pyscal Documentation

Release 2.10.13

Sarath Menon, Grisell Díaz Leines, Jutta Rogal

Mar 05, 2021

Contents

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

For general documentation and examples see here .

CHAPTER 3

pyscal reference

3.1 pyscal reference

3.1.1 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')
```

add_atoms(*atoms*)
Add a given list of atoms

Parameters **atoms** (*List of Atoms*) –

Returns

Return type None

atoms
Atom access

box
Wrap for inbuilt box

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*, *get_vals=True*)
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

Returns

Return type None

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 Greedy Edge Selection (GES) [1] as specified in [2] in used in this method. GES algorithm is implemented in LAMMPS and Ovito. Please see [2] for a detailed description of the algorithms.

References

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(*lattice_constant=None*)

Calculate the Common Neighbor Analysis indices

Parameters **lattice_constant** (*float, optional*) – lattice constant to calculate CNA. If not specified, adaptive CNA will be used

Returns **cna** – dict containing the cna signature of the system

Return type dict

Notes

Performs the common neighbor analysis [1][2] or the adaptive common neighbor analysis [2] and assigns a structure to each atom.

If *lattice_constant* is specified, a convential common neighbor analysis is used. If *lattice_constant* 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_energy(*species='Au'*, *pair_style=None*, *pair_coeff=None*, *mass=1.0*, *averaged=False*)
Calculate the potential energy of atom using LAMMPS

Parameters

- **species** (*str*) – Name of atomic species
- **pair_style** (*str*) – lammps pair style
- **pair_coeff** (*str*) – lammps pair coeff
- **mass** (*float*) – mass of the atoms
- **averaged** (*bool, optional*) – Average the energy over neighbors if True default False.

Returns

Return type None

Notes

Calculates the potential energy per atom using the given potential through LAMMPS. More documentation coming up...

Values can be accessed through *pyscal.catom.Atom.energy* Averaged values can be accessed through *pyscal.catom.Atom.avg_energy*

If *averaged* is True, the energy is averaged over the neighbors of an atom. If neighbors were calculated before calling this method, those neighbors are used for averaging. Otherwise neighbors are calculated on the fly with an adaptive cutoff method.

calculate_entropy(*rm*, *sigma=0.2*, *rstart=0.001*, *h=0.001*, *local=False*, *M=12*, *N=6*, *ra=None*, *averaged=False*, *switching_function=False*)

Calculate the entropy parameter for each atom

Parameters

- **rm** (*float*) – cutoff distance for integration of entropy parameter in distance units
- **sigma** (*float*) – broadening parameter
- **rstart** (*float, optional*) – minimum limit for integration, default 0.00001
- **h** (*float, optional*) – width for trapezoidal integration, default 0.0001
- **local** (*bool, optional*) – if True, use the local density instead of global density default False
- **averaged** (*bool, optional*) – if True find the averaged entropy parameters default False
- **switching_function** (*bool, optional*) – if True, use the switching function to average, otherwise do a simple average over the neighbors. Default False
- **ra** (*float, optional*) – cutoff length for switching function used only if *switching_function* is True
- **M** (*int, optional*) – power for switching function, default 12 used only if *switching_function* is True
- **N** (*int, optional*) – power for switching function, default 6 used only if *switching_function* is True

Returns

Return type None

Notes

The entropy parameters can be accessed by `entropy` and `avg_entropy`. For a complete description of the entropy parameter, see the documentation

The *local* keyword can be used to use a local density instead of the global one. This method will only work with neighbor methods that use a cutoff.

calculate_q(*q*, *averaged=False*, *only_averaged=False*, *condition=None*, *clear_condition=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
- **only_averaged** (*bool, optional*) – If True, only calculate the averaged part. default False
- **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.
- **clear_condition** (*bool, optional*) – clear the *condition* variable for all atoms

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. If only the averaged versions need to be calculated, *only_averaged* keyword can be set to False.

The neighbors over which the q values are calculated can also be filtered. This is done 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. See *cluster_atoms()* for more details. If the *condition* is equal for both host atom and the neighbor, the neighbor is considered for calculation of q parameters. This is slightly different from *cluster_atoms()* where the condition has to be True for both atoms. *condition* is only cleared when neighbors are recalculated. Additionally, the keyword *clear_condition* can also be used to clear the condition and reset it to 0. By default, *condition* is applied to both unaveraged and averaged q parameter calculation. If *condition* is needed for only averaged q parameters, this function can be called twice, initially without *condition* and *averaged=False*, and then with a condition specified and *averaged=True*. This way, the *condition* will only be applied to the averaged q calculation.

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*, *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.
- **cutoff** (*float, optional*) – If specified, use this cutoff for calculation of clusters. By default uses the cutoff used for neighbor calculation.

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.

embed_in_cubic_box()
Embedded the triclinic box in a cubic box

extract_cubic_box(*repeat=(3, 3, 3)*)
Extract a cubic representation of the box from triclinic cell

Parameters **repeat** (*list of ints*) – the number of times box should be repeat

Returns

- **cubebox** (*list of list of floats*) – cubic box
- **atoms** (*list of Atom objects*) – atoms in the cubic box

find_diamond_neighbors()
Find underlying fcc lattice in diamond

Parameters **None** –

Returns

Return type None

Notes

This method finds in the underlying fcc/hcp lattice in diamond. It works by the method described in this publication . For each atom, 4 atoms closest to it are identified. The neighbors of the its neighbors are further identified and the common neighbors shared with the host atom are selected. These atom will fall in the underlying fcc lattice for cubic diamond or hcp lattice for hexagonal lattice.

If neighbors are previously calculated, they are reset when this method is used.

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', 'type_r'}, 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. If *type_r*, only atoms of a different type will be included in the calculation. 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*, *cutoff=0*, *right=True*)

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.
- **cutoff** (*double, optional*) – Separate value used for cluster classification. If not specified, cutoff used for finding neighbors is used.
- **right** (*bool, optional*) – If true, greater than comparison is to be used for finding solid particles. default True.

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. In the above formulations, > comparison for *threshold* and *avgthreshold* can be changed to < by setting the keyword *right* to False.

If *cluster* is True, a clustering is done for all solid particles. See `find_clusters()` for more details.

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_concentration()
Return a dict containing the concentration of the system

Parameters **None** –

Returns **condict** – dict of concentration values

Return type dict

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.

identify_diamond(*find_neighbors=True*)
Identify diamond structure

Parameters **find_neighbors** (*bool, optional*) – If True, find 4 closest neighbors

Returns **diamondstructure** – dict of structure signature

Return type dict

Notes

Identify diamond structure using the algorithm mentioned in [1]. It is an extended CNA method. The integers 5, 6, 7, 8, 9 and 10 are assigned to the structure variable of the atom. 5 stands for cubic diamond, 6 stands for first nearest neighbors of cubic diamond and 7 stands for second nearest neighbors of cubic diamond. 8 signifies hexagonal diamond, the first nearest neighbors are marked with 9 and second nearest neighbors with 10.

References

iter_atoms()
Iter over atoms

read_inputfile(*filename*, *format='lammps-dump'*, *compressed=False*, *customkeys=None*)
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.
- **customkeys** (*list*) – A list containing names of headers of extra data that needs to be read in from the input file.

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.

Triclinic simulation boxes can also be read in.

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']*

remap_atoms(*remove_images=True*, *assign=True*, *remove_atoms=False*, *dtol=0.1*)
Remap atom back into simulation box

Parameters

- **pbc** (*bool, optional*) – If True, remove atoms on borders that are repeated
- **assign** (*bool, optional*) – If True, assign atoms to the system, otherwise return.
- **remove_atoms** (*bool, optional*) – If True, after the atoms, are remapped, remove those still outside the box.
- **rtol** (*float, optional*) – Tolerance for removing atomic positions. Defaullt 0.1

repeat(*reps*, *atoms=None*, *ghost=False*, *scale_box=True*)
Replicate simulation cell

Parameters

- **reps** (*list of ints of size 3*) – repetitions in each direction
- **atoms** (*list of atoms, optional*) – if not provided, use atoms that are assigned
- **ghost** (*bool, optional*) – If True, assign the new atoms as ghost instead of actual atoms

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.

set_atom_cutoff(*factor=1.0*)
Set cutoff for each atom

Parameters **factor** (*float, optional*) – factor for multiplication of cutoff value. default 1

Returns

Return type None

Notes

Assign cutoffs for each atom based on the nearest neighbor distance. The cutoff assigned is the average nearest neighbor distance multiplied by *factor*.

show(*colorby=None*, *filterby=None*)
Plot the system

Parameters

- **sys** (*System object*) –
- **colorby** (*string, optional*) – property over which the atoms are to be colored. It can be any attributed of Atom, a custom attribute, or calculated q values which can be accessed as *qx* or *aqx* where x stands for the q number.
- **filterby** (*string, optional*) – property over which the atoms are to be filtered before plotting. It can be any attribute of atom, or a custom value of atom. It should provide a True or False value.

Returns

Return type None

to_ase(*species*)
Convert system to an ASE Atoms object

Parameters **species** (*list of string*) – The chemical species

Returns

Return type None

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

Parameters

- **outfile** (*string*) – name of the output file
- **format** (*string, {'lammps-dump', 'lammps-data', 'poscar'}*) – 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.
- **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.
- **compressed** (*bool, optional*) – If true, the output is written as a compressed file.

- **timestep** (*int, optional*) – timestep to be written to file. default 0
- **species** (*None, optional*) – species of the atoms. Required if any format other than 'lammps-dump' is used. Required for convertion to ase object.

Returns

Return type None

Notes

to_file method can handle a number of file formats. The most customizable format is the *lammps-dump* which can take a custom options using customkeys and customvals. customkeys will be the header written to the dump file. It can be any Atom attribute, any property stored in custom variable of the Atom, or calculated q values which can be given by *q4*, *aq4* etc. External values can also be provided using *customvals* option. *customvals* array should be of the same length as the number of atoms in the system.

For all other formats, ASE is used to write out the file, and hence the *species* keyword needs to be specified. If initially, an ASE object was used to create the System, *species* keyword will already be saved, and need not be specified. In other cases, *species* should be a list of atomic species in the System. For example *["Cu"]* or *["Cu", "Al"]*, depending on the number of species in the System. In the above case, atoms of type 1 will be mapped to Cu and of type 2 will be mapped to Al. For a complete list of formats that ASE can handle, see here .

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_energy
Float. Value of averaged energy.

avg_entropy
Float. Value of averaged entropy 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.

bonds

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.

cna

common

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.

energy
Float. Value of energy.

entropy
Float. Value of entropy parameter.

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.

ghost
int. int specifying ghost status of the atom.

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.

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_positions
list of list of floats. Positions of Voronoi vertices. 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** (*{'sc', 'bcc', 'fcc', 'hcp', 'diamond', 'a15' 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.trajectory module

class pyscal.trajectory.**Timeslice**(*trajectory*, *blocklist*)

Bases: object

Timeslice containing info about a single time slice Timeslices can also be added to each

to_ase(*species=None*)

Get block as Ase objects

Parameters **blockno** (*int*) – number of the block to be read, starts from 0

Returns **sys**

Return type ASE object

to_dict()

Get the required block as data

to_file(*outfile*, *mode='w'*)

Get block as outputfile

Parameters

- **outfile** (*string*) – name of output file
- **mode** (*string*) – write mode to be used, optional default "w" write also can be "a" to append.

Returns

Return type None

to_system(*customkeys=None*)
Get block as pyscal system

Parameters **blockno** (*int*) – number of the block to be read, starts from 0

Returns **sys** – pyscal System

Return type *System*

class pyscal.trajectory.**Trajectory**(*filename*)
Bases: object

A Trajectory class for LAMMPS

get_block(*blockno*)
Get a block from the file as raw data

Parameters **blockno** (*int*) – number of the block to be read, starts from 0

Returns **data** – list of strings containing data

Return type list

load(*blockno*)
Load the data of a block into memory as a dictionary of numpy arrays

Parameters **blockno** (*int*) – number of the block to be read, starts from 0

Returns

Return type None

Notes

When the data of a block is loaded, it is accessible through *Trajectory.data[x]*. This data can then be modified. When the block is written out, the modified data is written instead of existing one. But, loaded data is kept in memory until unloaded using *unload* method.

unload(*blockno*)
Unload the data that is loaded to memory using *load* method

Parameters **blockno** (*int*) – number of the block to be read, starts from 0

Returns

Return type None

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_file**(*filename*, *format='lammps-dump'*, *compressed=False*, *customkeys=None*)
Read input file

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.
- **customkeys** (*list*) – A list containing names of headers of extra data that needs to be read in from the input file.

Returns

Return type None

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_file**(*sys*, *outfile*, *format='lammps-dump'*, *compressed=False*, *customkeys=None*, *customvals=None*, *timestep=0*, *species=None*)

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*) – format of the output file
- **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
- **species** (*None, optional*) – species of the atoms. Required if any format other than 'lammps-dump' is used. Required for convertion to ase object.

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

pyscal.misc.**find_tetrahedral_voids**(*infile*, *format='poscar'*, *print_results=True*, *return_system=False*, *direct_coordinates=True*, *precision=0.1*)

Check for tetrahedral voids in the system

Parameters

- **infile** (*string*) – name of the input file
- **format** (*string*) – format of the input file, optional 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
- **direct_coordinates** (*bool, optional*) – if True, results are provided in direct coordinates default False
- **precision** (*int, optional*) – the number of digits to check for distances. default 1

Returns

- **types** (*list of atom types*)
- **volumes** (*list of atom volumes*)
- **pos** (*list of atom positions*)
- **sys** (*system object, returns only if return_sys is True*)

Python Module Index

p

Index

I

L

M

N

P

R

S

T

U

V

W