comp-chem-utils

Ugochukwu Nwosu

Jun 22, 2023

CONTENTS

CHAPTER

ONE

COMPCHEMUTILS

CompChemUtils is a set of tools for computational catalysis workflows.

1.1 Requirements

- Python 3.10 or later
- Click (package for command line interfaces)
- NumPy (N-dimensional array package)
- SciPy (library for scientific computing)
- ASE (tools for atomistic simulations)

1.2 Installation

```
$ pip install comp-chem-utils
```

or, if you use poetry:

```
$ poetry add comp-chem-utils
```

You can also install the in-development version with:

```
$ pip install https://gitlab.com/ugognw/python-comp-chem-utils/-/archive/development/ccu-
↪main.zip
```

or, similarly:

```
$ poetry add git+https://gitlab.com/ugognw/python-comp-chem-utils/-/archive/development/
↪ccu-main.git
```

1.3 Documentation

https://python-comp-chem-utils.readthedocs.io/en/latest

1.4 Testing

To run all the tests run:

```
$ tox
```

Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append` `tox`
Other	`PYTEST_ADDOPTS=--cov-append tox`

1.5 Examples

Determine whether a water molecule is symmetric with respect to a 180 degree rotation about its secondary orientation axis.

```
>>> from ase.build import molecule
>>> from ccu.structure.axisfinder import find_secondary_axis
>>> from ccu.structure.symmetry import Rotation, RotationSymmetry
>>> h2o = molecule('H2O')
>>> axis = find_secondary_axis(h2o)
>>> r = Rotation(180, axis)
>>> sym = RotationSymmetry(r)
>>> sym.check_symmetry(h2o)
True
```

Retrieve reaction intermediates for the two-electron CO2 reduction reaction.

```
>>> from ccu.adsorption.adsorbates import get_adsorbate
>>> cooh = get_adsorbate('COOH_CIS')
>>> cooh.positions
array([[ 0.        ,  0.        ,  0.        ],
       [ 0.98582255, -0.68771934,  0.        ],
       [ 0.        ,  1.343     ,  0.        ],
       [ 0.93293074,  1.61580804,  0.        ]])
>>> ocho =  get_adsorbate('OCHO')
>>> ocho.positions
array([[ 0.        ,  0.        ,  0.        ],
       [ 1.16307212, -0.6715    ,  0.        ],
```

(continues on next page)

(continued from previous page)

```
       [ 0.         ,  1.343      ,  0.         ],
       [-0.95002987, -0.5485     ,  0.         ]])
```

Place adsorbates on a surface (namely, "Cu-THQ.traj") while considering the symmetry of the adsorbate and the adsorption sites.:

```
$ ccu adsorption place-adsorbate CO Cu-THQ.traj orientations/
```

1.6 Enable Shell Completion

Add this to your ~/.bashrc::

```
eval "$(_CCU_COMPLETE=bash_source ccu)"
```

Add this to ~/.zshrc::

```
eval "$(_CCU_COMPLETE=zsh_source ccu)"
```

Add this to ~/.config/fish/completions/ccu.fish::

```
eval (env _CCU_COMPLETE=fish_source ccu)
```

CHAPTER

TWO

INSTALLATION

At the command line:

```
poetry add comp-chem-utils

OR

pip install comp-chem-utils
```

CHAPTER

THREE

USAGE

To use ccu in a project:

```
import ccu
```

CHAPTER

FOUR

REFERENCE

4.1 ccu package

4.1.1 Subpackages

ccu.adsorption package

Submodules

ccu.adsorption.adsorbatecomplex module

Defines the AdsorbateComplex and AdsorbateComplexFactory classes.

class ccu.adsorption.adsorbatecomplex.**AdsorbateComplex**(*site_description: str*, *orientation_description: str*, *structure_desription: str*, *structure: Atoms*)

Bases: `object`

An adsorbate-surface complex.

Variables

- **structure_description** – A string describing the surface structure.
- **site_description** – A string describing the adsorption site.
- **orientation_description** – A string describing the orientation of the adsorbate.
- **structure** – An ase.Atoms object of the adsorbate-surface complex.

write(*destination: Path = None*) → Path

Writes the AdsorbateComplex object to an ASE .traj file.

Parameters
destination – A pathlib.Path instance indicating the directory in which to write the .traj file. Defaults to the current working directory.

Returns
A pathlib.Path instance indicating the path of the written .traj file.

class ccu.adsorption.adsorbatecomplex.**AdsorbateComplexFactory**(*adsorbate: Atoms*, *structure: Atoms*, *separation: float = 1.8*, *special_centres: bool = False*, *symmetric: bool = False*, *vertical: bool = False*)

Bases: object

An AdsorbateComplex factory.

Given an adsorbate, a structure, and various configuration specifications (e.g., "symmetric", "vertical"), an AdsorbateComplexFactory determines all of the adsorption sites and corresponding adsorbate configurations.

Variables

- **_adsorbate** – An ase.Atoms instance representing the adsorbate.
- **_structure** – An ase.Atoms instance representing the surface structure.
- **separation** – How far (in Angstroms) the adsorbate should be placed from the surface.
- **special_centres** – A boolean indicating whether or not atom-centred placement will be used.

  Note that in addition to be set to true, the ase.Atoms instance passed as the adsorbate argument must have the key 'special centres' in its info attribute. Further, this key must map to an iterable whose elements specify the indices of the atoms to be used to centre the adsorbate. If this key is not present in the info attribute, then the atom with index 0 will be used to centre the adsorbate.
- **symmetric** – A boolean indicating whether or not the adsorbate is to be treated as symmetric.
- **vertical** – A boolean indicating whether or not to consider vertical adsorption sites.

property adsorbate: Atoms

adsorbate_orientations(*site:* AdsorptionSite) → list[*ccu.adsorption.adsorbateorientation.AdsorbateOrientation*]

Returns a list of all adsorbate orientations for a given adsorption site.

Parameters

site – A sitefinder.AdsorptionSite instance representing the site for which to generate adsorbate orientations.

next_complex(*site:* AdsorptionSite, *adsorbate_tag: int = -99*) → Iterator[*AdsorbateComplex*]

Yields next adsorbate-surface complex for a given site as an AdsorbateComplex.

Parameters

- **site** – A sitefinder.AdsorptionSite instance which represents the site for which to generate complexes.
- **adsorbate_tag** – An integer with which to tag the adsorbate to enable tracking. Defaults to -99.

orient_adsorbate(*orientation:* AdsorbateOrientation) → Atoms

Orients the AdsorbateComplexFactory's adsorbate such that its primary axis is aligned with the primary orientation vector of the given AdsorbateOrientation object and its secondary axis is in the plane defined by the primary axis of the adsorbate and the secondary orientation.

Parameters

orientation – An adsorbateorientation.AdsorbateOrientation instance representing the orientation in which the adsorbate is to be directed.

Returns

An ase.Atoms instance representing the oriented adsorbate as a copy of the AdsorbateComplexFactory's adsorbate.

place_adsorbate(*adsorbate: Atoms*, *site:* AdsorptionSite, *centre: array = None*)

Moves adsorbate to specified site respecting the minimum specified separation.

Parameters

- **new_adsorbate** – An ase.Atoms instance representing theadsorbate to be moved.
- **centre** – A numpy.array designating the centre with which to align the adsorbate.
- **site** – A sitefinder.AdsorptionSite instance representing the site on which the adsorbate is to be placed.

property structure: Atoms

ccu.adsorption.adsorbatecomplex.**run**(*adsorbate: str*, *structure: Path*, *destination: Path = None*, *separation: float = 1.8*, *special_centres: bool = False*, *symmetric: bool = False*, *vertical: bool = False*)

Creates MOF-adsorbate complexes for adsorption configurations on the SBU of the given MOF and write them to a .traj file.

Parameters

- **adsorbate** – A string indicating the name of the adsorbate to place on the surface.
- **structure** – A pathlib.Path instance indicating the path to the surface on which the adsorbate will be placed.
- **destination** – A pathlib.Path instance indicating the directory in which to write the .traj files. The directory is created if it does not exist. Defaults to the current working directory.
- **separation** – A float indicating how far (in Angstroms) the adsorbate should be placed from the surface. Defaults to 1.8.
- **symmetric** – A boolean indicating whether or not the adsorbate is to be treated as symmetric. Defaults to False.
- **vertical** – A boolean indicating whether or not vertical adsorption configurations are to be generated. Defaults to False.

ccu.adsorption.adsorbateorientation module

This module defines the AdsorbateOrientation and AdsorbateOrientationFactory classes.

class ccu.adsorption.adsorbateorientation.**AdsorbateOrientation**(*description: str*, *orientation_vectors: Sequence[array]*)

Bases: object

An orientation of an adsorbate.

An AdsorbateOrientation object contains the information required to unambiguously orient an adsorbate in space.

Variables

- **description** – A string describing the adsorbate orientation.
- **vectors** – A tuple of numpy.array instances which are the vectors along which an adsorbate will be oriented. The sequence should contain two linearly independent unit vectors. The first vector is the primary orientation axis. The secondary vector is secondary orientation axis.

class ccu.adsorption.adsorbateorientation.**AdsorbateOrientationFactory**(*site: AdsorptionSite*, *adsorbate: Atoms*, *force_symmetry: bool = False*, *vertical: bool = False*)

Bases: object

An AdsorbateOrientation factory.

An AdsorbateOrientationFactory creates a collection of AdsorbateOrientation objects for a given AdsorptionSite subject to symmetry and orientation specifications.

Variables

- **site** – A sitefinder.AdsorptionSite instance indicating site for which the orientations are to be created.
- **adsorbate** – An ase.Atoms instance representing the adsorbate which will assume the orientations.
- **force_symmetry** – A boolean indicating whether or not to force the adsorbate to be treated as symmetric.
- **vertical** – A boolean indicating whether or not vertical orientations will be created.

create_orientations() → list[*ccu.adsorption.adsorbateorientation.AdsorbateOrientation*]

Creates a list of AdsorbateOrientation objects.

Returns
A list of AdsorbateOrientation objects.

ccu.adsorption.adsorbates module

This module defines CO2RR, NRR/UOR, OER/ORR, and HER intermediates.

CO2RR Intermediates from Chem. Rev. 2019, 119, 12, 7610-7672.

NRR/UOR Intermediates from ACS Catal. 2023, 13, 3, 1926-1933. and Angew. Chem. Int. Ed. 2021, 60, 51, 26656.

Bond lengths, angles and positions from cccbdb.nist.gov.

Usage:

```
>>> from ccu.adsorption.adsorbates import get_adsorbate
>>> get_adsorbate('CO2')
Atoms(symbols='CO2', pbc=False)
```

ccu.adsorption.adsorbates.**get_adsorbate**(*adsorbate: str*) → Atoms

Returns the requested adsorbate as an ase.Atoms object.

Parameters
adsorbate – The name of the adsorbate to retrieve as a string (case-insensitive).

Raises
NotImplementedError – The requested adsorbate is neither a molecule supported by ASE nor a defined adsorbate in ccu.adsorption. adsorbates.

Returns
An ase.Atoms instance representing the requested adsorbate.

ccu.adsorption.cli module

This module contains the ccu.structure package CLI logic.

ccu.adsorption.sitefinder module

Defines the AdsorptionSite, SiteFinder, and MOFSiteFinder classes.

class ccu.adsorption.sitefinder.**AdsorptionSite**(*location: Sequence[float]*, *description: str*, *alignments: Iterable[*SiteAlignment*]*, *surface_norm: Sequence[float]*)

Bases: `object`

An adsorption site for an adsorbate.

Variables

- **location** – A numpy.array representing the location of the adsorption site.
- **description** – A description of the adsorption site as a string.
- **alignments** – A list of SiteAlignment objects defining alignments for the site.
- **surface_norm** – A numpy.array representing the unit normal vector for
- **site.** (*the surface hosting the adsorption*) –

class ccu.adsorption.sitefinder.**MOFSite**(*location: Sequence[float]*, *description: str*, *alignment_atoms: Iterable[Atom]*, *site_anchor: Sequence[float]*, *surface_norm: Sequence[float]*, *intermediate_alignments: bool = False*)

Bases: *AdsorptionSite*

An adsorption site within a MOF.

Variables

- **location** – A numpy.array representing the location of the adsorption site.
- **description** – A description of the adsorption site as a string.
- **alignments** – A list of SiteAlignment objects defining alignments for the site.
- **surface_norm** – A numpy.array representing the normal vector for the surface hosting the adsorption site.
- **intermediate_alignments** – A boolean indicating whether or not to consider intermediate alignments.

create_alignments(*alignment_atoms: Iterable[Atom]*, *site_anchor: Sequence[float]*) → list[*ccu.adsorption.sitefinder.SiteAlignment*]

Creates the SiteAlignment objects for a MOFSite.

Parameters

- **alignment_atoms** – An iterable containing ase.Atom instances which will be used to define alignment directions.
- **site_anchor** – A sequence of floats representing a reference location using for defining alignment directions. This is usually the position of the metal atom in the site.

Returns

A list of SiteAlignment instances representing the alignments for a MOFSite instance.

create_intermediate_alignments(*colinear_vectors: Iterable[*SiteAlignment*]*) → list[*ccu.adsorption.sitefinder.SiteAlignment*]

class ccu.adsorption.sitefinder.**MOFSiteFinder**(*structure: Atoms*)

Bases: *SiteFinder*

A SiteFinder subclass which finds adsorption sites on MOF surfaces.

Currently, the atoms bonded to the metal within the SBU must possess tags of 1 and the metal must possess a tag of 2 for the implementation to work correctly.

Parameters
structure – An ase.Atoms object representing a metal-organic framework.

property adjacent_linkers: list[ase.atom.Atom]

A list of ase.Atom instances representing two adjacent linker atoms.

create_between_linker_site() → *MOFSite*

Returns a MOFSite instance representing an adsorption site centred between the MOF linker atoms.

create_linker_sites() → list[*ccu.adsorption.sitefinder.MOFSite*]

Returns a list of MOFSite instances representing adsorption sites centred on the MOF linker atoms.

create_metal_site() → *MOFSite*

Returns a MOFSite instance representing an adsorption site centred on the MOF metal atom.

property sbu_metal: Atom

An ase.Atom instance representing the metal atom within the SBU of the MOF.

sites() → list[*ccu.adsorption.sitefinder.AdsorptionSite*]

Determines all unique SBU adsorption sites for a given MOF.

Note that the AdsorptionSites are defined such that the first and second elements in their "alignment_atoms" attribute are linker atoms and the third element is the metal.

Returns
A list of AdsorptionSite instances representing the SBU adsorption sites of the given MOF.

property surface_norm: array

A unit vector normal to the plane determined by two adjacent linker atoms and the metal within the SBU.

class ccu.adsorption.sitefinder.**SiteAlignment**(*alignment_vector: Sequence[float]*, *description: str*)

Bases: object

An alignment that an adsorbate can assume on a site.

Variables

- **vector** – A numpy.array representing the alignment vector as a unit vector.
- **description** – A string describing the site alignment.

class ccu.adsorption.sitefinder.**SiteFinder**

Bases: ABC

An abstract base class for objects which find adsorption sites for particular surfaces.

Subclasses must define the abstract method "sites" which returns all adsorption sites for a given structure.

abstract sites() → Iterable[*AdsorptionSite*]

Subclasses should override this method.

Module contents

ccu.cli package

Submodules

ccu.cli.main module

Module that contains the command line app.

Why does this file exist, and why not put this in __main__?

> You might be tempted to import things from __main__ later, but that will cause problems: the code will get executed twice:
>
> - When you run *python -m ccu* python will execute `__main__.py` as a script. That means there won't be any `ccu.__main__` in `sys.modules`.
> - When you import __main__ it will get executed again (as a module) because there's no `ccu.__main__` in `sys.modules`.
>
> Also see (1) from https://click.palletsprojects.com/en/5.x/setuptools/ #setuptools-integration

`ccu.cli.main.`**`add_subcommands`**`()`

Module contents

ccu.structure package

Submodules

ccu.structure.axisfinder module

This module defines functions to determine a molecule's orientation axes.

The function get_axes returns all three orientation axes for a given molecule. For example,

```
>>> import ase
>>> from ccu.structure.axisfinder import get_axes
>>> coh = ase.Atoms('COH', positions=[[0, 0, 0], [-2, 0, 0], [-1, 0.5, 0]])
>>> get_axes(coh)
(array([1., 0., 0.]), array([0., 1., 0.]), array([0., 0., 1.]))
```

The function find_farthest_atoms returns the two atoms within a molecule whose separation is the greatest. For example,

```
>>> import ase
>>> from ccu.structure.axisfinder import find_farthest_atoms
>>> coh = ase.Atoms('COH', positions=[[0, 0, 0], [-2, 0, 0], [-1, 0.5, 0]])
>>> find_farthest_atoms(coh)
(Atom('C', [0.0, 0.0, 0.0], index=0), Atom('O', [-2.0, 0.0, 0.0], index=1))
```

The function find_primary_axis returns the primary orientation axis for a given molecule. For example,

```
>>> import ase
>>> from ccu.structure.axisfinder import find_primary_axis
>>> coh = ase.Atoms('COH', positions=[[0, 0, 0], [-2, 0, 0], [-1, 0.5, 0]])
>>> find_primary_axis(coh)
array([1., 0., 0.])
```

The function find_secondary_axis returns the primary orientation axis for a given molecule. For example,

```
>>> import ase
>>> from ccu.structure.axisfinder import find_secondary_axis
>>> coh = ase.Atoms('COH', positions=[[0, 0, 0], [-2, 0, 0], [-1, 0.5, 0]])
>>> find_secondary_axis(coh)
array([0., 1., 0.])
```

The function find_tertiary_axis returns the primary orientation axis for a given molecule. For example,

```
>>> import ase
>>> from ccu.structure.axisfinder import find_tertiary_axis
>>> coh = ase.Atoms('COH', positions=[[0, 0, 0], [-2, 0, 0], [-1, 0.5, 0]])
>>> find_tertiary_axis(coh)
array([0., 0., 1.])
```

ccu.structure.axisfinder.**find_farthest_atoms**(*molecule: Atoms*, *tol: float = 1e-05*) → tuple[ase.atoms.Atoms]

Finds the two atoms in the molecule separated by the greatest distance.

In molecules for which there are several pairs of atoms with equidistant separations, this function will return the pair of atoms with lowest indices whose separation is within a given tolerance of the largest atomic separation in the molecule. Each pair is sorted according to the index of the lowest index atom and then the index of the second atom. For .. rubric:: Example

- If atoms 0 and 1 have the same separation as atoms 2 and 3, atoms 0 and 1 will be returned since 0 < 2.
- If atoms 0 and 1 have the same separation as atoms 0 and 3, atoms 0 and 1 will be returned since 1 < 3.
- If atoms 1 and 2 have the same separation as atoms 0 and 4, atoms 0 and 4 will be returned since 0 < 1.
- If atoms 1 and 2 have the same separation as atoms 0 and 2, atoms 0 and 2 will be returned since 0 < 1.

Parameters

- **molecule** – The molecule for whom the two farthest atoms are to be determined.
- **tol** – A float indicating the resolution (in Angstroms) between atomic distances.

Returns

A tuple containing the two atoms in the molecule separated by the greatest distance. The atoms are ordered by lowest index within the structure.

ccu.structure.axisfinder.**find_primary_axis**(*molecule: Atoms*) → array

Determines the unit vector representing the primary orientation axis of a molecule.

The primary axis is defined as the unit vector which is parallel to the direction vector between the two most distant atoms in the molecule and points from the higher index atom to the lower index atom.

Parameters

molecule – An ase.Atoms instance representing the molecule for whom the primary axis is to be determined.

Returns
A numpy.array representing a unit vector in the direction of the primary orientation axis. Note that for zero-dimensional molecules, this function will return the zero vector.

ccu.structure.axisfinder.**find_secondary_axis**(*molecule: Atoms*, *min_distance: float = 0.1*) → array

Determines the unit vector representing the secondary orientation axis of a molecule.

Let L be the line between the two farthest atoms in the molecule, let v be the vector which defines the primary axis, and let P be the position of the atom farthest from L. Further, let w be the vector from L to P, and let z be the component of w which is orthogonal to v. The secondary axis is defined as the unit vector in the direction of z.

Parameters

- **molecule** – An ase.Atoms instance representing the molecule for whom the secondary axis is to be determined.
- **min_distance** – A float specifying the minimum distance from the primary axis (in Angstroms) to be considered for defining the secondary axis. Defaults to 0.1.

Returns
A numpy.array representing a unit vector in the direction of the secondary orientation axis. Note that for zero- and one-dimensional molecules, this function will return the zero vector.

ccu.structure.axisfinder.**find_tertiary_axis**(*molecule: Atoms*) → array

Determines the unit vector representing the tertiary orientation axis of a molecule.

The tertiary orientation axis is simply the cross product of the primary and secondary orientation axes. See find_primary_axis and find_secondary_axis for information on how these axes are defined.

Parameters
molecule – An ase.Atoms instance representing the molecule for whom the tertiary axis is to be determined.

Returns
A numpy.array representing a unit vector in the direction of the tertiary orientation axis. Note that if the molecule is zero- or one-dimensional, this function will return the zero vector.

ccu.structure.axisfinder.**get_axes**(*molecule: Atoms*) → tuple[numpy.array]

Determines a molecule's three orientation axes.

The primary axis is defined as the vector between the two most distant atoms. The secondary axis is defined as the orthogonal component (to the primary axis) of the vector from the primary axis to the atom farthest from the line between the two most distant atoms. The tertiary axis is the cross product of the primary and secondary axes. The axes so defined are orthogonal. Note that if the molecule is unimolecular, all three vectors will be the zero vector, and that if the molecule is linear only the primary axis will be nonzero.

Parameters
molecule – An ase.Atoms instance whose axes are to be determined.

Returns
A tuple containing unit vectors reprsenting the three orientation axes. The first, second, and third entries are the primary, secondary, and tertiary axes, respectively. For nonlinear molecules, the axes form an orthonormal set.

ccu.structure.cli module

This module contains the ccu.structure package CLI logic.

ccu.structure.comparator module

This module defines the Comparator class.

The Comparator class can be used to determine teh similarity of two structures as follows:

```
>>> import ase
>>> from ccu.structure.comparator import Comparator
>>> co1 = ase.Atoms('CO', positions=[[0, 0, 0], [1, 0, 0]])
>>> co2 = ase.Atoms('CO', positions=[[0, 1, 1], [1, 1, 1]])
>>> oc = ase.Atoms('OC', positions=[[0, 0, 0], [1, 0, 0]])
>>> Comparator.check_similarity(co1, co2)
True
>>> Comparator.check_similarity(co1, oc)
False
```

class ccu.structure.comparator.**Comparator**

Bases: `object`

An object which compares the similarity of two structures.

static calculate_cumulative_displacement(*fingerprint1:* Fingerprint, *fingerprint2:* Fingerprint) → float

Calculates the cumulative displacement of each atomic position in fingerprint2 relative to the corresponding atomic position in fingerprint1.

The cumulative displacement is defined as follows:

Note that each row in each np.ndarray associated with each histogram key corresponds to a displacement vector between two atoms. With each such displacement vector in the histogram of fingerprint1, we can identify a corresponding displacement vector in the histogram of fingerprint2 as the displacement vector associated with the same histogram key and index. We then define a difference vector as the difference between a displacement vector in fingerprint1 and its counterpart in fingerprint2. The set of all difference vectors is defined on the basis of fingerprint1. That is, if X is the set of all displacement vectors in fingerprint1 and Y is the set of all corresponding vectors in fingerprint2, the set of all difference vectors is the set of all vectors x - y where x is a displacement vector in fingerprint1 and y is the corresponding displacement vector in Y. (Note that this requires that the histogram of fingerprint2 must include all the keys that that of the histogram of fingerprint1 includes. Additionally, this requires that for each key in the histogram of fingerprint1, the value in fingerprint2 includes at least as many displacement vectors as the value in fingerprint1.) The cumulative displacement is then defined as the sum of the norms of all the difference vectors corresponding to fingerprint1 and fingerprint2.

Parameters

- **fingerprint1** – The Fingerprint instance used as a reference to calculate the cumulative displacement.
- **fingerprint2** – The second Fingerprint instance used to calculate the cumulative displacement.

Returns

A float representing the cumulative displacement for fingerprint2 relative to fingerprint1.

static check_similarity(*structure1: Atoms*, *structure2: Atoms*, *tol: float = 0.05*) → bool

Determines whether the atomic positions of two structures are similar to within a given tolerance.

Parameters

- **structure1** – An ase.Atoms instance representing the first structure to compare.
- **structure2** – An ase.Atoms instance representing the second structure to compare.
- **tol** – A float specifying the tolerance for the cumulative displacement for fingerprint in Angstroms. Defaults to 5e-2.

Returns

A boolean indicating whether or not the two structures are similar within the specified tolerance. Two structures are similar if they can be superimposed via a translation operation.

static cosort_fingerprints(*fingerprints1: Iterable[*Fingerprint*]*, *fingerprints2: Iterable[*Fingerprint*]*) → tuple[*ccu.structure.fingerprint.Fingerprint*]

Determines the ordering of the second supplied iterable of Fingerprints which minimizes the cumulative displacement across the two iterables of Fingerprints.

Parameters

- **fingerprints1** – An iterable containing Fingerprint instances.
- **fingerprints2** – An iterable containing Fingerprint instances.
- **Note that the two iterables must be of the same length and that the**
- **values() methods of all Fingerprint instances across the two**
- **iterables must be of the same length.**

Returns

A tuple containing the ordering of fingerprints2 which minimizes the cumulative displacement across the two iterables of Fingerprints.

static cosort_histograms(*fingerprint1:* Fingerprint, *fingerprint2:* Fingerprint) → dict[str, numpy.ndarray]

Determines the ordering of the second fingerprint's histogram which minimizes the cumulative displacement of the atoms in each structure.

The two supplied Fingerprints need not have the same keys or the same number of entries under each key. Such cases are handled as follows:

Let k be a key in both the histograms of fingerprint1 and fingerprint2. Let p be the iterable corresponding to the key k in the histogram of fingerprint1, and let q be the iterable corresponding to the key k in the histogram of fingerprint2.

If len(p) > len(q), then q is ordered according to its match with the first len(q) elements of p.

If len(p) <= len(q), then q is ordered according to the best match with p and the first len(p) elements of q.

Parameters

- **fingerprint1** – The Fingerprint object to be used as a reference for each displacement in the other Fingerprint's histogram.
- **fingerprint2** – The Fingerprint object for which the optimally ordered histogram is to be determined.

Returns

A dict constructed from fingerprint2._histogram mapping chemical symbols to a

numpy.ndarray containing the displacement vectors to atoms with the corresponding chemical symbol. The order of the displacement vectors is such that the cumulative displacement of the displacement vectors is minimized relative to fingerprint1._histogram.

ccu.structure.fingerprint module

This module defines the Fingerprint class.

class ccu.structure.fingerprint.**Fingerprint**(*structure: Atoms*, *reference: int*, *indices: Iterable[int] = None*)

Bases: MutableMapping

A set of displacement vectors relative to a particular atom within an ase.Atoms object.

The displacement vectors for atoms of a given chemical symbol can be accessed through the MutableMapping interface. For example:

structure = ase.Atoms('CO', positions=[[0, 0, 0], [1, 0, 0]]) fp = Fingerprint(structure, 0, [0, 1]) fp['C']

Variables

- **structure** – The ase.Atoms instance to which the Fingerprint instance is related. reference: An int indicating the index of the reference atom used to construct the Fingerprint instance.
- **indices** – A tuple indicating the indices of the atoms within the structure used to construct the Fingerprint instance.

classmethod from_structure(*structure: Atoms*) → list[*ccu.structure.fingerprint.Fingerprint*]

Creates a list of Fingerprint objects corresponding to each atom within an ase.Atoms object.

Parameters
structure – An ase.Atoms instance representing the structure from which to create the list of Fingerprints.

Returns
A list of the Fingerprints for each atom.

ccu.structure.geometry module

This module defines useful geometry related functions for ase.Atoms instances.

ccu.structure.geometry.**calculate_separation**(*structure1: Atoms*, *structure2: Atoms*) → float

Calculates the separation between two ase.Atoms instances defined as the smallest distance between an atom in one structure and an atom in the second structure.

Parameters

- **structure1** – An ase.Atoms instance.
- **structure2** – An ase.Atoms instance.

Returns
A float representing the separation between the two structures.

ccu.structure.resizecell module

This script resizes the c vector of all the .traj files in the current working directory to the specified positive number (default is 10)

ccu.structure.resizecell.**run**(*structure: Path*, *length: float*)

Resize c-vector of structure and centres atoms in cell.

Parameters

- **structure** – A pathlib.Path instance leading to the structure whose cell is to be resized.
- **length** – A float specifying the new c-vector of the cell.

ccu.structure.symmetry module

This class defines the SymmetryOperation and Symmetry classes and subclasses.

Symmetry and SymmetryOperation subclasses can be used as follows:

```
>>> import ase
>>> from ccu.structure.symmetry import Rotation, RotationSymmetry
>>> rotation1 = Rotation(90, [0, 0, 1])
>>> symmetry1 = RotationSymmetry(rotation1)
>>> co = ase.Atoms('CO', positions=[[0, 0, 0], [1, 0, 0]])
>>> rotated = rotation1.transform(co)
>>> rotated.positions
array([[0.000000e+00, 0.000000e+00, 0.000000e+00],
       [6.123234e-17, 1.000000e+00, 0.000000e+00]])
>>> symmetry1.check_symmetry(co)
False
>>> h2 = ase.Atoms('HH', positions=[[0, 0, 0], [1, 0, 0]])
>>> rotation2 = Rotation(180, [0, 0, 1])
>>> symmetry2 = RotationSymmetry(rotation2)
>>> symmetry2.check_symmetry(h2)
True
```

class ccu.structure.symmetry.**Inversion**

Bases: *SymmetryOperation*

class ccu.structure.symmetry.**Rotation**(*angle: float*, *axis: Iterable[float]*)

Bases: *SymmetryOperation*

A rotation operation.

Variables

- **angle** – A float specifying a rotation angle in degrees.
- **axis** – A numpy.array representing the axis of rotation.

as_matrix() → ndarray

Returns the rotation operation of this instance as a numpy.ndarray which represents the rotation matrix.

transform(*structure: Atoms*) → Atoms

Rotates the given structure by the angle and about the axis specified as attributes of the Rotation object.

Parameters
structure – An ase.Atoms instance representing structure to be rotated.

Returns
A rotated copy of the original ase.Atoms instance.

class ccu.structure.symmetry.**RotationSymmetry**(*operation:* Rotation)

Bases: *Symmetry*

A rotational symmetry.

check_symmetry(*structure: Atoms*, *tol: float = 0.05*) → bool

Determines the symmetry represented by the instance belongs to the given structure.

Parameters

- **structure** – An ase.Atoms instance representing the structure whose symmetry is to be determined.
- **tol** – A float specifying the absolute tolerance for positions. Defaults to 5e-2.

Returns

A boolean indicating whether or not the given structure possesses
the symmetry of the RotationSymmetry object subject to the specified tolerance.

property operation: *Rotation*

The Rotation instance associated with this RotationSymmetry instance.

class ccu.structure.symmetry.**Symmetry**

Bases: ABC

An abstract base class for molecule symmetries.

abstract check_symmetry(*structure: Atoms*, *tol: float*) → bool

Subclasses should override this method.

abstract property operation: *SymmetryOperation*

Subclasses should override this method.

class ccu.structure.symmetry.**SymmetryOperation**

Bases: ABC

An abstract base class for symmetry operations.

abstract transform(*structure: Atoms*) → Atoms

Subclasses should override this method.

Module contents

4.1.2 Module contents

Utilities for computational catalysis.

4.2 ccu

CHAPTER

FIVE

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.2 Documentation improvements

CompChemUtils could always use more documentation, whether as part of the official CompChemUtils docs, in docstrings, or even on the web in blog posts, articles, and such.

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://gitlab.com/ugognw/python-comp-chem-utils/-/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

5.4 Development

To set up *CompChemUtils* for local development:

1. Fork CompChemUtils (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@gitlab.com:YOURGITLABNAME/ccu.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes run all the checks and docs builder with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitLab website.

5.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`).
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

5.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel*:

```
tox -p auto
```

CHAPTER

SIX

AUTHORS

- Ugochukwu Nwosu - https://www.law-two.com

CHAPTER

SEVEN

CHANGELOG

7.1 0.0.1 (2023-06-22)

- First release on PyPI.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

c

s

INDEX