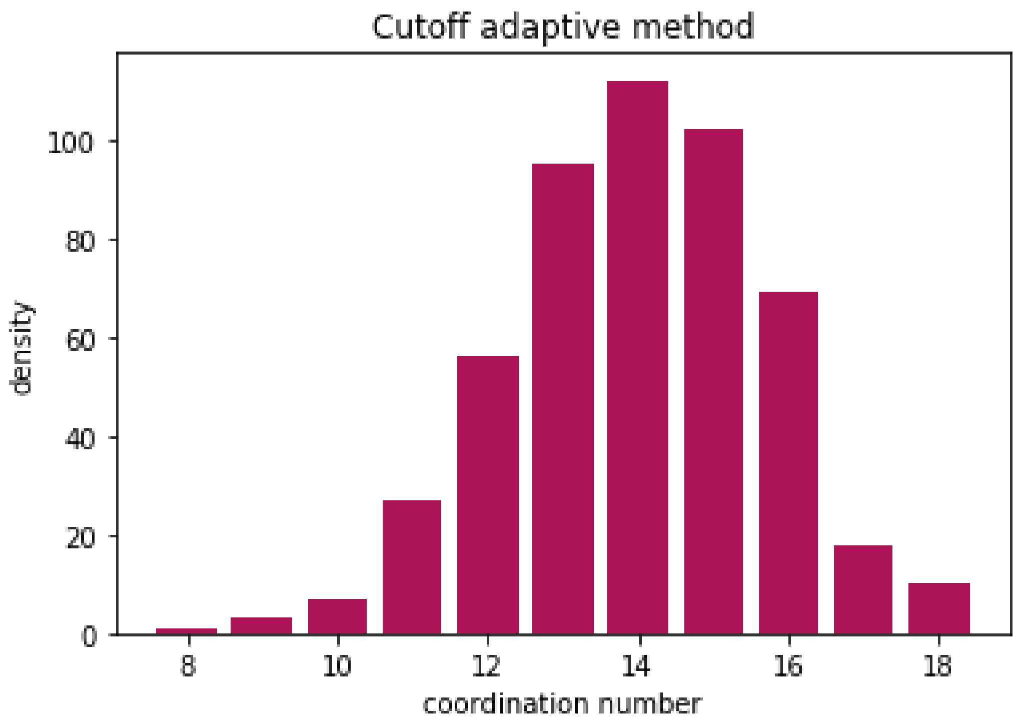
pyscal also has adaptive cutoff methods implemented. These methods remove the restriction on having a global cutoff. A distinct cutoff is selected for each atom during runtime. pyscal uses two distinct algorithms to do this - `sann` and `adaptive`. Please check the documentation for a explanation of these algorithms. For the purpose of this example, we will use the `adaptive` algorithm.

adaptive algorithm

```
sys.find_neighbors(method='cutoff', cutoff='adaptive', padding=1.5)
atoms = sys.atoms
coord = [atom.coordination for atom in atoms]
```

Now let us plot

```
nos, counts = np.unique(coord, return_counts=True)
plt.bar(nos, counts, color="#AD1457")
plt.ylabel("density")
plt.xlabel("coordination number")
plt.title("Cutoff adaptive method")
```



The adaptive method also gives similar results!

Voronoi method

Voronoi method calculates the voronoi polyhedra of all atoms. Any atom that shares a voronoi face area with the host atom are considered neighbors. Voronoi polyhedra is calculated using the Voro++ code. However, you do not need to install this specifically as it is linked to pyscal.

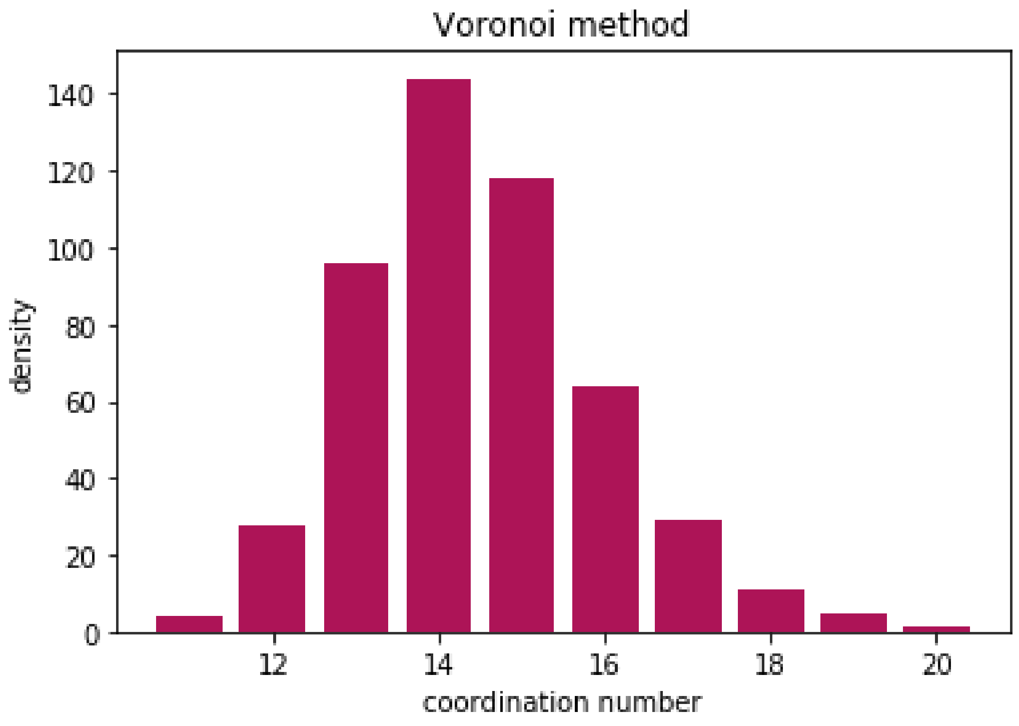
```
sys.find_neighbors(method='voronoi')
```

Once again, let us get all atoms and find their coordination

```
atoms = sys.atoms
coord = [atom.coordination for atom in atoms]
```

And visualise the results

```
nos, counts = np.unique(coord, return_counts=True)
plt.bar(nos, counts, color="#AD1457")
plt.ylabel("density")
plt.xlabel("coordination number")
plt.title("Voronoi method")
```



Finally..

All methods find the coordination number, and the results are comparable. Cutoff method is very sensitive to the choice of cutoff radius, but Voronoi method can slightly overestimate the neighbors due to thermal vibrations.

4.1.3 Steinhardt parameters

Calculating bond orientational order parameters

This example illustrates the calculation of bond orientational order parameters. Bond order parameters, q_l and their averaged versions, $\bar{q}_l$ are widely used to identify atoms belong to different crystal structures. In this example, we will consider MD snapshots for bcc, fcc, hcp and liquid, and calculate the q_4 and q_6 parameters and their averaged versions which are widely used in literature. More details can be found here.

```
import pyscal.core as pc
import pyscal.crystal_structures as pcs
import numpy as np
import matplotlib.pyplot as plt
```

In this example, we analyse MD configurations, first a set of perfect bcc, fcc and hcp structures and another set with thermal vibrations.

Perfect structures

To create atoms and box for perfect structures, the :mod:~`pyscal.crystal_structures` module is used. The created atoms and boxes are then assigned to `System` objects.

```
bcc_atoms, bcc_box = pcs.make_crystal('bcc', lattice_constant=3.147, repetitions=[4,4,
↪4])
bcc = pc.System()
bcc.atoms = bcc_atoms
bcc.box = bcc_box
```

```
fcc_atoms, fcc_box = pcs.make_crystal('fcc', lattice_constant=3.147, repetitions=[4,4,
↪4])
fcc = pc.System()
fcc.atoms = fcc_atoms
fcc.box = fcc_box
```

```
hcp_atoms, hcp_box = pcs.make_crystal('hcp', lattice_constant=3.147, repetitions=[4,4,
↪4])
hcp = pc.System()
hcp.atoms = hcp_atoms
hcp.box = hcp_box
```

Next step is calculation of nearest neighbors. There are two ways to calculate neighbors, by using a cutoff distance or by using the voronoi cells. In this example, we will use the cutoff method and provide a cutoff distance for each structure.

Finding the cutoff distance

The cutoff distance is normally calculated in a such a way that the atoms within the first shell is incorporated in this distance. The :func:`pyscal.core.System.calculate_rdf` function can be used to find this cutoff distance.

```
bccrdf = bcc.calculate_rdf()
fccrdf = fcc.calculate_rdf()
hcprdf = hcp.calculate_rdf()
```

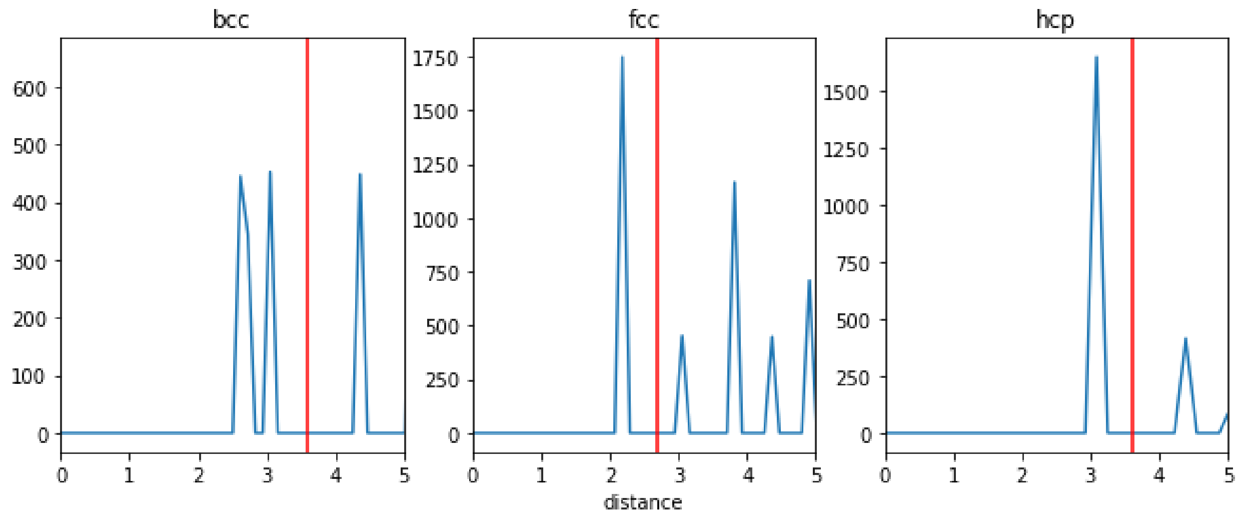
Now the calculated rdf is plotted

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(11,4))
ax1.plot(bccrdf[1], bccrdf[0])
ax2.plot(fccrdf[1], fccrdf[0])
ax3.plot(hcprdf[1], hcprdf[0])
ax1.set_xlim(0,5)
ax2.set_xlim(0,5)
ax3.set_xlim(0,5)
```

(continues on next page)

(continued from previous page)

```
ax1.set_title('bcc')
ax2.set_title('fcc')
ax3.set_title('hcp')
ax2.set_xlabel("distance")
ax1.axvline(3.6, color='red')
ax2.axvline(2.7, color='red')
ax3.axvline(3.6, color='red')
```



The selected cutoff distances are marked in red in the above plot. For bcc, since the first two shells are close to each other, for this example, we will take the cutoff in such a way that both shells are included.

Steinhardt's parameters - cutoff neighbor method

```
bcc.find_neighbors(method='cutoff', cutoff=3.6)
fcc.find_neighbors(method='cutoff', cutoff=2.7)
hcp.find_neighbors(method='cutoff', cutoff=3.6)
```

We have used a cutoff of 3 here, but this is a parameter that has to be tuned. Using a different cutoff for each structure is possible, but it would complicate the method if the system has a mix of structures. Now we can calculate the q_4 and q_6 distributions

```
bcc.calculate_q([4,6])
fcc.calculate_q([4,6])
hcp.calculate_q([4,6])
```

Thats it! Now lets gather the results and plot them.

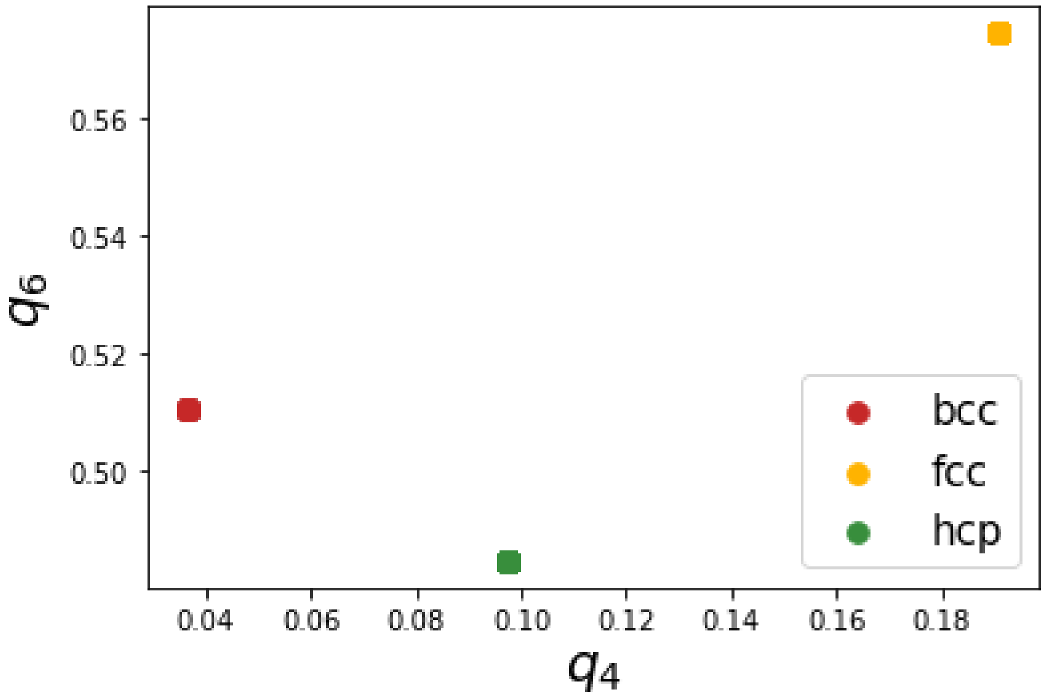
```
bccq = bcc.get_qvals([4, 6])
fccq = fcc.get_qvals([4, 6])
hcpq = hcp.get_qvals([4, 6])
```

```
plt.scatter(bccq[0], bccq[1], s=60, label='bcc', color='#C62828')
plt.scatter(fccq[0], fccq[1], s=60, label='fcc', color='#FFB300')
plt.scatter(hcpq[0], hcpq[1], s=60, label='hcp', color='#388E3C')
plt.xlabel("$q_4$", fontsize=20)
```

(continues on next page)

(continued from previous page)

```
plt.ylabel("$q_6$", fontsize=20)
plt.legend(loc=4, fontsize=15)
```



Firstly, we can see that Steinhardt parameter values of all the atoms fall on one specific point which is due to the absence of thermal vibrations. Next, all the points are well separated and show good distinction. However, at finite temperatures, the atomic positions are affected by thermal vibrations and hence show a spread in the distribution. We will show the effect of thermal vibrations in the next example.

Structures with thermal vibrations

Once again, we create the reqd structures using the :mod:~`pyscal.crystal_structures` module. Noise can be applied to atomic positions using the `noise` keyword as shown below.

```
bcc_atoms, bcc_box = pcs.make_crystal('bcc', lattice_constant=3.147, repetitions=[10,
↪10,10], noise=0.1)
bcc = pc.System()
bcc.atoms = bcc_atoms
bcc.box = bcc_box
```

```
fcc_atoms, fcc_box = pcs.make_crystal('fcc', lattice_constant=3.147, repetitions=[10,
↪10,10], noise=0.1)
fcc = pc.System()
fcc.atoms = fcc_atoms
fcc.box = fcc_box
```

```
hcp_atoms, hcp_box = pcs.make_crystal('hcp', lattice_constant=3.147, repetitions=[10,
↪10,10], noise=0.1)
hcp = pc.System()
```

(continues on next page)

(continued from previous page)

```
hcp.atoms = hcp_atoms
hcp.box = hcp_box
```

cutoff method

```
bcc.find_neighbors(method='cutoff', cutoff=3.6)
fcc.find_neighbors(method='cutoff', cutoff=2.7)
hcp.find_neighbors(method='cutoff', cutoff=3.6)
```

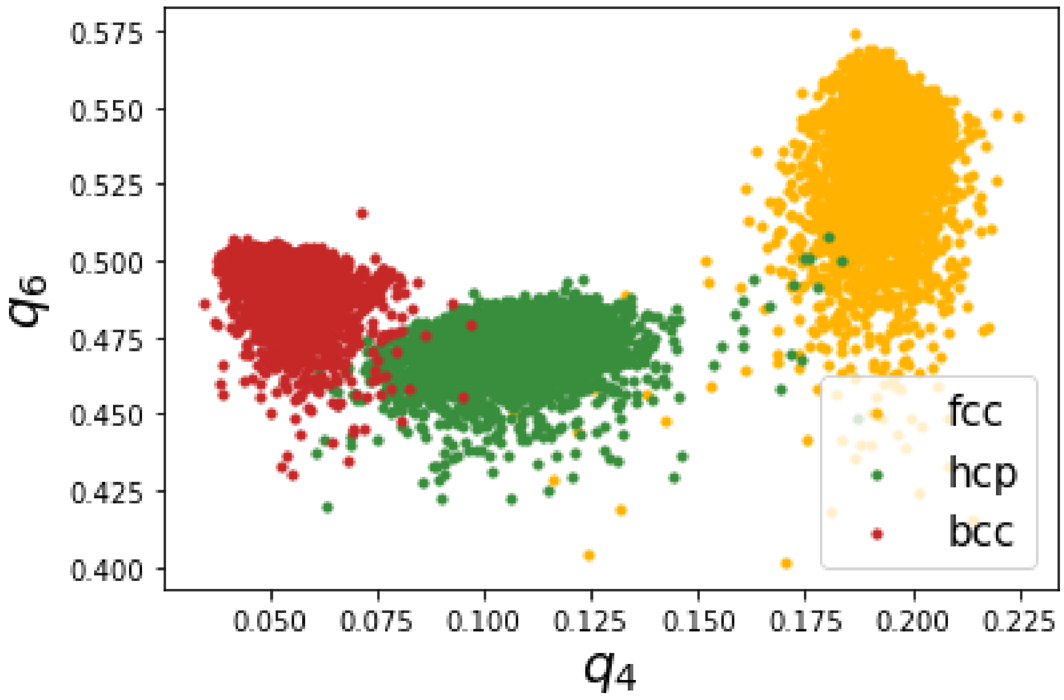
And now, calculate $q_4$, $q_6$ parameters

```
bcc.calculate_q([4,6])
fcc.calculate_q([4,6])
hcp.calculate_q([4,6])
```

Gather the q vales and plot them

```
bccq = bcc.get_qvals([4, 6])
fccq = fcc.get_qvals([4, 6])
hcpq = hcp.get_qvals([4, 6])
```

```
plt.scatter(fccq[0], fccq[1], s=10, label='fcc', color='#FFB300')
plt.scatter(hcpq[0], hcpq[1], s=10, label='hcp', color='#388E3C')
plt.scatter(bccq[0], bccq[1], s=10, label='bcc', color='#C62828')
plt.xlabel("$q_4$", fontsize=20)
plt.ylabel("$q_6$", fontsize=20)
plt.legend(loc=4, fontsize=15)
```



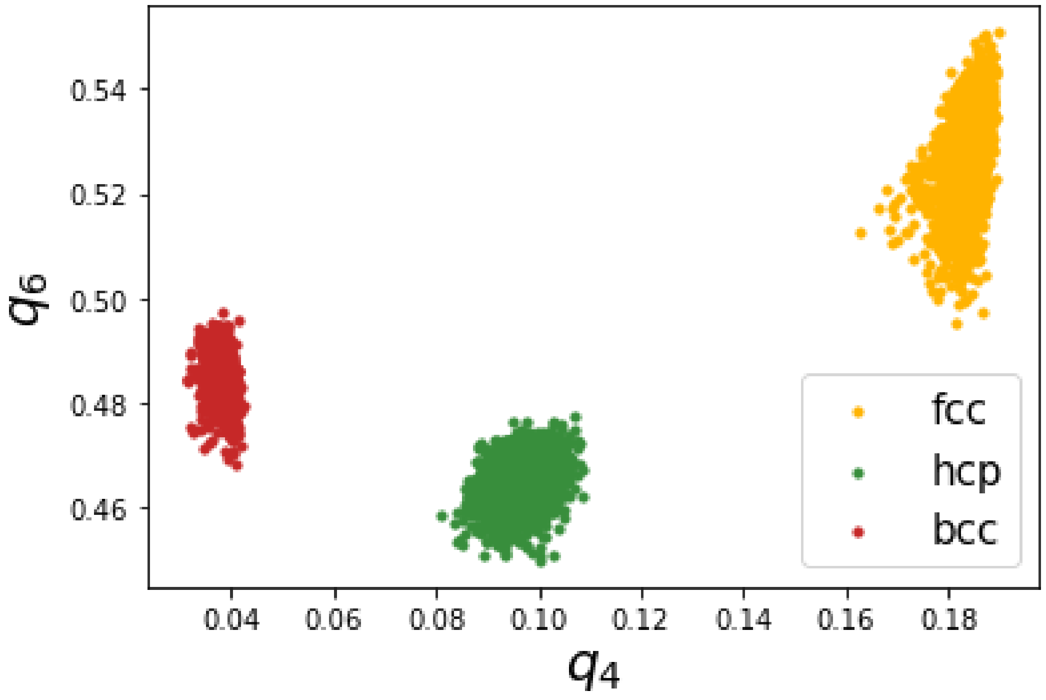
The thermal vibrations cause the distributions to spread, but it still very good. Lechner and Dellago proposed using the averaged distributions, $\bar{q}_4 - \bar{q}_6$ to better distinguish the distributions. Lets try that.

```
bcc.calculate_q([4,6], averaged=True)
fcc.calculate_q([4,6], averaged=True)
hcp.calculate_q([4,6], averaged=True)
```

```
bccaq = bcc.get_qvals([4, 6], averaged=True)
fccaq = fcc.get_qvals([4, 6], averaged=True)
hcpaq = hcp.get_qvals([4, 6], averaged=True)
```

Lets see if these distributions are better..

```
plt.scatter(fccaq[0], fccaq[1], s=10, label='fcc', color='#FFB300')
plt.scatter(hcpaq[0], hcpaq[1], s=10, label='hcp', color='#388E3C')
plt.scatter(bccaq[0], bccaq[1], s=10, label='bcc', color='#C62828')
plt.xlabel("$q_4$", fontsize=20)
plt.ylabel("$q_6$", fontsize=20)
plt.legend(loc=4, fontsize=15)
```



This looks much better! We can see that the resolution is much better than the non averaged versions.

There is also the possibility to calculate structures using Voronoi based neighbor identification too. Let's try that now.

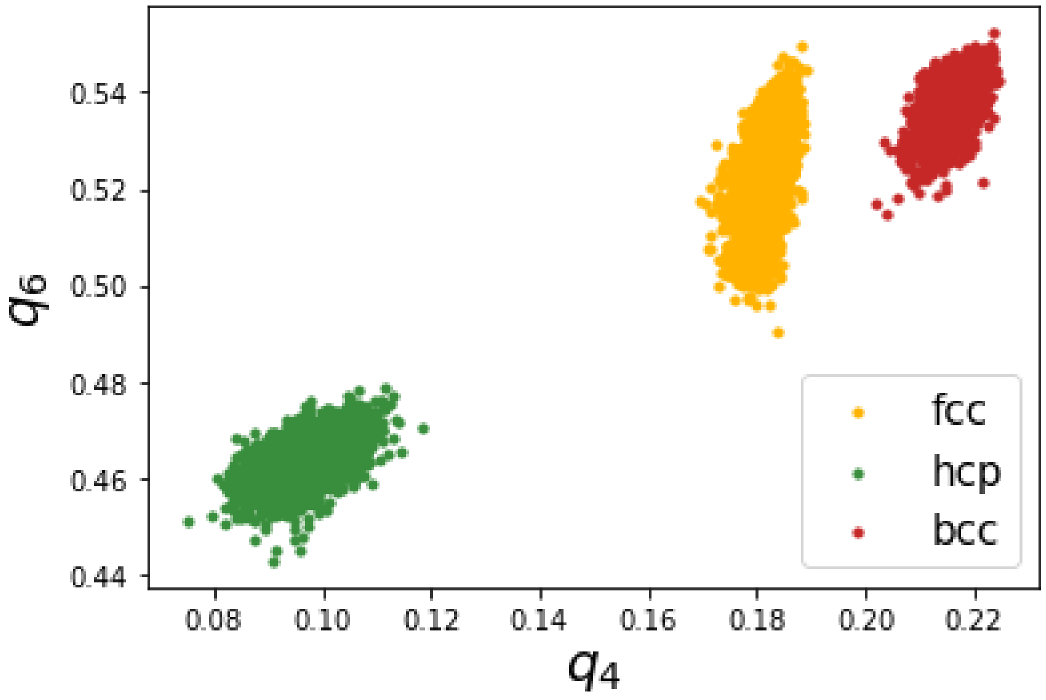
```
bcc.find_neighbors(method='voronoi')
fcc.find_neighbors(method='voronoi')
hcp.find_neighbors(method='voronoi')
```

```
bcc.calculate_q([4,6], averaged=True)
fcc.calculate_q([4,6], averaged=True)
hcp.calculate_q([4,6], averaged=True)
```

```
bccaq = bcc.get_qvals([4, 6], averaged=True)
fccaq = fcc.get_qvals([4, 6], averaged=True)
hcpaq = hcp.get_qvals([4, 6], averaged=True)
```

Plot the calculated points..

```
plt.scatter(fccaq[0], fccaq[1], s=10, label='fcc', color='#FFB300')
plt.scatter(hcpaq[0], hcpaq[1], s=10, label='hcp', color='#388E3C')
plt.scatter(bccaq[0], bccaq[1], s=10, label='bcc', color='#C62828')
plt.xlabel("$q_4$", fontsize=20)
plt.ylabel("$q_6$", fontsize=20)
plt.legend(loc=4, fontsize=15)
```



Voronoi based method also provides good resolution,the major difference being that the location of bcc distribution is different.

Disorder variable

In this example, disorder variable which was introduced to measure the disorder of a system is explored. We start by importing the necessary modules. We will use *crystal_structures* to create the necessary crystal structures.

```
import pyscal.core as pc
import pyscal.crystal_structures as pcs
import matplotlib.pyplot as plt
import numpy as np
```

First an fcc structure with a lattice constant of 4.00 is created.

```
fcc_atoms, fcc_box = pcs.make_crystal('fcc', lattice_constant=4, repetitions=[4,4,4])
```

The created atoms and box are assigned to a *System* object.

```
fcc = pc.System()
fcc.atoms = fcc_atoms
fcc.box = fcc_box
```

The next step is find the neighbors, and the calculate the Steinhardt parameter based on which we could calculate the disorder variable.

```
fcc.find_neighbors(method='cutoff', cutoff='adaptive')
```

Once the neighbors are found, we can calculate the Steinhardt parameter value. In this example $q = 6$ will be used.

```
fcc.calculate_q(6)
```

Finally, disorder parameter can be calculated.

```
fcc.calculate_disorder()
```

The calculated disorder value can be accessed for each atom using the `disorder` variable.

```
atoms = fcc.atoms
```

```
disorder = [atom.disorder for atom in atoms]
```

```
np.mean(disorder)
```

```
-1.041556887034408e-16
```

As expected, for a perfect fcc structure, we can see that the disorder is zero. The variation of disorder variable on a distorted lattice can be explored now. We will once again use the `noise` keyword along with *`make_crystal()`* to create a distorted lattice.

```
fcc_atoms_d1, fcc_box_d1 = pcs.make_crystal('fcc', lattice_constant=4, repetitions=[4,
↪4,4], noise=0.01)
fcc_d1 = pc.System()
fcc_d1.atoms = fcc_atoms_d1
fcc_d1.box = fcc_box_d1
```

Once again, find neighbors and then calculate disorder

```
fcc_d1.find_neighbors(method='cutoff', cutoff='adaptive')
fcc_d1.calculate_q(6)
fcc_d1.calculate_disorder()
```

Check the value of disorder

```
atoms_d1 = fcc_d1.atoms
```

```
disorder = [atom.disorder for atom in atoms_d1]
```

```
np.mean(disorder)
```

```
0.013889967380485688
```

The value of average disorder for the system has increased with noise. Finally trying with a high amount of noise.

```
fcc_atoms_d2, fcc_box_d2 = pcs.make_crystal('fcc', lattice_constant=4, repetitions=[4,
↪4,4], noise=0.1)
fcc_d2 = pc.System()
fcc_d2.atoms = fcc_atoms_d2
fcc_d2.box = fcc_box_d2
```

```
fcc_d2.find_neighbors(method='cutoff', cutoff='adaptive')
fcc_d2.calculate_q(6)
fcc_d2.calculate_disorder()
```

```
atoms_d2 = fcc_d2.atoms
```

```
disorder = [atom.disorder for atom in atoms_d2]
np.mean(disorder)
```

```
1.8469165876016702
```

The value of disorder parameter shows an increase with the amount of lattice distortion. An averaged version of disorder parameter, averaged over the neighbors for each atom can also be calculated as shown below.

```
fcc_d2.calculate_disorder(averaged=True)
```

```
atoms_d2 = fcc_d2.atoms
disorder = [atom.avg_disorder for atom in atoms_d2]
np.mean(disorder)
```

```
1.850630088115515
```

The disorder parameter can also be calculated for values of Steinhardt parameter other than 6. For example,

```
fcc_d2.find_neighbors(method='cutoff', cutoff='adaptive')
fcc_d2.calculate_q([4, 6])
fcc_d2.calculate_disorder(q=4, averaged=True)
```

```
atoms_d2 = fcc_d2.atoms
disorder = [atom.disorder for atom in atoms_d2]
np.mean(disorder)
```

```
1.880741277448693
```

$q = 4$, for example, can be useful when measuring disorder in bcc crystals

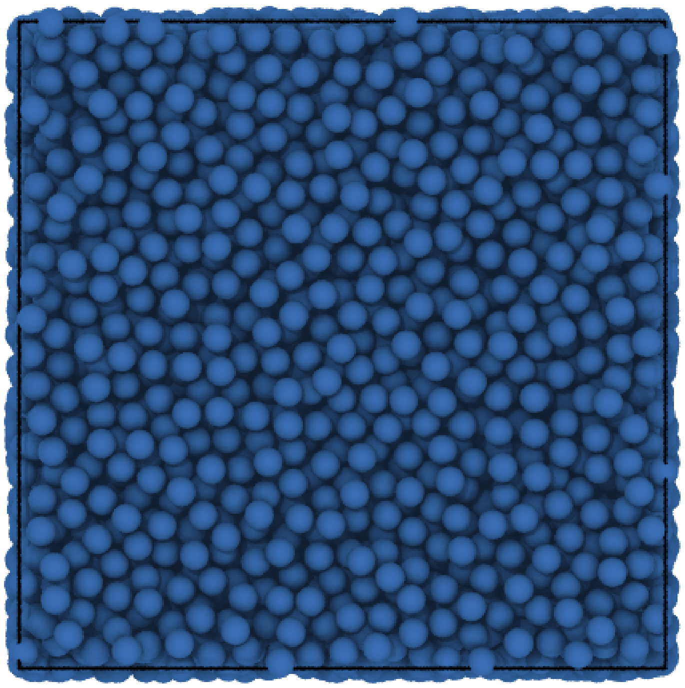
Distinction of solid liquid atoms and clustering

In this example, we will take one configuration from a molecular dynamics simulation which has a solid cluster in liquid. The task is to identify solid atoms and cluster them. More details about the method can be found here.

The first step is, of course, importing all the necessary module. For visualisation, we will use Ovito.

The above image shows a visualization of the system using Ovito. Importing modules,

```
import pyscal.core as pc
```



Now we will set up a *System* with this input file, and calculate neighbors. Here we will use a cutoff method to find neighbors. More details about finding neighbors can be found here.

```
sys = pc.System()
sys.read_inputfile('cluster.dump')
sys.find_neighbors(method='cutoff', cutoff=3.63)
```

Once we compute the neighbors, the next step is to find solid atoms. This can be done using *find_solids()* method.

```
sys.find_solids(bonds=6, threshold=0.5, avgthreshold=0.6, cluster=False)
```

The above statement found all the solid atoms. Solid atoms can be identified by the value of the *solid* attribute. For that we first get the atom objects and select those with *solid* value as True.

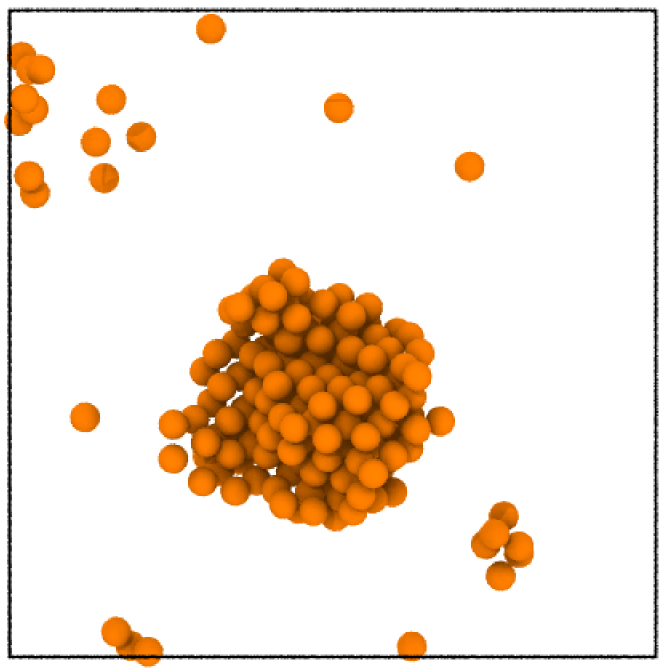
```
atoms = sys.atoms
solids = [atom for atom in atoms if atom.solid]
len(solids)
```

```
202
```

There are 202 solid atoms in the system. In order to visualise in Ovito, we need to first write it out to a trajectory file. This can be done with the help of *to_file()* method of System. This method can help to save any attribute of the atom or any Steinhardt parameter value.

```
sys.to_file('sys.solid.dat', custom = ['solid'])
```

We can now visualize this file in Ovito. After opening the file in Ovito, the modifier compute property can be selected. The `Output property` should be `selection` and in the expression field, `solid==0` can be selected to select all the non solid atoms. Applying a modifier delete selected particles can be applied to delete all the non solid particles. The system after removing all the liquid atoms is shown below.



Clustering algorithm

You can see that there is a cluster of atom. The clustering functions that pyscal offers helps to select atoms that belong to this cluster. If you used *find_clusters()* with `cluster=True`, the clustering is carried out. Since we did used `cluster=False` above, we will rerun the function

```
sys.find_solids(bonds=6, threshold=0.5, avgthreshold=0.6, cluster=True)
```

```
176
```

You can see that the above function call returned the number of atoms belonging to the largest cluster as an output. In order to extract atoms that belong to the largest cluster, we can use the *largest_cluster* attribute of the atom.

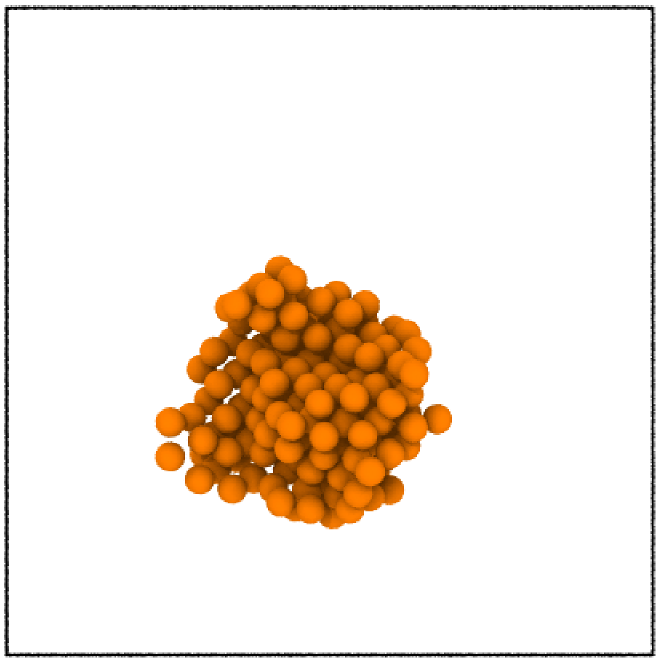
```
atoms = sys.atoms
largest_cluster = [atom for atom in atoms if atom.largest_cluster]
len(largest_cluster)
```

```
176
```

The value matches that given by the function. Once again we will save this information to a file and visualize it in Ovito.

```
sys.to_file('sys.cluster.dat', custom = ['solid', 'largest_cluster'])
```

The system visualized in Ovito following similar steps as above is shown below.



It is clear from the image that the largest cluster of solid atoms was successfully identified. Clustering can be done over any property. The following example with the same system will illustrate this.

Clustering based on a custom property

In pyscal, clustering can be done based on any property. The following example illustrates this. To find the clusters based on a custom property, the `cluster_atoms()` method has to be used. The simulation box shown above has the centre roughly at (25, 25, 25). For the custom clustering, we will cluster all atoms within a distance of 10 from the the rough centre of the box at (25, 25, 25). Let us define a function that checks the above condition.

```
def check_distance(atom):
    #get position of atom
    pos = atom.pos
    #calculate distance from (25, 25, 25)
    dist = ((pos[0]-25)**2 + (pos[1]-25)**2 + (pos[2]-25)**2)**0.5
    #check if dist < 10
    return (dist <= 10)
```

The above function would return True or False depending on a condition and takes the Atom as an argument. These are the two important conditions to be satisfied. Now we can pass this function to cluster. First, set up the system and find the neighbors.

```
sys = pc.System()
sys.read_inputfile('cluster.dump')
sys.find_neighbors(method='cutoff', cutoff=3.63)
```

Now cluster

```
sys.cluster_atoms(check_distance)
```

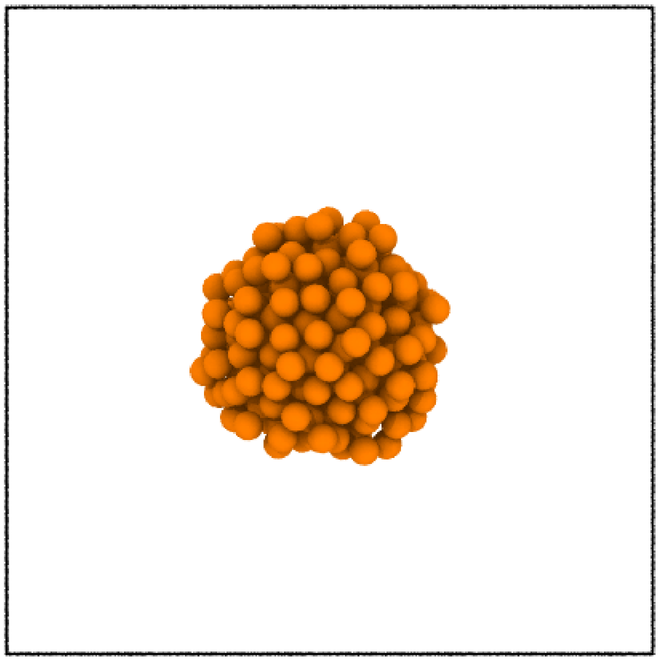
```
242
```

There are 242 atoms in the cluster! Once again we can check this, save to a file and visualize in ovito.

```
atoms = sys.atoms
largest_cluster = [atom for atom in atoms if atom.largest_cluster]
len(largest_cluster)
```

```
242
```

```
sys.to_file('sys.dist.dat', custom = ['solid', 'largest_cluster'])
```



This example illustrates that any property can be used to cluster the atoms!

4.1.4 Angle based methods

Angular parameter

This illustrates the use of angular parameter to identify diamond structure. Angular parameter was introduced by Uttormark et al., and measures the tetrahedrality of the local atomic structure. An atom belonging to diamond structure has four nearest neighbors which gives rise to six three body angles around the atom. The angular parameter A is then defined as,

$A = \sum_{i=1}^{6}(\cos(\theta_i) + \frac{1}{3})^2$

An atom belonging to diamond structure would show the value of angular params close to 0. The following example illustrates the use of this parameter.

```
import pyscal.core as pc
import pyscal.crystal_structures as pcs
import numpy as np
import matplotlib.pyplot as plt
```

Create structures

The first step is to create some structures using the *crystal_structures* module and assign it to a System. This can be done as follows-

```
atoms, box = pcs.make_crystal('diamond', lattice_constant=4, repetitions=[3,3,3])
sys = pc.System()
sys.atoms = atoms
sys.box = box
```

Now we can find the neighbors of all atoms. In this case we will use an adaptive method using the `find_neighbors()` which can find an individual cutoff for each atom.

```
sys.find_neighbors(method='cutoff', cutoff='adaptive')
```

Finally, the angular criteria can be calculated by,

```
sys.calculate_angularcriteria()
```

The above function assigns the angular value for each atom in the attribute `angular` which can be accessed using,

```
atoms = sys.atoms
angular = [atom.angular for atom in atoms]
```

The angular values are zero for atoms that belong to diamond structure.

χ parameters

χ parameters introduced by Ackland and Jones measures the angles generated by pairs of neighbor atom around the host atom, and assigns it to a histogram to calculate a local structure. In this example, we will create different crystal structures and see how the χ parameters change with respect to the local coordination.

```
import pyscal.core as pc
import pyscal.crystal_structures as pcs
import matplotlib.pyplot as plt
import numpy as np
```

The *crystal_structures* module is used to create different perfect crystal structures. The created atoms and simulation box is then assigned to a *System* object. For this example, fcc, bcc, hcp and diamond structures are created.

```
fcc_atoms, fcc_box = pcs.make_crystal('fcc', lattice_constant=4, repetitions=[4,4,4])
fcc = pc.System()
fcc.atoms = fcc_atoms
fcc.box = fcc_box
```

```
bcc_atoms, bcc_box = pcs.make_crystal('bcc', lattice_constant=4, repetitions=[4,4,4])
bcc = pc.System()
bcc.atoms = bcc_atoms
bcc.box = bcc_box
```

```
hcp_atoms, hcp_box = pcs.make_crystal('hcp', lattice_constant=4, repetitions=[4,4,4])
hcp = pc.System()
hcp.atoms = hcp_atoms
hcp.box = hcp_box
```

```
dia_atoms, dia_box = pcs.make_crystal('diamond', lattice_constant=4, repetitions=[4,4,
↪4])
dia = pc.System()
dia.atoms = dia_atoms
dia.box = dia_box
```

Before calculating χ parameters, the neighbors for each atom need to be found.

```
fcc.find_neighbors(method='cutoff', cutoff='adaptive')
bcc.find_neighbors(method='cutoff', cutoff='adaptive')
hcp.find_neighbors(method='cutoff', cutoff='adaptive')
dia.find_neighbors(method='cutoff', cutoff='adaptive')
```

Now, χ parameters can be calculated

```
fcc.calculate_chiparams()
bcc.calculate_chiparams()
hcp.calculate_chiparams()
dia.calculate_chiparams()
```

The calculated parameters for each atom can be accessed using the *chiparams* attribute.

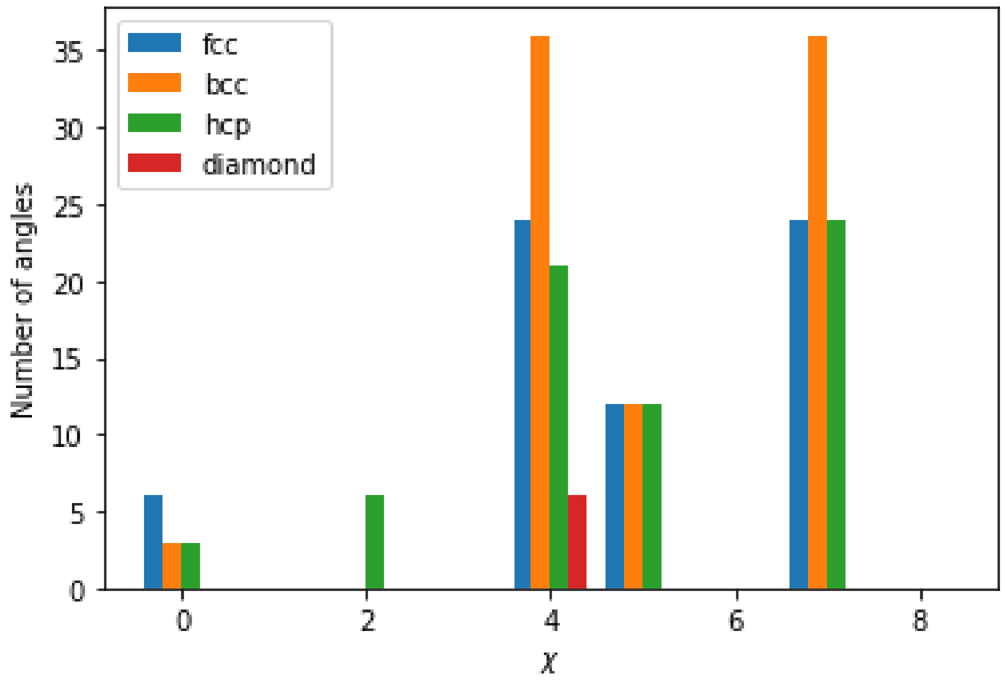
```
fcc_atoms = fcc.atoms
bcc_atoms = bcc.atoms
hcp_atoms = hcp.atoms
dia_atoms = dia.atoms
```

```
fcc_atoms[10].chiparams
```

```
[6, 0, 0, 0, 24, 12, 0, 24, 0]
```

The output is an array of length 9 which shows the number of neighbor angles found within specific bins as explained here. The output for one atom from each structure is shown below.

```
plt.bar(np.array(range(9))-0.3, fcc_atoms[10].chiparams, width=0.2, label="fcc")
plt.bar(np.array(range(9))-0.1, bcc_atoms[10].chiparams, width=0.2, label="bcc")
plt.bar(np.array(range(9))+0.1, hcp_atoms[10].chiparams, width=0.2, label="hcp")
plt.bar(np.array(range(9))+0.3, dia_atoms[10].chiparams, width=0.2, label="diamond")
plt.xlabel("$\chi$")
plt.ylabel("Number of angles")
plt.legend()
```



The atoms exhibit a distinct fingerprint for each structure. Structural identification can be made up comparing the ratio of various χ parameters as described in the original publication.

4.1.5 Voronoi tessellation

Voronoi parameters

Voronoi tessellation can be used to identify local structure by counting the number of faces of the Voronoi polyhedra of an atom. For each atom a vector $\langle n3\ n4\ n5\ n6$ can be calculated where n_3 is the number of Voronoi faces of the associated Voronoi polyhedron with three vertices, n_4 is with four vertices and so on. Each perfect crystal structure such as a signature vector, for example, bcc can be identified by $\langle 0\ 6\ 0\ 8\rangle$ and fcc can be identified using $\langle 0\ 12\ 0\ 0\rangle$. It is also a useful tool for identifying icosahedral structure which has the fingerprint $\langle 0\ 0\ 12\ 0\rangle$.

```
import pyscal.core as pc
import pyscal.crystal_structures as pcs
import matplotlib.pyplot as plt
import numpy as np
```

The `crystal_structures` module is used to create different perfect crystal structures. The created atoms and simulation box is then assigned to a `System` object. For this example, fcc, bcc, hcp and diamond structures are created.

```
fcc_atoms, fcc_box = pcs.make_crystal('fcc', lattice_constant=4, repetitions=[4,4,4])
fcc = pc.System()
fcc.atoms = fcc_atoms
fcc.box = fcc_box
```

```
bcc_atoms, bcc_box = pcs.make_crystal('bcc', lattice_constant=4, repetitions=[4,4,4])
bcc = pc.System()
bcc.atoms = bcc_atoms
bcc.box = bcc_box
```

```
hcp_atoms, hcp_box = pcs.make_crystal('hcp', lattice_constant=4, repetitions=[4,4,4])
hcp = pc.System()
hcp.atoms = hcp_atoms
hcp.box = hcp_box
```

Before calculating the Voronoi polyhedron, the neighbors for each atom need to be found using Voronoi method.

```
fcc.find_neighbors(method='voronoi')
bcc.find_neighbors(method='voronoi')
hcp.find_neighbors(method='voronoi')
```

Now, Voronoi vector can be calculated

```
fcc.calculate_vorovector()
bcc.calculate_vorovector()
hcp.calculate_vorovector()
```

The calculated parameters for each atom can be accessed using the `vorovector` attribute.

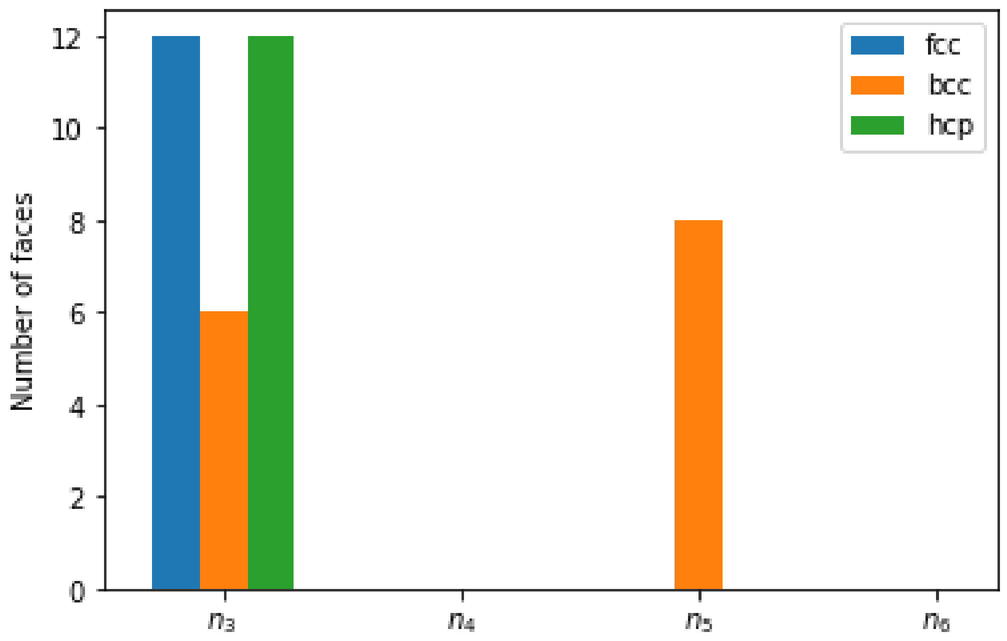
```
fcc_atoms = fcc.atoms
bcc_atoms = bcc.atoms
hcp_atoms = hcp.atoms
```

```
fcc_atoms[10].vorovector
```

```
[0, 12, 0, 0]
```

As expected, fcc structure exhibits 12 faces with four vertices each. For a single atom, the difference in the Voronoi fingerprint is shown below

```
fig, ax = plt.subplots()
ax.bar(np.array(range(4))-0.2, fcc_atoms[10].vorovector, width=0.2, label="fcc")
ax.bar(np.array(range(4)), bcc_atoms[10].vorovector, width=0.2, label="bcc")
ax.bar(np.array(range(4))+0.2, hcp_atoms[10].vorovector, width=0.2, label="hcp")
ax.set_xticks([1,2,3,4])
ax.set_xlim(0.5, 4.25)
ax.set_xticklabels(['$n_3$', '$n_4$', '$n_5$', '$n_6$'])
ax.set_ylabel("Number of faces")
ax.legend()
```



The difference in Voronoi fingerprint for bcc and the closed packed structures is clearly visible. Voronoi tessellation, however, is incapable of distinction between fcc and hcp structures.

Voronoi volume

Voronoi volume, which is the volume of the Voronoi polyhedron is calculated when the neighbors are found. The volume can be accessed using the `volume` attribute.

```
fcc_atoms = fcc.atoms
```

```
fcc_vols = [atom.volume for atom in fcc_atoms]
```

```
np.mean(fcc_vols)
```

```
16.0
```

4.1.6 Other examples

CHAPTER 5

Methods implemented in pyscal

5.1 Methods implemented in pyscal

5.1.1 Methods to calculate neighbors of a particle

pyscal includes different methods to explore the local environment of a particle that rely on the calculation of nearest neighbors. Various approaches to compute the neighbors of particles are discussed here.

Fixed cutoff method

The most common method to calculate the nearest neighbors of an atom is using a cutoff radius. The neighborhood of an atom for calculation of Steinhardt's parameters[1] is often carried out using this method. Commonly, a cutoff is selected as the first minimum of the radial distribution functions. Once a cutoff is selected, the neighbors of an atom are those that fall within this selected radius. The following code snippet will use the cutoff method to calculate neighbors. Please check the examples section for basic use of the module. In this example, `conf.dump` is assumed to be the input configuration of the system. A cutoff radius of 3 is assumed for calculation of neighbors.

```
import pyscal.core as pc
sys = pc.System()
sys.read_inputfile('conf.dump')
sys.find_neighbors(method='cutoff', cutoff=3)
```

Adaptive cutoff methods

A fixed cutoff radius can introduce limitations to explore the local environment of the particle in some cases:

- At finite temperatures, when thermal fluctuations take place, the selection of a fixed cutoff may result in an inaccurate description of the local environment.

[1] Steinhardt, PJ, Nelson, DR, Ronchetti, M. Phys Rev B 28, 1983.

- If there is more than one structure present in the system, for example, bcc and fcc, the selection of cutoff such that it includes the first shell of both structures can be difficult.

In order to achieve a more accurate description of the local environment, various adaptive approaches have been proposed. Two of the methods implemented in the module are discussed below.

Solid angle based nearest neighbor algorithm (SANN)

SANN algorithm[1] determines the cutoff radius by counting the solid angles around an atom and equating it to 4π. The algorithm solves the following equation iteratively.

$$R_i^{(m)} = \frac{\sum_{j=1}^{m} r_{i,j}}{m-2} < r_{i,m+1}$$

where i is the host atom, j are its neighbors with r_{ij} is the distance between atoms i and j. R_i is the cutoff radius for each particle i which is found by increasing the neighbor of neighbors m iteratively. For a description of the algorithm and more details, please check the reference[2]. SANN algorithm can be used to find the neighbors by,

```
import pyscal.core as pc
sys = pc.System()
sys.read_inputfile('conf.dump')
sys.find_neighbors(method='cutoff', cutoff='sann')
```

Since SANN algorithm involves sorting, a sufficiently large cutoff is used in the beginning to reduce the number entries to be sorted. This parameter is calculated by,

$$r_{initial} = \text{threshold} \times \left(\frac{\text{Simulation box volume}}{\text{Number of particles}} \right)^{\frac{1}{3}}$$

a tunable `threshold` parameter can be set through function arguments.

Adaptive cutoff method

An adaptive cutoff specific for each atom can also be found using an algorithm similar to adaptive common neighbor analysis[2]. This adaptive cutoff is calculated by first making a list of all neighbor distances for each atom similar to SANN method. Once this list is available, then the cutoff is calculated from,

$$r_{cut}(i) = \text{padding} \times \left(\frac{1}{\text{nlimit}} \sum_{j=1}^{\text{nlimit}} r_{ij} \right)$$

This method can be chosen by,

```
import pyscal.core as pc
sys = pc.System()
sys.read_inputfile('conf.dump')
sys.find_neighbors(method='cutoff', cutoff='adaptive')
```

The `padding` and `nlimit` parameters in the above equation can be tuned using the respective keywords.

Either of the adaptive method can be used to find neighbors, which can then be used to calculate Steinhardt's parameters or their averaged version.

[1] van Meel, JA, Filion, L, Valeriani, C, Frenkel, D, J Chem Phys 234107, 2012.
[2] Stukowski, A, Model Simul Mater SC 20, 2012.

Voronoi tessellation

Voronoi tessellation provides a completely parameter free geometric approach for calculation of neighbors. Voro++ code is used for Voronoi tessellation. Neighbors can be calculated using this method by,

```python
import pyscal.core as pc
sys = pc.System()
sys.read_inputfile('conf.dump')
sys.find_neighbors(method='voronoi')
```

Finding neighbors using Voronoi tessellation also calculates a weight for each neighbor. The weight of a neighbor j towards a host atom i is given by,

$$W_{ij} = \frac{A_{ij}}{\sum_{j=1}^{N} A_{ij}}$$

where A_{ij} is the area of Voronoi facet between atom i and j, N are all the neighbors identified through Voronoi tessellation. This weight can be used later for calculation of weighted Steinhardt's parameters. Optionally, it is possible to choose the exponent for this weight. Option `voroexp` is used to set this option. For example if `voroexp=2`, the weight would be calculated as,

$$W_{ij} = \frac{A_{ij}^2}{\sum_{j=1}^{N} A_{ij}^2}$$

5.1.2 Steinhardt's bond orientational order parameters

Steinhardt's parameters

Steinhardt's bond orientational order parameters[1] are a set of parameters based on spherical harmonics to explore the local atomic environment. These parameters have been used extensively for various uses such as distinction of crystal structures, identification of solid and liquid atoms and identification of defects[2].

These parameters, which are rotationally and translationally invariant are defined by,

$$q_l(i) = \left(\frac{4\pi}{2l+1} \sum_{m=-l}^{l} |q_{lm}(i)|^2 \right)^{\frac{1}{2}}$$

where,

$$q_{lm}(i) = \frac{1}{N(i)} \sum_{j=1}^{N(i)} Y_{lm}(\boldsymbol{r}_{ij})$$

in which Y_{lm} are the spherical harmonics and $N(i)$ is the number of neighbours of particle i, $\boldsymbol{r}_{ij}$ is the vector connecting particles i and j, and l and m are both intergers with $m \in [-l, +l]$. Various parameters have found specific uses, such as q_2 and q_6 for identification of crystallinity, q_6 for identification of solidity, and q_4 and q_6 for distinction of crystal structures[2]. Commonly this method uses a cutoff radius to identify the neighbors of an atom. The cutoff can be chosen based on different methods available. Once the cutoff is chosen and neighbors are calculated, the calculation of Steinhardt's parameters is straightforward.

[1] Steinhardt, PJ, Nelson, DR, Ronchetti, M. Phys Rev B 28, 1983.

[2] Mickel, W, Kapfer, SC, Schroder-Turk, GE, Mecke, K, J Chem Phys 138, 2013.

```
sys.calculate_q([4, 6])
q = sys.get_qvals([4, 6])
```

Note: Associated methods

find_neighbors() calculate_q() get_qvals() get_q() Example

Averaged Steinhardt's parameters

At high temperatures, thermal vibrations affect the atomic positions. This in turn leads to overlapping distributions of q_l parameters, which makes the identification of crystal structures difficult. To address this problem, the averaged version $\bar{q}_l$ of Steinhardt's parameters was introduced by Lechner and Dellago[1]. $\bar{q}_l$ is given by,

$$\bar{q}_l(i) = \left(\frac{4\pi}{2l+1} \sum_{m=-l}^{l} \left| \frac{1}{\tilde{N}(i)} \sum_{k=0}^{\tilde{N}(i)} q_{lm}(k) \right|^2 \right)^{\frac{1}{2}}$$

where the sum from $k = 0$ to $\tilde{N}(i)$ is over all the neighbors and the particle itself. The averaged parameters takes into account the first neighbor shell and also information from the neighboring atoms and thus reduces the overlap between the distributions. Commonly $\bar{q}_4$ and $\bar{q}_6$ are used in identification of crystal structures. Averaged versions can be calculated by setting the keyword `averaged=True` as follows.

```
sys.calculate_q([4, 6], averaged=True)
q = sys.get_qvals([4, 6], averaged=True)
```

Note: Associated methods

find_neighbors() calculate_q() get_qvals() get_q() Example

Voronoi weighted Steinhardt's parameters

In order to improve the resolution of crystal structures Mickel et al[1] proposed weighting the contribution of each neighbor to the Steinhardt parameters by the ratio of the area of the Voronoi facet shared between the neighbor and host atom. The weighted parameters are given by,

$$q_{lm}(i) = \frac{1}{N(i)} \sum_{j=1}^{N(i)} \frac{A_{ij}}{A} Y_{lm}(\boldsymbol{r}_{ij})$$

where A_{ij} is the area of the Voronoi facet between atoms i and j and A is the sum of the face areas of atom i. In pyscal, the area weights are already assigned during the neighbor calculation phase when the Voronoi method is used to calculate neighbors in the *find_neighbors()*. The Voronoi weighted Steinhardt's parameters can be calculated as follows,

```
sys.find_neighbors(method='voronoi')
sys.calculate_q([4, 6])
q = sys.get_qvals([4, 6])
```

[1] Lechner, W, Dellago, C, J Chem Phys, 2013.
[1] Mickel, W, Kapfer, SC, Schroder-Turk, GE, Mecke, K, J Chem Phys 138, 2013.

The weighted Steinhardt's parameters can also be averaged as described above. Once again, the keyword `averaged=True` can be used for this purpose.

```
sys.find_neighbors(method='voronoi')
sys.calculate_q([4, 6], averaged=True)
q = sys.get_qvals([4, 6], averaged=True)
```

It was also proposed that higher powers of the weight[2] $\frac{A_{ij}^{\alpha}}{A(\alpha)}$ where $\alpha = 2, 3$ can also be used, where $A(\alpha) = \sum_{j=1}^{N(i)} A_{ij}^{\alpha}$ The value of this can be set using the keyword `voroexp` during the neighbor calculation phase.

```
sys.find_neighbors(method='voronoi', voroexp=2)
```

If the value of `voroexp` is set to 0, the neighbors would be found using Voronoi method, but the calculated Steinhardt's parameters will not be weighted.

Note: Associated methods

find_neighbors() calculate_q() get_qvals() get_q()

Disorder parameter

Kawasaki and Onuki[1] proposed a disorder variable based on Steinhardt's order paramaters which can be used to distinguish between ordered and disordered structures.

The disorder variable for an atom is defined as,

$$D_j = \frac{1}{n_b^j} \sum_{k \in neighbors} [S_{jj} + S_{kk} - 2S_{jk}]$$

where S is given by,

$$S_{jk} = \sum_{-l \leq m \leq l} q_{lm}^j (q_{lm}^k)^*$$

l = 6 was used in the original publication as it is a good indicator of crystallinity. However, l = 4 can also be used for treating bcc structures. An averaged disorder parameter for each atom can also be calculated in pyscal,

$$\bar{D}_j = \frac{1}{n_b^j} \sum_{k \in neighbors} D_j$$

In pyscal, disorder parameter can be calculated by the following code-block,

```
import pyscal.core as pc
sys = pc.System()
sys.read_inputfile('conf.dump')
sys.find_neighbors(method='cutoff', cutoff=0)
sys.calculate_q(6)
sys.calculate_disorder(averaged=True, q=6)
```

The value of q can be replaced with whichever is required from 2-12. The calculated values can be accessed by, *disorder* and *avg_disorder*.

Note: Associated methods

[2] Haeberle, J, Sperl, M, Born, P Arxiv 2019.
[1] Kawasaki .T, Onuki .A, JCP 135, 174109 (2011)

calculate_q() calculate_disorder() disorder avg_disorder Example

Classification of atoms as solid or liquid

pyscal can also be used to distinguish solid and liquid atoms. The classification is based on Steinhardt's parameters, specifically q_6. The method defines two neighboring atoms i and j as having solid bonds if a parameter s_{ij}[1],

$$s_{ij} = \sum_{m=-6}^{6} q_{6m}(i) q_{6m}^*(j) \geq \text{threshold}$$

Additionally, a second order parameter is used to improve the distinction in solid-liquid boundaries[2]. This is defined by the criteria,

$$\langle s_{ij} \rangle > \text{avgthreshold}$$

If a particle has n number of bonds with $s_{ij} \geq \text{threshold}$ and the above condition is also satisfied, it is considered as a solid. The solid atoms can be clustered to find the largest solid cluster of atoms. Please check the examples on how to do this.

Finding solid atoms in liquid start with reading in a file and calculation of neighbors.

```
import pyscal.core as pc
sys = pc.System()
sys.read_inputfile('conf.dump')
sys.find_neighbors(method='cutoff', cutoff=4)
```

Once again, there are various methods for finding neighbors. Please check here for details on neighbor calculation methods. Once the neighbors are calculated, solid atoms can be found directly by,

```
sys.find_solids(bonds=6, threshold=0.5, avgthreshold=0.6, cluster=True)
```

`bonds` set the number of minimum bonds a particle should have (as defined above), `threshold` and `avgthreshold` are the same quantities that appear in the equations above. Setting the keyword `cluster` to True returns the size of the largest solid cluster. It is also possible to check if each atom is solid or not.

```
atoms = sys.atom
solids = [atom.solid for atom in atoms]
```

Note: Associated methods

find_neighbors() calculate_q() find_solids() solid() Example

5.1.3 Angle based methods

Angular criteria for identification of diamond structure

Angular parameter introduced by Uttormark et al[1] is used to measure the tetrahedrality of local atomic structure. An atom belonging to diamond structure has four nearest neighbors which gives rise to six three body angles around the atom. The angular parameter A is then defined as,

[1] Auer, S, Frenkel, D. Adv Polym Sci 173, 2005
[2] Bokeloh, J, Rozas, RE, Horbach, J, Wilde, G, Phys. Rev. Lett. 107, 2011
[1] Uttormark, MJ, Thompson, MO, Clancy, P, Phys. Rev. B 47, 1993

$A = \sum_{i=1}^{6} (\cos(\theta_i) + \frac{1}{3})^2$

An atom belonging to diamond structure would show the value of angular params close to 0. Angular parameter can be calculated in pyscal using the following method -

```
import pyscal.core as pc
sys = pc.System()
sys.read_inputfile('conf.dump')
sys.find_neighbors(method='cutoff', cutoff='adaptive')
sys.calculate_angularcriteria()
```

The calculated angular criteria value can be accessed for each atom using *angular*.

Note: Associated methods

calculate_angularcriteria() angular Example

χ parameters for structural identification

χ parameters introduced by Ackland and Jones[1] measures all local angles created by an atom with its neighbors and creates a histogram of these angles to produce vector which can be used to identify structures. After finding the neighbors of an atom, $\cos\theta_{ijk}$ for atoms j and k which are neighbors of i is calculated for all combinations of i, j and k. The set of all calculated cosine values are then added to a histogram with the following bins - [-1.0, -0.945, -0.915, -0.755, -0.705, -0.195, 0.195, 0.245, 0.795, 1.0]. Compared to χ parameters from χ_0 to χ_7 in the associated publication, the vector calculated in pyscal contains values from χ_0 to χ_8 which is due to an additional χ parameter which measures the number of neighbors between cosines -0.705 to -0.195. The χ vector is characteristic of the local atomic environment and can be used to identify crystal structures, details of which can be found in the publication[1].

χ parameters can be calculated in pyscal using,

```
import pyscal.core as pc
sys = pc.System()
sys.read_inputfile('conf.dump')
sys.find_neighbors(method='cutoff', cutoff='adaptive')
sys.calculate_chiparams()
```

The calculated values for each atom can be accessed using *chiparams*.

Note: Associated methods

calculate_chiparams() chiparams Example

5.1.4 Voronoi tessellation to identify local structures

Voronoi tessellation can be used for identification of local structure by counting the number of faces of the Voronoi polyhedra of an atom[1][2]. For each atom a vector $\langle n_3\ n_4\ n_5\ n_6 \rangle$ can be calculated where n_3 is the number of Voronoi faces of the associated Voronoi polyhedron with three vertices, n_4 is with four vertices and so on. Each perfect crystal structure such as a signature vector, for example, bcc can be identified by $\langle 0\ 6\ 0\ 8 \rangle$ and fcc can be identified using $\langle 0\ 12\ 0\ 0 \rangle$. It is also a useful tool for identifying icosahedral structure which has the fingerprint $\langle 0\ 0\ 12\ 0 \rangle$. In pyscal, the voronoi vector can be calculated using,

[1] Ackland, Jones, Phys. Rev. B 73, 2006
[1] Finney, J. L. PRS 319 ,1970
[2] Tanemura et al., PTP 58, 1977

```
import pyscal.core as pc
sys = pc.System()
sys.read_inputfile('conf.dump')
sys.find_neighbors(method='voronoi')
sys.calculate_vorovector()
```

The vector for each atom can be accessed using *vorovector*. Furthermore, the associated Voronoi volume of the polyhedron, which may be indicative of the local structure, is also automatically calculated when finding neighbors using *find_neighbors()*. This value for each atom can be accessed by *volume*. An averaged version of the volume, which is averaged over the neighbors of an atom can be accessed using *avg_volume*.

Note: Associated methods

find_neighbors() *calculate_vorovector()* *vorovector* *volume* *avg_volume* Example

5.1.5 Centrosymmetry parameter

Centrosymmetry parameter (CSP) was introduced by Kelchner et al. to identify defects in crystals. The parameter measures the loss of local symmetry. For an atom with N nearest neighbors, the parameter is given by,

$$\text{CSP} = \sum_{i=1}^{N/2} \left|\mathbf{r}_i + \mathbf{r}_{i+N/2}\right|^2$$

$\mathbf{r}_i$ and $\mathbf{r}_{i+N/2}$ are vectors from the central atom to two opposite pairs of neighbors. There are two main methods to identify the opposite pairs of neighbors as described in this publication. The first of the approaches is called Greedy Edge Selection(GES) and is implemented in LAMMPS and Ovito. GES algorithm calculates a weight $w_{ij} = |\mathbf{r}_i + \mathbf{r}_j|$ for all combinations of neighbors around an atom and calculates CSP over the smallest $N/2$ weights.

A centrosymmetry parameter calculation using GES algorithm can be carried out as follows-

```
import pyscal.core as pc
sys = pc.System()
sys.read_inputfile('conf.dump')
sys.find_neighbors(method='voronoi')
sys.calculate_centrosymmetry(nmax = 12)
```

`nmax` parameter specifies the number of nearest neighbors to be considered for the calculation of CSP. The second algorithm is called the Greedy Vertex Matching and is implemented in AtomEye and Atomsk. This algorithm orders the neighbors atoms in order of increasing distance from the central atom. From this list, the closest neighbor is paired with its lowest weight partner and both atoms removed from the list. This process is continued until no more atoms are remaining in the list. CSP calculation using this algorithm can be carried out by,

```
import pyscal.core as pc
sys = pc.System()
sys.read_inputfile('conf.dump')
sys.find_neighbors(method='voronoi')
sys.calculate_centrosymmetry(nmax = 12, algorithm = "gvm")
```

CHAPTER 6

News and updates

6.1 News and updates

August 15, 2020

pyscal is cited in the publication *Evolution of local atomic structure during solidification of Fe-RE (RE=La, Ce) alloy*. See the publication here.

November 21, 2019

pyscal is selected as the E-CAM module of the month. See the news here.

November 1, 2019

pyscal paper is accepted in the Journal of Open Source Software. See the paper here.

October 17, 2019

Publication for pyscal submitted to the Journal of Open Source Software. See the review here.

July 12, 2019

Version 1.0.0 of pyscal is released.

CHAPTER 7

pyscal reference

7.1 pyscal reference

7.1.1 pyscal Reference

pyscal.core module

Main module of pyscal. This module contains definitions of the two major classes in pyscal - the `System` and `Atom`. Atom is a pure pybind11 class whereas System is a hybrid class with additional python definitions. For the ease of use, Atom class should be imported from the `core` module. The original pybind11 definitions of Atom and System can be found in `catom` and `csystem` respectively.

class `pyscal.core.`**`System`**

Bases: `pyscal.csystem.System`

A python/pybind11 hybrid class for holding the properties of a system.

`box`

A list containing the dimensions of the simulation box in the format *[[x_low, x_high], [y_low, y_high], [z_low, z_high]]*

Type list of list of floats

`atoms`

Type list of `Atom` objects

Notes

A *System* consists of two major components - the simulation box and the atoms. All the associated variables are then calculated using this class.

Note: atoms can be accessed or set as `atoms`. However, due to technical reasons individual atoms should be accessed using the `get_atom()` method. An atom can be assigned to the atom using the `set_atom()` method.

Examples

```
>>> sys = System()
>>> sys.read_inputfile('atoms.dat')
```

calculate_angularcriteria()

Calculate the angular criteria for each atom :param None:

Returns

Return type None

Notes

Calculates the angular criteria for each atom as defined in [1]_. Angular criteria is useful for identification of diamond cubic structures. Angular criteria is defined by,

$$A = \sum_{i=1}^{6} (\cos(\theta_i) + \frac{1}{3})^2$$

where cos(theta) is the angle size suspended by each pair of neighbors of the central atom. A will have a value close to 0 for structures if the angles are close to 109 degrees. The calculated A parameter for each atom is stored in `angular`.

References

calculate_centrosymmetry(*nmax=12*, *calculate_neighbors=True*, *algorithm='ges'*)

Calculate the centrosymmetry parameter

Parameters

- **nmax** (*int, optional*) – number of neighbors to be considered for centrosymmetry parameters. Has to be a positive, even integer. Default 12
- **calculate_neighbors** (*bool, optional*) – if True recalculate neighbors using number method, if False neighbor calculation is not done.
- **algorithm** (*{'ges', 'gvm'}, optional*) – *ges* uses the Greedy Edge Selection algorithm, *gvm* uses Greedy Vertex Matching. Default *ges*.

Returns

Return type None

References

Notes

Calculate the centrosymmetry parameter for each atom which can be accessed by `centrosymmetry` attribute. It calculates the degree of inversion symmetry of an atomic environment. Centrosymmetry recalculates the neighbor using the number method as specified in `¬pyscal.core.System.find_neighbors()` method. This is the ensure that the required number of neighbors are found for calculation of the parameter. The calculation of neighbors through number method can be suppressed by setting `calculate_neighbors=False`.

This method uses two different algorithms, The Greedy Edge Selection (GES) [1] or the Greedy Vertex Matching (GVM) [2] as specified in [3]. GES algorithm is implemented in LAMMPS and Ovito, whereas GVM is used in AtomEye and Atomsk. Please see [3] for a detailed description of the algorithms. The algorithm can be selected using the *algorithm* argument. GVM values are not normalised currently.

`calculate_chiparams`(*angles=False*)

Calculate the chi param vector for each atom

Parameters **`angles`** (*`bool, optional`*) – If True, return the list of cosines of all neighbor pairs

Returns **angles** – list of all cosine values, returned only if *angles* is True.

Return type array of floats

Notes

This method tries to distinguish between crystal structures by finding the cosines of angles formed by an atom with its neighbors. These cosines are then historgrammed with bins *[-1.0, -0.945, -0.915, -0.755, -0.705, -0.195, 0.195, 0.245, 0.795, 1.0]* to find a vector for each atom that is indicative of its local coordination. Compared to chi parameters from chi_0 to chi_7 in the associated publication, the vector here is from chi_0 to chi_8. This is due to an additional chi parameter which measures the number of neighbors between cosines -0.705 to -0.195.

Parameter *nlimit* specifies the number of nearest neighbors to be included in the analysis to find the cutoff. If parameter *angles* is true, an array of all cosine values is returned. The publication further provides combinations of chi parameters for structural identification which is not implemented here. The calculated chi params can be accessed using `chiparams`.

References

`calculate_cna`(*cutoff=None, calculate_neighbors=True*)

Calculate the Common Neighbor Analysis indices

Parameters

- **`cutoff`** (*`float, optional`*) – cutoff value to calculate CNA. If not specified, adaptive CNA will be used
- **`calculate_neighbors`** (*`bool, optional`*) – If True, calculate neighbors using number method. If False, existing neighbors will be used. Default True. Only used if the cutoff is None.

Returns **cna** – list of length 5 with number of atoms that belong to each structure.

Return type list of ints

Notes

Performs the common neighbor analysis [1] and assigns a structure to each atom. If *cutoff* is not specified, adaptive common neighbor analysis is used. The assigned structures can be accessed by `structure`. The values assigned for stucture are 0 Unknown, 1 fcc, 2 hcp, 3 bcc, 4 icosahedral.

References

`calculate_disorder`(*averaged=False*, *q=6*)

Calculate the disorder criteria for each atom

Parameters

- **`averaged`** (*`bool, optional`*) – If True, calculate the averaged disorder. Default False.
- **`q`** (*`int, optional`*) – The Steinhardt parameter value over which the bonds have to be calculated. Default 6.

Returns

Return type None

Notes

Calculate the disorder criteria as introduced in [1]. The disorder criteria value for each atom is defined by,

$$D_j = \frac{1}{N_b^j} \sum_{i=1}^{N_b} [S_{jj} + S_{kk} - 2S_{jk}]$$

where .. math:: S_{ij} = sum_{m=-6}^6 q_{6m}(i) q_{6m}^*(i)

The keyword *averaged* is True, the disorder value is averaged over the atom and its neighbors. The disorder value can be accessed using `disorder` and the averaged version can be accessed using `avg_disorder`. For ordered systems, the value of disorder would be zero which would increase and reach one for disordered systems.

References

`calculate_q`(*q*, *averaged=False*)

Find the Steinhardt parameter q_l for all atoms.

Parameters

- **`q_l`** (*`int or list of ints`*) – A list of all Steinhardt parameters to be found from 2-12.
- **`averaged`** (*`bool, optional`*) – If True, return the averaged q values, default False

Returns

Return type None

Notes

Enables calculation of the Steinhardt parameters [1] q from 2-12. The type of q values depend on the method used to calculate neighbors. See the description `find_neighbors()` for more details. If the keyword *average* is set to True, the averaged versions of the bond order parameter [2] is returned.

References

`calculate_rdf`(*histobins=100*, *histomin=0.0*, *histomax=None*)
Calculate the radial distribution function.

Parameters

- **`histobins`** (*`int`*) – number of bins in the histogram
- **`histomin`** (*`float, optional`*) – minimum value of the distance histogram. Default 0.0.
- **`histomax`** (*`float, optional`*) – maximum value of the distance histogram. Default, the maximum value in all pair distances is used.

Returns

- **rdf** (*array of ints*) – Radial distribution function
- **r** (*array of floats*) – radius in distance units

`calculate_solidneighbors`()
Find Solid neighbors of all atoms in the system.

Parameters **`None`** –

Returns

Return type None

Notes

A solid bond is considered between two atoms if the connection between them is greater than 0.6.

`calculate_sro`(*reference_type=1*, *average=True*, *shells=2*)
Calculate short range order

Parameters

- **`reference_type`** (*`int, optional`*) – type of the atom to be used a reference. default 1
- **`average`** (*`bool, optional`*) – if True, average over all atoms of the reference type in the system. default True.

Returns **vec** – The short range order averaged over the whole system for atom of the reference type. Only returned if *average* is True. First value is SRO of the first neighbor shell and the second value corresponds to the second nearest neighbor shell.

Return type list of float

Notes

Calculates the short range order for an AB alloy using the approach by Cowley [1]. Short range order is calculated as,

$$\alpha_i = 1 - \frac{n_i}{m_A c_i}$$

where n_i is the number of atoms of the non reference type among the c_i atoms in the ith shell. m_A is the concentration of the non reference atom. Please note that the value is calculated for shells 1 and 2 by default. In order for this to be possible, neighbors have to be found first using the `find_neighbors()` method. The selected neighbor method should include the second shell as well. For this purpose *method=cutoff* can be chosen with a cutoff long enough to include the second shell. In order to estimate this cutoff, one can use the `calculate_rdf()` method.

References

`calculate_vorovector`(*edge_cutoff=0.05*, *area_cutoff=0.01*, *edge_length=False*)
get the voronoi structure identification vector.

Parameters **`edge_cutoff`** (*`float, optional`*) – cutoff for edge length. Default 0.05.

area_cutoff [float, optional] cutoff for face area. Default 0.01.

edge_length [bool, optional] if True, a list of unrefined edge lengths are returned. Default false.

Returns **vorovector** – array of the form (n3, n4, n5, n6)

Return type array like, int

Notes

Returns a vector of the form *(n3, n4, n5, n6)*, where *n3* is the number of faces with 3 vertices, *n4* is the number of faces with 4 vertices and so on. This can be used to identify structures [1] [2].

The keywords *edge_cutoff* and *area_cutoff* can be used to tune the values to minimise the effect of thermal distortions. Edges are only considered in the analysis if the *edge_length/sum(edge_lengths)* is at least *edge_cutoff*. Similarly, faces are only considered in the analysis if the *face_area/sum(face_areas)* is at least *face_cutoff*.

References

`cluster_atoms`(*condition*, *largest=True*, *new_algo=False*, *cutoff=0*)
Cluster atoms based on a property

Parameters

- **`condition`** (*`callable or atom property`*) – Either function which should take an `Atom` object, and give a True/False output or an attribute of atom class which has value or 1 or 0.
- **`largest`** (*`bool, optional`*) – If True returns the size of the largest cluster. Default False.

Returns **lc** – Size of the largest cluster. Returned only if *largest* is True.

Return type int

Notes

This function helps to cluster atoms based on a defined property. This property is defined by the user through the argument *condition* which is passed as a parameter. *condition* can be of two types. The first type is a function which takes an `Atom` object and should give a True/False value. *condition* can also be an `Atom` attribute or a value from *custom* values stored in an atom.

When clustering, the code loops over each atom and its neighbors. If the *condition* is true for both host atom and the neighbor, they are assigned to the same cluster. For example, a condition to cluster solid atoms would be,

```
def condition(atom):
    #if both atom is solid
    return (atom1.solid)
```

The same can be done by passing *"solid"* as the condition argument instead of the above function. Passing a function allows to evaluate complex conditions, but is slower than passing an attribute.

find_clusters(*recursive=True*, *largest=True*)
Find the clusters of all atoms in the system.

Parameters

- **recursive** (*Bool, optional*) – If True, use a recursive clustering algorithm, otherwise use an id based clustering. The difference in values between two methods can be upto 3 particles. Default True.
- **largest** (*Bool, optional*) – If True, return the number of particles in the largest cluster. Default True.

Returns **cluster** – The size of the largest cluster in the system. Only returned if *largest* is set to True.

Return type int

Notes

Go through all the atoms in the system and cluster them together based on the *issolid* parameter of the atom. To cluster based on any user defined criteria, you can use *set_solid* method of *Atom* to explicitely set the *issolid* value.

Warning: This function is deprecated and will be removed in a future release. Please use `cluster_atoms()` instead.

find_largestcluster()
Find the largest solid cluster of atoms in the system from all the clusters.

Parameters **None** –

Returns **cluster** – the size of the largest cluster

Return type int

Notes

`pyscal.core.System.find_clusters()` has to be used before using this function.

find_neighbors(*method='cutoff'*, *cutoff=None*, *threshold=2*, *filter=None*, *voroexp=1*, *padding=1.2*, *nlimit=6*, *cells=False*, *nmax=12*, *assign_neighbor=True*)

Find neighbors of all atoms in the `System`.

Parameters method (*{'cutoff', 'voronoi', 'number'}*) – *cutoff* method finds neighbors of an atom within a specified or adaptive cutoff distance from the atom. *voronoi* method finds atoms that share a Voronoi polyhedra face with the atom. Default, *cutoff number* method finds a specified number of closest neighbors to the given atom. Number only populates

cutoff [{ float, 'sann', 'adaptive'}] the cutoff distance to be used for the *cutoff* based neighbor calculation method described above. If the value is specified as 0 or *adaptive*, adaptive method is used. If the value is specified as *sann*, sann algorithm is used.

threshold [float, optional] only used if `cutoff=adaptive`. A threshold which is used as safe limit for calculation of cutoff.

filter [{'None', 'type'}, optional] apply a filter to nearest neighbor calculation. If the *filter* keyword is set to *type*, only atoms of the same type would be included in the neighbor calculations. Default None.

voroexp [int, optional] only used if `method=voronoi`. Power of the neighbor weight used to weight the contribution of each atom towards Steinhardt parameter values. Default 1.

padding [double, optional] only used if `cutoff=adaptive` or `cutoff=number`. A safe padding value used after an adaptive cutoff is found. Default 1.2.

nlimit [int, optional] only used if `cutoff=adaptive`. The number of particles to be considered for the calculation of adaptive cutoff. Default 6.

nmax [int, optional] only used if `cutoff=number`. The number of closest neighbors to be found for each atom. Default 12

Returns

Return type None

Raises

- `RuntimeWarning` – raised when *threshold* value is too low. A low threshold value will lead to 'sann' algorithm not converging when finding a neighbor. This function will try to automatically increase *threshold* and check again.
- `RuntimeError` – raised when neighbor search was unsuccessful. This is due to a low *threshold* value.

Notes

This function calculates the neighbors of each particle. There are several ways to do this. A complete description of the methods can be found here.

Method cutoff and specifying a cutoff radius uses the traditional approach being the one in which the neighbors of an atom are the ones that lie in the cutoff distance around it.

In order to reduce time during the distance sorting during the adaptive methods, pyscal sets an initial guess for a cutoff distance. This is calculated as,

$$r_{initial} = threshold * (simulation\ box\ volume/number\ of\ particles)^{(1/3)}$$

threshold is a safe multiplier used for the guess value and can be set using the *threshold* keyword.

In Method cutoff, if `cutoff='adaptive'`, an adaptive cutoff is found during runtime for each atom [1]. Setting the cutoff radius to 0 also uses this algorithm. The cutoff for an atom i is found using,

$$r_c(i) = padding * ((1/nlimit) * \sum_{j=1}^{nlimit} (r_{ij}))$$

padding is a safe multiplier to the cutoff distance that can be set through the keyword *padding*. *nlimit* keyword sets the limit for the top nlimit atoms to be taken into account to calculate the cutoff radius.

In Method cutoff, if `cutoff='sann'`, sann algorithm is used [2]. There are no parameters to tune sann algorithm.

The second approach is using Voronoi polyhedra which also assigns a weight to each neighbor in the ratio of the face area between the two atoms. Higher powers of this weight can also be used [3]. The keyword *voroexp* can be used to set this weight.

If method os *number*, instead of using a cutoff value for finding neighbors, a specified number of closest atoms are found. This number can be set through the argument *nmax*.

Warning: Adaptive and number cutoff uses a padding over the intial guessed "neighbor distance". By default it is 2. In case of a warning that `threshold` is inadequate, this parameter should be further increased. High/low value of this parameter will correspond to the time taken for finding neighbors.

References

`find_solids`(*bonds=0.5*, *threshold=0.5*, *avgthreshold=0.6*, *cluster=True*, *q=6*, *new_algo=False*, *cutoff=0*)

Distinguish solid and liquid atoms in the system.

Parameters

- **`bonds`** (*`int or float, optional`*) – Minimum number of solid bonds for an atom to be identified as a solid if the value is an integer. Minimum fraction of neighbors of an atom that should be solid for an atom to be solid if the value is float between 0-1. Default 0.5.
- **`threshold`** (*`double, optional`*) – Solid bond cutoff value. Default 0.5.
- **`avgthreshold`** (*`double, optional`*) – Value required for Averaged solid bond cutoff for an atom to be identified as solid. Default 0.6.
- **`cluster`** (*`bool, optional`*) – If True, cluster the solid atoms and return the number of atoms in the largest cluster.
- **`q`** (*`int, optional`*) – The Steinhardt parameter value over which the bonds have to be calculated. Default 6.

Returns **solid** – Size of the largest solid cluster. Returned only if *cluster=True*.

Return type int

Notes

The neighbors should be calculated before running this function. Check *`find_neighbors()`* method.

bonds define the number of solid bonds of an atom to be identified as solid. Two particles are said to be 'bonded' if [1],

$$s_{ij} = \sum_{m=-6}^{6} q_{6m}(i) q_{6m}^*(i) \geq threshold$$

where *threshold* values is also an optional parameter.

If the value of *bonds* is a fraction between 0 and 1, at least that much of an atom's neighbors should be solid for the atom to be solid.

An additional parameter *avgthreshold* is an additional parameter to improve solid-liquid distinction. In addition to having a the specified number of *bonds*,

$$\langle s_{ij} \rangle > avgthreshold$$

also needs to be satisfied. In case another q value has to be used for calculation of S_ij, it can be set used the *q* attribute.

References

get_atom (*index*)

Get the `Atom` object at the queried position in the list of all atoms in the `System`.

Parameters **index** (`int`) – index of required atom in the list of all atoms.

Returns **atom** – atom object at the queried position.

Return type Atom object

get_custom (*atom*, *customkeys*)

Get a custom attribute from Atom

Parameters

- **atom** (`Atom object`) –
- **customkeys** (`list of strings`) – the list of keys to be found

Returns **vals** – array of custom values

Return type list

get_distance (*atom1*, *atom2*, *vector=False*)

Get the distance between two atoms.

Parameters

- **atom1** (*Atom* object) – first atom
- **atom2** (*Atom* object) – second atom
- **vector** (`bool, optional`) – If True, the displacement vector connecting the atoms is also returned. default false.

Returns **distance** – distance between the first and second atom.

Return type double

Notes

Periodic boundary conditions are assumed by default.

get_qvals(*q*, *averaged=False*)

Get the required q_l (Steinhardt parameter) values of all atoms.

Parameters

- **q_l** (*int or list of ints*) – required q_l value with l from 2-12
- **averaged** (*bool, optional*) – If True, return the averaged q values, default False

Returns **qvals** – list of q_l of all atoms.

Return type list of floats

Notes

The function returns a list of q_l values in the same order as the list of the atoms in the system.

greedy_edge_selection(*nmax*)

Greedy edge selection scheme for centrosymmetry parameters

Parameters **nmax** (*int*) – number of neighbors to be considered for centrosymmetry parameters

Returns

Return type None

References

greedy_vertex_matching(*nmax*)

Greedy vertex matching scheme for centrosymmetry parameters

Parameters **nmax** (*int*) – number of neighbors to be considered for centrosymmetry parameters

Returns

Return type None

References

read_inputfile(*filename*, *format='lammps-dump'*, *frame=-1*, *compressed=False*, *customkeys=None*, *is_triclinic=False*)

Read input file that contains the information of system configuration.

Parameters

- **filename** (*string*) – name of the input file.
- **format** (*{'lammps-dump', 'poscar', 'ase', 'mdtraj'}*) – format of the input file, in case of *ase* the ASE Atoms object
- **compressed** (*bool, optional*) – If True, force to read a *gz* compressed format, default False.

- **frame** (*int*) – If the trajectory contains more than one time step, the slice can be specified using the *frame* option.

  Note: works only with *lammps-dump* format.

- **customkeys** (*list*) – A list containing names of headers of extra data that needs to be read in from the input file.
- **is_triclinc** (*bool, optional*) – Only used in the case of *format='ase'*. If the read ase object is triclinic, this options should be set to True.

Returns

Return type None

Notes

format keyword specifies the format of the input file. Currently only a *lammps-dump* and *poscar* files are supported. Additionaly, the widely use Atomic Simulation environment (https://wiki.fysik.dtu.dk/ase/ase/ase.html). mdtraj objects (http://mdtraj.org/1.9.3/) are also supported by using the keyword *'mdtraj'* for format. Please note that triclinic boxes are not yet supported for mdtraj format. Atoms object can also be used directly. This function uses the `traj_process()` module to process a file which is then assigned to system.

compressed keyword is not required if a file ends with *.gz* extension, it is automatically treated as a compressed file.

frame keyword allows to read in a particular slice from a long trajectory. If all slices need to analysed, this is a very inefficient way. For handling multiple time slices, the `traj_process()` module offers a better set of tools.

Triclinic simulation boxes can also be read in for *lammps-dump*. No special keyword is necessary.

If *custom_keys* are provided, this extra information is read in from input files if available. This information is not passed to the C++ instance of atom, and is stored as a dictionary. It can be accessed directly as *atom.custom['customval']*

reset_neighbors()

Reset the neighbors of all atoms in the system.

Parameters **None** –

Returns

Return type None

Notes

It is used automatically when neighbors are recalculated.

set_atom(*atom*)

Return the atom to its original location after modification.

Parameters **atom** (`Atom`) – atom to be replaced

Returns

Return type None

Notes

For example, an *`Atom`* at location *i* in the list of all atoms in *`System`* can be queried by, `atom = System.get_atom(i)`, then any kind of modification, for example, the position of the *Atom* can done by, `atom.pos = [2.3, 4.5, 4.5]`. After modification, the *Atom* can be set back to its position in *System* by *`set_atom()`*.

Although the complete list of atoms can be accessed or set using `atoms = sys.atoms`, *get_atom* and *set_atom* functions should be used for accessing individual atoms. If an atom already exists at that index in the list, it will be overwritten and will lead to loss of information.

`to_file`(*outfile*, *format='lammps-dump'*, *customkeys=None*, *compressed=False*, *timestep=0*)
Save the system instance to a trajectory file.

Parameters

outfile [string] name of the output file

format [string, optional] format of the output file, default *lammps-dump* Currently only *lammps-dump* format is supported.

customkeys [list of strings, optional] a list of extra atom wise values to be written in the output file.

compressed [bool, optional] If true, the output is written as a compressed file.

timestep [int, optional] timestep to be written to file. default 0

Returns

Return type None

Notes

`pyscal.core.`**`test`**()
A simple function to test if the module works

Parameters **`None`** –

Returns **works** – True if the module works and could create a System and Atom object False otherwise.

Return type bool

`class` `pyscal.catom.`**`Atom`**
Bases: `pybind11_builtins.pybind11_object`

Class to store atom details.

Parameters

- **`pos`** (*`list of floats of length 3`*) – position of the *Atom*, default [0,0,0]
- **`id`** (*`int`*) – id of the *Atom*, default 0
- **`type`** (*`int`*) – type of the *Atom*, default 1

Notes

A pybind11 class for holding the properties of a single atom. Various properties of the atom can be accessed through the attributes and member functions which are described below in detail. Atoms can be created individually or directly by reading a file. Check the examples for more details on how atoms are created. For creating atoms directly from an input file check the documentation of `System` class.

Although an *Atom* object can be created independently, *Atom* should be thought of inherently as members of the `System` class. All the properties that define an atom are relative to the parent class. `System` has a list of all atoms. All the properties of an atom, hence should be calculated through `System`.

Examples

```
>>> #method 1 - individually
>>> atom = Atom()
>>> #now set positions of the atoms
>>> atom.pos = [23.0, 45.2, 34.2]
>>> #now set id
>>> atom.id = 23
>>> #now set type
>>> atom.type = 1
>>> #Setting through constructor
>>> atom = Atom([23.0, 45.2, 34.2], 23, 1)
```

References

Creation of atoms.

allaq
list of floats. list of all averaged q values of the atom.

allq
list of floats. list of all q values of the atom.

angular
Float. The value of angular parameter A of an atom. The angular parameter measures the tetrahedral coordination of an atom. Meaningful values are only returned if the property is calculated using `calculate_angularcriteria()`.

avg_angular
Float. The average angular parameter value. Not used currently.

avg_disorder
Float. The value of averaged disorder parameter.

avg_sij
float. Value of averaged s_ij which is used for identification of solid atoms. s_ij is defined by

$$s_{ij} = \sum_{m=-l}^{l} q_{lm}(i) q_{lm}^*(i)$$

avg_volume
float. Averaged version of the Voronoi volume which is calculated as an average over itself and its neighbors. Only calculated when the `find_neighbors()` using the *method='voronoi'* option is used.

bond_chain_count

bonds
Int. The number of solid bonds of an atom.

calculate_adaptive_cna()

centrosymmetry
Float. The value of centrosymmetry parameter.

chiparams
Float. The value of chiparameter of an atom. The return value is a vector of length 8. Meaningful values are only returned if chi params are calculated using *calculate_chiparams()*.

cluster
int. identification number of the cluster that the atom belongs to.

common_neighbor_bond_count

common_neighbor_bonds

common_neighbor_count

common_neighbors

condition
int. condition that specifies if an atom is included in the clustering algorithm or not. Only atoms with the value of condition=1 will be used for clustering in *cluster_atoms()*.

coordination
int. coordination number of the atom. Coordination will only be updated after neighbors are calculated using *find_neighbors()*.

custom
dict. dictionary specfying custom values for an atom. The module only stores the id, type and position of the atom. If any extra values need to be stored, they can be stored in custom using *atom.custom = {“velocity”:12}*. *read_inputfile()* can also read in extra atom information. By default, custom values are treated as string.

cutoff
double. cutoff used for finding neighbors for each atom.

disorder
Float. The value of disorder parameter.

edge_lengths
list of floats. For each face, this vector contains the lengths of edges that make up the Voronoi polyhedra of the atom. Only calculated when the *find_neighbors()* using the *method=’voronoi’* option is used.

face_perimeters
list of floats. List consisting of the perimeters of each Voronoi face of an atom. Only calculated when the *find_neighbors()* using the *method=’voronoi’* option is used.

face_vertices
list of floats. A list of the number of vertices shared between an atom and its neighbors. Only calculated when the *find_neighbors()* using the *method=’voronoi’* option is used.

get_q()
Calculate the steinhardt parameter q_l value.

Parameters

- **q** (*int or list of ints*) – number of the required q_l - from 2-12
- **averaged** (*bool, optional*) – If True, return the averaged q values, If False, return the non averaged ones default False

Returns **q_l** – the value(s) of the queried Steinhardt parameter(s).

Return type float or list of floats

Notes

Please check this link for more details about Steinhardts parameters and the averaged versions.

Meaningful values are only returned if `calculate_q()` is used.

`get_qlm()`

Get the q_lm values.

Parameters

- **`q`** (*`int`*) – number of the required q_l - from 2-12
- **`averaged`** (*`bool, optional`*) – If True, return the averaged qlm values, If False, return the non averaged ones default False

Returns

- **q_lm** (*complex vector*) – vector of complex numbers.
- Meaningful values are only returned if `calculate_q()` is used.

`id`

int. Id of the atom.

`largest_cluster`

bool. True if the atom belongs to the largest cluster, False otherwise. Largest cluster is only identified after using the `cluster_atoms()` function.

`lcutlarge`

`lcutsmall`

`loc`

int. indicates the position of the atom in the list of all atoms.

`local_angles`

List of floats of length 2. List of longitude and colatitude of an atom to its neighbors.

`mask`

bool. Mask variable for atom. If mask is true, the atom is ignored from calculations.

`neighbor_distance`

List of floats. List of neighbor distances of the atom.

`neighbor_vector`

List of floats of length 3. List of vectors connecting an atom to its neighbors.

`neighbor_weights`

List of floats. Used to weight the contribution of each neighbor atom towards the value of Steinhardt's parameters. By default, each atom has a weight of 1 each. However, if `find_neighbors()` is used with *method='voronoi'*, each neighbor gets a weight proportional to the area shared between the neighboring atom and host atom.

`neighbors`

List of ints. List of neighbors of the atom. The list contains indices of neighbor atoms which indicate their position in the list of all atoms.

`next_neighbor_distances`

double. cutoff used for finding neighbors for each atom.

next_neighbors
double. cutoff used for finding neighbors for each atom.

pos
List of floats of the type [x, y, z], default [0, 0, 0]. Position of the atom.

set_q()
Set the value of steinhardt parameter q_l.

Parameters

- **q** (*int or list of ints*) – number of the required q_l - from 2-12
- **val** (*float or list of floats*) – value(s) of Steinhardt parameter(s).
- **averaged** (*bool, optional*) – If True, return the averaged q values, If False, return the non averaged ones default False

Returns

Return type None

sij
float. Value of s_ij which is used for identification of solid atoms. s_ij is defined by

$$s_{ij} = \sum_{m=-l}^{l} q_{lm}(i) q_{lm}^*(i)$$

solid
bool. True if the atom is solid, False otherwise. Solid atoms are only identified after using the `find_solids()` function.

sro
Float. The value of short range order parameter.

structure
int. Indicates the structure of atom. Not used currently.

surface
bool. True if the atom has at least one liquid neighbor, False otherwise. Surface atoms are only identified after using the `find_solids()` function.

type
int. int specifying type of the atom.

vertex_numbers
list of floats. For each Voronoi face of the atom, this values includes a List of vertices that constitute the face. Only calculated when the `find_neighbors()` using the *method='voronoi'* option is used.

vertex_vectors
list of floats. A list of positions of each vertex of the Voronoi polyhedra of the atom. Only calculated when the `find_neighbors()` using the *method='voronoi'* option is used.

volume
float. Voronoi volume of the atom. The Voronoi volume is only calculated if neighbors are found using the `find_neighbors()` using the *method='voronoi'* option.

vorovector
list of ints. A vector of the form *(n3, n4, n5, n6)* where n3 is the number of faces with 3 vertices, n4 is the number of faces with 4 vertices and so on. This can be used to identify structures [1][2]. Vorovector is calculated if the `calculate_vorovector()` method is used.

References

pyscal.crystal_structures module

pyscal module for creating crystal structures.

`pyscal.crystal_structures.`**`make_crystal`**(*structure*, *lattice_constant=1.0*, *repetitions=None*, *ca_ratio=1.633*, *noise=0*)

Create a basic crystal structure and return it as a list of *Atom* objects and box dimensions.

Parameters

- **`structure`** (*{'bcc', 'fcc', 'hcp', 'diamond' or 'l12'}*) – type of the crystal structure
- **`lattice_constant`** (*float, optional*) – lattice constant of the crystal structure, default 1
- **`repetitions`** (*list of ints of len 3, optional*) – of type *[nx, ny, nz]*, repetions of the unit cell in x, y and z directions. default *[1, 1, 1]*.
- **`ca_ratio`** (*float, optional*) – ratio of c/a for hcp structures, default 1.633
- **`noise`** (*float, optional*) – If provided add normally distributed noise with standard deviation *noise* to the atomic positions.

Returns

- **atoms** (list of *Atom* objects) – list of all atoms as created by user input
- **box** (*list of list of floats*) – list of the type *[[xlow, xhigh], [ylow, yhigh], [zlow, zhigh]]* where each of them are the lower and upper limits of the simulation box in x, y and z directions respectively.

Examples

```
>>> atoms, box = make_crystal('bcc', lattice_constant=3.48, repetitions=[2,2,2])
>>> sys = System()
>>> sys.assign_atoms(atoms, box)
```

pyscal.traj_process module

pyscal module containing methods for processing of a trajectory. Methods for reading of input files formats, writing of output files etc are provided in this module.

`pyscal.traj_process.`**`read_ase`**(*aseobject*, *check_triclinic=False*, *box_vectors=False*)

Function to read from a ASE atoms objects

Parameters

- **`aseobject`** (*ASE Atoms object*) – name of the ASE atoms object
- **`triclinic`** (*bool, optional*) – True if the configuration is triclinic
- **`box_vectors`** (*bool, optional*) – If true, return the full box vectors along with *boxdims* which gives upper and lower bounds. default False.

`pyscal.traj_process.`**`read_lammps_dump`**(*infile*, *compressed=False*, *check_triclinic=False*, *box_vectors=False*, *customkeys=None*)

Function to read a lammps dump file format - single time slice.

Parameters

- **infile** (*string*) – name of the input file
- **compressed** (*bool, optional*) – force to read a *gz* zipped file. If the filename ends with *.gz*, use of this keyword is not necessary. Default True.
- **check_triclinic** (*bool, optional*) – If true check if the sim box is triclinic. Default False.
- **box_vectors** (*bool, optional*) – If true, return the full box vectors along with *boxdims* which gives upper and lower bounds. default False.
- **customkeys** (*list of strings, optional*) – A list of extra keywords to read from trajectory file.

Returns

- **atoms** (list of *Atom* objects) – list of all atoms as created by user input
- **boxdims** (*list of list of floats*) – The dimensions of the box. This is of the form *[[xlo, xhi],[ylo, yhi],[zlo, zhi]]* where *lo* and *hi* are the upper and lower bounds of the simulation box along each axes. For triclinic boxes, this is scaled to *[0, scalar length of the vector]*.
- **box** (*list of list of floats*) – list of the type *[[x1, x2, x3], [y1, y2, y3], [zz1, z2, z3]]* which are the box vectors. Only returned if *box_vectors* is set to True.
- **triclinic** (*bool*) – True if the box is triclinic. Only returned if *check_triclinic* is set to True
- .. *note::* – Values are always returned in the order *atoms, boxdims, box, triclinic* if all return keywords are selected. For example, ff *check_triclinic* is not selected, the return values would still preserve the order and fall back to *atoms, boxdims, box*.

Notes

Read a lammps-dump style snapshot that can have variable headers, reads in type and so on. Zipped files which end with a *.gz* can also be read automatically. However, if the file does not end with a *.gz* extension, keyword *compressed = True* can also be used.

Examples

```
>>> atoms, box = read_lammps_dump('conf.dump')
>>> atoms, box = read_lammps_dump('conf.dump.gz')
>>> atoms, box = read_lammps_dump('conf.d', compressed=True)
```

`pyscal.traj_process.`**`read_mdtraj`**(*mdobject*, *check_triclinic=False*, *box_vectors=False*)
Function to read from an MDTraj atoms objects

Parameters

- **mdobject** (*MDTraj Atoms object*) – name of the MDTraj atoms object
- **triclinic** (*bool, optional*) – True if the configuration is triclinic
- **box_vectors** (*bool, optional*) – If true, return the full box vectors along with *boxdims* which gives upper and lower bounds. default False.

`pyscal.traj_process.`**`read_poscar`**(*infile*, *compressed=False*, *box_vectors=False*)
Function to read a POSCAR format.

Parameters

- **infile** (*string*) – name of the input file
- **compressed** (*bool, optional*) – force to read a *gz* zipped file. If the filename ends with *.gz*, use of this keyword is not necessary, Default False

Returns

- **atoms** (list of *Atom* objects) – list of all atoms as created by user input
- **box** (*list of list of floats*) – list of the type *[[xlow, xhigh], [ylow, yhigh], [zlow, zhigh]]* where each of them are the lower and upper limits of the simulation box in x, y and z directions respectively.

Examples

```
>>> atoms, box = read_poscar('POSCAR')
>>> atoms, box = read_poscar('POSCAR.gz')
>>> atoms, box = read_poscar('POSCAR.dat', compressed=True)
```

pyscal.traj_process.**split_traj_lammps_dump**(*infile*, *compressed=False*)
Read in a LAMMPS dump trajectory file and convert it to individual time slices.

Parameters

- **filename** (*string*) – name of input file
- **compressed** (*bool, optional*) – force to read a *gz* zipped file. If the filename ends with *.gz*, use of this keyword is not necessary, Default False.

Returns **snaps** – a list of filenames which contain individual frames from the main trajectory.

Return type list of strings

pyscal.traj_process.**split_trajectory**(*infile*, *format='lammps-dump'*, *compressed=False*)
Read in a trajectory file and convert it to individual time slices.

Parameters

- **filename** (*string*) – name of input file
- **format** (*format of the input file*) – only *lammps-dump* is supported now.
- **compressed** (*bool, optional*) – force to read a *gz* zipped file. If the filename ends with *.gz*, use of this keyword is not necessary.

Returns **snaps** – a list of filenames which contain individual frames from the main trajectory.

Return type list of strings

Notes

This is a wrapper function around *split_traj_lammps_dump* function.

pyscal.traj_process.**write_poscar**(*sys*, *outfile*, *comments='pyscal'*)
Function to read a POSCAR format.

Parameters **outfile** (*string*) – name of the input file

pyscal.traj_process.**write_structure**(*sys*, *outfile*, *format='lammps-dump'*, *compressed=False*, *customkey=None*, *customvals=None*, *timestep=0*)
Write the state of the system to a trajectory file.

Parameters

- **sys** (*System* object) – the system object to be written out
- **outfile** (`string`) – name of the output file
- **format** (`string, optional`) – the format of the output file, as of now only *lammps-dump* format is supported.
- **compressed** (`bool, default false`) – write a *.gz* format
- **customkey** (`string or list of strings, optional`) – If specified, it adds this custom column to the dump file. Default None.
- **customvals** (`list or list of lists, optional`) – If *customkey* is specified, *customvals* take an array of the same length as number of atoms, which contains the values to be written out.
- **timestep** (`int, optional`) – Specify the timestep value, default 0

Returns

Return type None

pyscal.misc module

`pyscal.misc.`**`compare_atomic_env`**(*infile*, *atomtype=2*, *precision=2*, *format='poscar'*, *print_results=True*, *return_system=False*)

Compare the atomic environment of given types of atoms in the inputfile. The comparison is made in terms of Voronoi volume and Voronoi fingerprint.

Parameters

- **infile** (`string`) – name of the inputfile
- **atomtype** (`int, optional`) – type of the atom default 2
- **precision** (`float, optional`) – precision for comparing Voronoi volumes default 3
- **format** (`string, optional`) – format of the input file default poscar
- **print_results** (`bool, optional`) – if True, print the results. If False, return the data instead. default True
- **return_system** (`bool, optional`) – if True, return the system object. default False

Returns

- **vvx** (*list of floats*) – unique Voronoi volumes. Returned only if print results is False
- **vrx** (*list of strings*) – unique Voronoi polyhedra. Returned only if print results is False
- **vvc** (*list of ints*) – number of unique quantities specified above. Returned only if print results is False

CHAPTER 8

Support, contributing and extending

8.1 Support, contributing and extending

pyscal welcomes and appreciates contribution and extension to the module. Rather than local modifications, we request that the modifications be submitted through a pull request, so that the module can be continuously improved.

The following links will help you get started with contributing to pyscal.

8.1.1 Contributions to existing features

Reporting and fixing bugs

In case a bug is found in the module, it can be reported on the issues page of the repository. After clicking on the new issue button, there is template already provided for Bug report. Please choose this and make sure the necessary fields are filled. Once a bug is reported, the status can once again monitored on the issues page. Additionally, you are of course very welcome to fix any existing bugs.

8.1.2 New feautures

If you have an idea for new feature, you can submit a feature idea through the issues page of the repository. After choosing new issue, please choose the template for feature request. As much as information as you can provide about the new feauture would be greatly helpful. Additionally, you could also work on feature requests already on the issues page. The following instructions will help you get started with local feature development.

Setting up local environment

1. The first step is to fork pyscal. A detailed tutorial on forking can be found here. After forking, clone the repository to your local machine.
2. We recommend creating a virtual environment to test new features or improvements to features. See this link for help on managing environments.

3. Once the environment is set up, you can create a new branch for your feature by `git checkout -b new_feauture`.
4. Now implement the necessary feature.
5. Once done, you can reinstall pyscal by `python setup.py install`. After that please make sure that the existing tests work by running `pytest tests/` from the main module folder.
6. If the tests work, you are almost done! If the new feature is not covered in existing tests, you can to write a new test in the tests folder. pyscal uses pytest for tests. This link will help you get started.
7. Add the necessary docstrings for the new functions implemented. pyscal uses the numpy docstring format for documentation.
8. Bonus task: Set up few examples that document how the feature works in the `docs/source/` folder and link it to the examples section.
9. Final step - Submit a pull request through github. Before you submit, please make sure that the new feature is documented and has tests. Once the request is submitted, automated tests would be done. Your pull request will fail the tests if - the unit tests fail, or if the test coverage falls below 80%. If all tests are successful, your feauture will be incorporated to pyscal and your contributions will be credited.

If you have trouble with any of the steps, or you need help, please send an email and we will be happy to help! All of the contributions are greatly appreciated and will be credited in Developers/Acknowledgements page.

8.2 Help and support

In case of bugs and feature improvements, you are welcome to create a new issue on the github repo. You are also welcome to fix a bug or implement a feature. Please see the extending and contributing section for more details.

Any other questions or suggestions are welcome, please contact us.

8.3 Common issues

- **Installation of the package without administrator privileges**

  In case of any problems with installation, we recommend that you install the package in a conda environment. Details on managing and creating environments can be found here .

- **C++11 is not available**

  pyscal needs C++11 to install and run. In case you do not have c++11, it can be installed in a conda environment by following this link . After installing gcc, once the conda environment is reactivated, the environment variables are set. The installation should proceed now without problems.

- **cmake is not available**

  cmake can be installed in a conda environment. Details on managing and creating environments can be found here . The cmake package can be installed from conda following this link.

CHAPTER 9

Credits

9.1 Citing the code

If you use pyscal in your work, the citation of the following article will be greatly appreciated:

Sarath Menon, Grisell Díaz Leines and Jutta Rogal (2019). pyscal: A python module for structural analysis of atomic environments. Journal of Open Source Software, 4(43), 1824, https://doi.org/10.21105/joss.01824

Citation in bib format can be downloaded here.

9.2 Developers

- Sarath Menon
- Grisell Díaz Leines
- Jutta Rogal

9.3 Contributers

- Jan Janßen - developing and maintaining a conda-forge recipe.

9.4 Acknowledgements

We acknowledge Bond order analysis code for the inspiration and the base for what later grew to be `pyscal`. We are also thankful to the developers of Voro++ and pybind11 for developing the great tools that we could use in `pyscal`.

We are grateful to Alberto Ferrari, Abril Azócar Guzmán, Matteo Rinaldi and Yanyan Liang for helping with testing the code and providing valuable feedback. We also thank Mahesh Prasad for the helpful discussions in the implementation of this module.

Sarath Menon acknowledges a scholarship from the International Max Planck Research School for Interface Controlled Materials for Energy Conversion, and is grateful for the help and support received during the E-CAM High Throughput Computing ESDW held in Turin in 2018 and 2019. The authors acknowledge support by the Mexican National Council for Science and Technology (CONACYT) through project 232090 and by the German Research Foundation (DFG) through project 262052203.

This module was developed at the Interdisciplinary Centre for Advanced Materials Simulation, at the Ruhr University Bochum, Germany.

CHAPTER 10

License

10.1 `pyscal` License

`pyscal` uses the GNU General Public License Version 3 . A full description is available in the above link or in the repository.

In addition, `pyscal` license of other codes that `pyscal` uses are given below-

10.1.1 Voro++ license

10.1.2 pybind 11 license

10.1.3 sphinx theme license

Python Module Index

p
pyscal.catom, 57
pyscal.core, 45
pyscal.crystal_structures, 62
pyscal.misc, 65
pyscal.traj_process, 62

Index

A

allaq (*pyscal.catom.Atom attribute*), 58
allq (*pyscal.catom.Atom attribute*), 58
angular (*pyscal.catom.Atom attribute*), 58
Atom (*class in pyscal.catom*), 57
atoms (*pyscal.core.System attribute*), 45
avg_angular (*pyscal.catom.Atom attribute*), 58
avg_disorder (*pyscal.catom.Atom attribute*), 58
avg_sij (*pyscal.catom.Atom attribute*), 58
avg_volume (*pyscal.catom.Atom attribute*), 58

B

bond_chain_count (*pyscal.catom.Atom attribute*), 58
bonds (*pyscal.catom.Atom attribute*), 58
box (*pyscal.core.System attribute*), 45

C

calculate_adaptive_cna() (*pyscal.catom.Atom method*), 59
calculate_angularcriteria() (*pyscal.core.System method*), 46
calculate_centrosymmetry() (*pyscal.core.System method*), 46
calculate_chiparams() (*pyscal.core.System method*), 47
calculate_cna() (*pyscal.core.System method*), 47
calculate_disorder() (*pyscal.core.System method*), 48
calculate_q() (*pyscal.core.System method*), 48
calculate_rdf() (*pyscal.core.System method*), 49
calculate_solidneighbors() (*pyscal.core.System method*), 49
calculate_sro() (*pyscal.core.System method*), 49
calculate_vorovector() (*pyscal.core.System method*), 50
centrosymmetry (*pyscal.catom.Atom attribute*), 59
chiparams (*pyscal.catom.Atom attribute*), 59
cluster (*pyscal.catom.Atom attribute*), 59
cluster_atoms() (*pyscal.core.System method*), 50
common_neighbor_bond_count (*pyscal.catom.Atom attribute*), 59
common_neighbor_bonds (*pyscal.catom.Atom attribute*), 59
common_neighbor_count (*pyscal.catom.Atom attribute*), 59
common_neighbors (*pyscal.catom.Atom attribute*), 59
compare_atomic_env() (*in module pyscal.misc*), 65
condition (*pyscal.catom.Atom attribute*), 59
coordination (*pyscal.catom.Atom attribute*), 59
custom (*pyscal.catom.Atom attribute*), 59
cutoff (*pyscal.catom.Atom attribute*), 59

D

disorder (*pyscal.catom.Atom attribute*), 59

E

edge_lengths (*pyscal.catom.Atom attribute*), 59

F

face_perimeters (*pyscal.catom.Atom attribute*), 59
face_vertices (*pyscal.catom.Atom attribute*), 59
find_clusters() (*pyscal.core.System method*), 51
find_largestcluster() (*pyscal.core.System method*), 51
find_neighbors() (*pyscal.core.System method*), 51
find_solids() (*pyscal.core.System method*), 53

G

get_atom() (*pyscal.core.System method*), 54
get_custom() (*pyscal.core.System method*), 54
get_distance() (*pyscal.core.System method*), 54
get_q() (*pyscal.catom.Atom method*), 59
get_qlm() (*pyscal.catom.Atom method*), 60
get_qvals() (*pyscal.core.System method*), 55

greedy_edge_selection() (*pyscal.core.System method*), 55
greedy_vertex_matching() (*pyscal.core.System method*), 55

I

id (*pyscal.catom.Atom attribute*), 60

L

largest_cluster (*pyscal.catom.Atom attribute*), 60
lcutlarge (*pyscal.catom.Atom attribute*), 60
lcutsmall (*pyscal.catom.Atom attribute*), 60
loc (*pyscal.catom.Atom attribute*), 60
local_angles (*pyscal.catom.Atom attribute*), 60

M

make_crystal() (*in module pyscal.crystal_structures*), 62
mask (*pyscal.catom.Atom attribute*), 60

N

neighbor_distance (*pyscal.catom.Atom attribute*), 60
neighbor_vector (*pyscal.catom.Atom attribute*), 60
neighbor_weights (*pyscal.catom.Atom attribute*), 60
neighbors (*pyscal.catom.Atom attribute*), 60
next_neighbor_distances (*pyscal.catom.Atom attribute*), 60
next_neighbors (*pyscal.catom.Atom attribute*), 61

P

pos (*pyscal.catom.Atom attribute*), 61
pyscal.catom (*module*), 57
pyscal.core (*module*), 45
pyscal.crystal_structures (*module*), 62
pyscal.misc (*module*), 65
pyscal.traj_process (*module*), 62

R

read_ase() (*in module pyscal.traj_process*), 62
read_inputfile() (*pyscal.core.System method*), 55
read_lammps_dump() (*in module pyscal.traj_process*), 62
read_mdtraj() (*in module pyscal.traj_process*), 63
read_poscar() (*in module pyscal.traj_process*), 63
reset_neighbors() (*pyscal.core.System method*), 56

S

set_atom() (*pyscal.core.System method*), 56
set_q() (*pyscal.catom.Atom method*), 61
sij (*pyscal.catom.Atom attribute*), 61
solid (*pyscal.catom.Atom attribute*), 61
split_traj_lammps_dump() (*in module pyscal.traj_process*), 64
split_trajectory() (*in module pyscal.traj_process*), 64
sro (*pyscal.catom.Atom attribute*), 61
structure (*pyscal.catom.Atom attribute*), 61
surface (*pyscal.catom.Atom attribute*), 61
System (*class in pyscal.core*), 45

T

test() (*in module pyscal.core*), 57
to_file() (*pyscal.core.System method*), 57
type (*pyscal.catom.Atom attribute*), 61

V

vertex_numbers (*pyscal.catom.Atom attribute*), 61
vertex_vectors (*pyscal.catom.Atom attribute*), 61
volume (*pyscal.catom.Atom attribute*), 61
vorovector (*pyscal.catom.Atom attribute*), 61

W

write_poscar() (*in module pyscal.traj_process*), 64
write_structure() (*in module pyscal.traj_process*), 64