spyrmsd Documentation

spyrmsd

Apr 15, 2024

CONTENTS:

`spyrmsd` is a Python tool for symmetry-corrected RMSD calculations.

CHAPTER

ONE

INSTALLATION

`spyrmsd` is available on PyPI and conda-forge and can be easily installed from source.

1.1 Installation

Installing `spyrmsd` with `pip`, `conda` or from source will install the package as a library. In order to install the package as a standalone tool, Open Babel or RDKit need to be installed as well (see *Dependencies*).

1.1.1 pip

```
pip install spyrmsd
```

1.1.2 conda

```
conda install spyrmsd -c conda-forge
```

1.1.3 GitHub

```
git clone https://github.com/RMeli/spyrmsd.git
cd spyrmsd
pip install .
```

1.2 Dependencies

`spyrmsd` can be used both as a module or as a standalone tool.

1.2.1 Module

The following packages are required to use `spyrmsd` as a module:

- graph-tool or NetworkX
- numpy
- scipy

Note: `spyrmsd` uses graph-tool by default but will fall back to NetworkX if the former is not installed (e.g. on Windows).

1.2.2 Standalone Tool

Additionally, one of the following packages is required to use `spyrmsd` as a standalone tool:

- Open Babel
- RDKit

CHAPTER

TWO

TUTORIAL

```
import spyrmsd
from spyrmsd import io, rmsd
```

2.1 OpenBabel or RDKit

`spyrmsd` natively supports OpenBabel and RDKit to load molecules in order to work as a standalone tool. However, the API for RMSD calculations is extremely minimal and only needs the following information:

- Atomic coordinates
- Atomic numbers
- Molecular adjacency matrix (for symmetry)

This means that `spyrmsd` can be used in combination with any library that can provide such information.

2.2 Loading Molecules

The `spyrmsd.io` module provides functions to easily load molecules from a file and to transform them into a `spyrmsd.molecule.Molecule` object:

```
ref = io.loadmol("molecules/1a4k_ligand.sdf")
```

`io.loadmol` load a single molecule from a file. In order to load all molecules we need to use `io.loadallmols`:

```
mols = io.loadallmols("molecules/1a4k_dock.sdf")
```

2.3 Loading RDKit or OpenBabel Molecules

`spyrmsd` natively supports Open Babel and RDKit (if installed). The `Molecule` class provides `from_openbabel()` and `from_rdkit()` constructors.

```
from rdkit import Chem
from rdkit.Chem import AllChem

rdmol1 = Chem.MolFromSmiles("c1ccccc1")
```

(continues on next page)

(continued from previous page)

```
rdmol2 = Chem.MolFromSmiles("c1ccccc1")
AllChem.EmbedMolecule(rdmol1)
AllChem.EmbedMolecule(rdmol2)

from spyrmsd.molecule import Molecule
from spyrmsd.rmsd import rmsdwrapper

mol1 = Molecule.from_rdkit(rdmol1)
mol2 = Molecule.from_rdkit(rdmol2)

rmsdwrapper(mol1, mol2)
```

```
<frozen importlib._bootstrap>:241: RuntimeWarning: to-Python converter for std::__1::pair
↪<double, double> already registered; second conversion method ignored.
[21:58:01] Molecule does not have explicit Hs. Consider calling AddHs()
[21:58:01] Molecule does not have explicit Hs. Consider calling AddHs()
```

```
[0.019162902039384797]
```

2.4 Removing Hydrogen Atoms

Hydrogen atoms can be removed with the `strip()` function:

```
ref.strip()
```

```
for mol in mols:
    mol.strip()
```

2.5 Symmetry-Corrected RMSD

`spyrmsd` only needs atomic coordinates, atomic number and the molecular adjacency matrix to compute the standard RMSD with `spyrmsd.rmsd.symmrmsd`. The `spyrmsd.molecule.Molecule` class provides easy access to such information:

```
coords_ref = ref.coordinates
anum_ref = ref.atomicnums
adj_ref = ref.adjacency_matrix
```

```
coords = [mol.coordinates for mol in mols]
anum = mols[0].atomicnums
adj = mols[0].adjacency_matrix
```

With this information we can easily compute the RMSD between the reference molecule and all other molecules:

```
RMSD = rmsd.symmrmsd(
    coords_ref,
    coords,
```

(continues on next page)

(continued from previous page)

```
    anum_ref,
    anum,
    adj_ref,
    adj,
)

print(RMSD)
```

```
[2.0246085732404446, 1.4951562971486378, 10.028009301306854, 7.900570020309068, 7.
↪578344354783399, 9.52999506817054, 4.952371789159667, 7.762808670066815, 9.
↪996922964463582, 7.1732072690335755]
```

2.5.1 Minimum RMSD

We can also compute the minimum RMSD obtained by superimposing the molecular structures:

```
RMSD = rmsd.symmrmsd(
    coords_ref,
    coords,
    anum_ref,
    anum,
    adj_ref,
    adj,
    minimize=True,
)

print(RMSD)
```

```
[1.2012368667355435, 1.0533413220699535, 1.153253104575529, 1.036542688936588, 0.
↪8407673221224187, 1.1758143217869736, 0.7817315189656655, 1.0933314311267845, 1.
↪0260767175206462, 0.9586369647000478]
```

2.6 Change Backend

`spyrmsd` supports multiple backends. You see which backends are available by looking at the `available_backends` attribute:

```
spyrmsd.available_backends
```

```
['graph_tool', 'networkx']
```

The available backends are a subset of the supported backends. Only the backends that are installed will be available.

You can check the current backend with

```
spyrmsd.get_backend()
```

```
'graph_tool'
```

You can switch the backend using

```
spyrmsd.set_backend("networkx")
spyrmsd.get_backend()
```

```
'networkx'
```

CHAPTER

THREE

API

3.1 spyrmsd package

Python RMSD tool with symmetry correction.

3.1.1 Subpackages

spyrmsd.graphs package

Submodules

spyrmsd.graphs.gt module

spyrmsd.graphs.gt.**cycle**(*n*)

Build cycle graph

Parameters
n (*int*) – Number of nodes

Returns
Cycle graph

Return type
Graph

spyrmsd.graphs.gt.**graph_from_adjacency_matrix**(*adjacency_matrix: Union[ndarray, List[List[int]]]*, *aprops: Optional[Union[ndarray, List[Any]]] = None*)

Graph from adjacency matrix.

Parameters

- **adjacency_matrix** (*Union[np.ndarray, List[List[int]]]*) – Adjacency matrix
- **aprops** (*Union[np.ndarray, List[Any]], optional*) – Atomic properties

Returns
Molecular graph

Return type
Graph

Notes

It the atomic numbers are passed, they are used as node attributes.

spyrmsd.graphs.gt.**lattice**(*n1: int*, *n2: int*)

Build 2D lattice graph

Parameters

- **n1** (*int*) – Number of nodes in dimension 1
- **n2** (*int*) – Number of nodes in dimension 2

Returns
Lattice graph

Return type
Graph

spyrmsd.graphs.gt.**match_graphs**(*G1*, *G2*) → List[Tuple[List[int], List[int]]]

Compute graph isomorphisms.

Parameters

- **G1** – Graph 1
- **G2** – Graph 2

Returns
All possible mappings between nodes of graph 1 and graph 2 (isomorphisms)

Return type
List[Tuple[List[int],List[int]]]

Raises
NonIsomorphicGraphs – If the graphs *G1* and *G2* are not isomorphic

spyrmsd.graphs.gt.**num_edges**(*G*) → int

Number of edges

Parameters
G – Graph

Returns
Number of edges

Return type
int

spyrmsd.graphs.gt.**num_vertices**(*G*) → int

Number of vertices

Parameters
G – Graph

Returns
Number of vertices (nodes)

Return type
int

spyrmsd.graphs.gt.**vertex_property**(*G*, *vproperty: str*, *idx: int*) → Any

Get vertex (node) property from graph

Parameters

- **G** – Graph
- **vproperty** (*str*) – Vertex property name
- **idx** (*int*) – Vertex index

Returns
Vertex property value

Return type
Any

spyrmsd.graphs.nx module

spyrmsd.graphs.nx.**cycle**(*n*)

Build cycle graph

Parameters
n (*int*) – Number of nodes

Returns
Cycle graph

Return type
Graph

spyrmsd.graphs.nx.**graph_from_adjacency_matrix**(*adjacency_matrix: Union[ndarray, List[List[int]]]*, *aprops: Optional[Union[ndarray, List[Any]]] = None*) → Graph

Graph from adjacency matrix.

Parameters

- **adjacency_matrix** (*Union[np.ndarray, List[List[int]]]*) – Adjacency matrix
- **aprops** (*Union[np.ndarray, List[Any]], optional*) – Atomic properties

Returns
Molecular graph

Return type
Graph

Notes

It the atomic numbers are passed, they are used as node attributes.

spyrmsd.graphs.nx.**lattice**(*n1*, *n2*)

Build 2D lattice graph

Parameters

- **n1** (*int*) – Number of nodes in dimension 1
- **n2** (*int*) – Number of nodes in dimension 2

Returns
Lattice graph

Return type
Graph

spyrmsd.graphs.nx.**match_graphs**(*G1*, *G2*) → List[Tuple[List[int], List[int]]]

Compute graph isomorphisms.

Parameters

- **G1** – Graph 1
- **G2** – Graph 2

Returns
All possible mappings between nodes of graph 1 and graph 2 (isomorphisms)

Return type
List[Tuple[List[int],List[int]]]

Raises
NonIsomorphicGraphs – If the graphs *G1* and *G2* are not isomorphic

spyrmsd.graphs.nx.**num_edges**(*G*) → int

Number of edges

Parameters
G – Graph

Returns
Number of edges

Return type
int

spyrmsd.graphs.nx.**num_vertices**(*G*) → int

Number of vertices

Parameters
G – Graph

Returns
Number of vertices (nodes)

Return type
int

spyrmsd.graphs.nx.**vertex_property**(*G*, *vproperty: str*, *idx: int*) → Any

Get vertex (node) property from graph

Parameters

- **G** – Graph
- **vproperty** (*str*) – Vertex property name
- **idx** (*int*) – Vertex index

Returns
Vertex property value

Return type
Any

spyrmsd.optional package

Submodules

spyrmsd.optional.obabel module

spyrmsd.optional.obabel.**adjacency_matrix**(*mol*) → ndarray

Adjacency matrix from OpenBabel molecule.

Parameters
mol – Molecule

Returns
Adjacency matrix of the molecule

Return type
np.ndarray

spyrmsd.optional.obabel.**bonds**(*mol*) → List[Tuple[int, int]]

List of bonds

Parameters
mol – Molecule

Returns
List of bonds

Return type
List[Tuple[int, int]]

Notes

A bond is defined by a tuple of (0-based) indices of two atoms.

spyrmsd.optional.obabel.**load**(*fname: str*)

Load molecule from file.

Parameters
fname (*str*) – File name

Return type
Molecule

spyrmsd.optional.obabel.**loadall**(*fname: str*)

Load molecules from file.

Parameters
fname (*str*) – File name

Return type
List of molecules

spyrmsd.optional.obabel.**numatoms**(*mol*) → int

Number of atoms.

Parameters
mol – Molecule

Returns
Number of atoms

Return type
int

spyrmsd.optional.obabel.**numbonds**(*mol*) → int

Number of bonds.

Parameters
mol – Molecule

Returns
Number of bonds

Return type
int

spyrmsd.optional.obabel.**to_molecule**(*mol*, *adjacency: bool = True*)

Transform molecule to *pyrmsd* molecule.

Parameters

- **mol** – Molecule
- **adjacency** (*boolean, optional*) – Flag to decide wether to build the adjacency matrix from molecule

Returns
pyrmsd molecule

Return type
pyrmsd.molecule.Molecule

spyrmsd.optional.rdkit module

spyrmsd.optional.rdkit.**adjacency_matrix**(*mol*) → ndarray

Adjacency matrix from OpenBabel molecule.

Parameters
mol – Molecule

Returns
Adjacency matrix of the molecule

Return type
np.ndarray

spyrmsd.optional.rdkit.**bonds**(*mol*) → List[Tuple[int, int]]

List of bonds.

Parameters
mol – Molecule

Returns
List of bonds

Return type
List[Tuple[int, int]]

Notes

A bond is defined by a tuple of (0-based) indices of two atoms.

spyrmsd.optional.rdkit.**load**(*fname: str*)

Load molecule from file.

Parameters
fname (*str*) – File name

Return type
Molecule

spyrmsd.optional.rdkit.**loadall**(*fname: str*)

Load molecules from file.

Parameters
fname (*str*) – File name

Return type
List of molecules

spyrmsd.optional.rdkit.**numatoms**(*mol*) → int

Number of atoms.

Parameters
mol – Molecule

Returns
Number of atoms

Return type
int

spyrmsd.optional.rdkit.**numbonds**(*mol*) → int

Number of bonds.

Parameters
mol – Molecule

Returns
Number of bonds

Return type
int

spyrmsd.optional.rdkit.**to_molecule**(*mol*, *adjacency: bool = True*)

Transform molecule to *pyrmsd* molecule.

Parameters

- **mol** – Molecule
- **adjacency** (*boolean, optional*) – Flag to decide wether to build the adjacency matrix from molecule

Returns
spyrmsd molecule

Return type
spyrmsd.molecule.Molecule

3.1.2 Submodules

spyrmsd.constants module

Useful constants.

Notes

Periodic table data (atomic masses and covalent radii) are extracted from QCElemental

spyrmsd.due module

Stub file for a guaranteed safe import of duecredit constructs: if duecredit is not available.

To use it, place it into your project codebase to be imported, e.g. copy as

    cp stub.py /path/tomodule/module/due.py

Note that it might be better to avoid naming it duecredit.py to avoid shadowing installed duecredit.

Then use in your code as

    from .due import due, Doi, BibTeX, Text

See https://github.com/duecredit/duecredit/blob/master/README.md for examples.

Origin: Originally a part of the duecredit Copyright: 2015-2019 DueCredit developers License: BSD-2

`spyrmsd.due.BibTeX(*args, **kwargs)`
    Perform no good and no bad

`spyrmsd.due.Doi(*args, **kwargs)`
    Perform no good and no bad

`class spyrmsd.due.InactiveDueCreditCollector`
    Bases: `object`

    Just a stub at the Collector which would not do anything

    `activate(*args, **kwargs)`
        Perform no good and no bad

    `active = False`

    `add(*args, **kwargs)`
        Perform no good and no bad

    `cite(*args, **kwargs)`
        Perform no good and no bad

    `dcite(*args, **kwargs)`
        If I could cite I would

    `dump(*args, **kwargs)`
        Perform no good and no bad

    `load(*args, **kwargs)`
        Perform no good and no bad

spyrmsd.due.**Text**(**args*, ***kwargs*)

Perform no good and no bad

spyrmsd.due.**Url**(**args*, ***kwargs*)

Perform no good and no bad

spyrmsd.exceptions module

exception spyrmsd.exceptions.**NonIsomorphicGraphs**

Bases: ValueError

Raised when graphs are not isomorphic

spyrmsd.graph module

spyrmsd.graph.**adjacency_matrix_from_atomic_coordinates**(*aprops: ndarray*, *coordinates: ndarray*) → ndarray

Compute adjacency matrix from atomic coordinates.

Parameters

- **aprops** (*numpy.ndarray*) – Atomic properties
- **coordinates** (*numpy.ndarray*) – Atomic coordinates

Returns
Adjacency matrix

Return type
numpy.ndarray

Notes

This function is based on an automatic bond perception algorithm: two atoms are considered to be bonded when their distance is smaller than the sum of their covalent radii plus a tolerance value.[3]

Warning: The automatic bond perceptron rule implemented in this functions is very simple and only depends on atomic coordinates. Use with care!

spyrmsd.hungarian module

spyrmsd.hungarian.**cost_mtx**(*A: ndarray*, *B: ndarray*)

Compute the cost matrix for atom-atom assignment.

Parameters

- **A** (*numpy.ndarray*) – Atomic coordinates of molecule A
- **B** (*numpy.ndarray*) – Atomic coordinates of molecule B

Returns
Cost matrix of squared atomic distances between atoms of molecules A and B

[3] E. C. Meng and R. A. Lewis, *Determination of molecular topology and atomic hybridization states from heavy atom coordinates*, J. Comp. Chem. **12**, 891-898 (1991).

Return type
np.ndarray

spyrmsd.hungarian.**hungarian_rmsd**(*A: ndarray*, *B: ndarray*, *apropsA: ndarray*, *apropsB: ndarray*) → float

Solve the optimal assignment problems between atomic coordinates of molecules A and B.

Parameters

- **A** (*numpy.ndarray*) – Atomic coordinates of molecule A
- **B** (*numpy.ndarray*) – Atomic coordinates of molecule B
- **apropsA** (*numpy.ndarray*) – Atomic properties of molecule A
- **apropsB** (*numpy.ndarray*) – Atomic properties of molecule B

Returns
RMSD computed with the Hungarian method

Return type
float

Notes

The Hungarian algorithm is used to solve the linear assignment problem, which is a minimum weight matching of the molecular graphs (bipartite).

The linear assignment problem is solved for every element separately.[1]

spyrmsd.hungarian.**optimal_assignment**(*A: ndarray*, *B: ndarray*)

Solve the optimal assignment problems between atomic coordinates of molecules A and B.

Parameters

- **A** (*numpy.ndarray*) – Atomic coordinates of molecule A
- **B** (*numpy.ndarray*) – Atomic coordinates of molecule B

Returns
Cost of the optimal assignment, together with the row and column indices of said assignment

Return type
Tuple[float, nd.array, nd.array]

spyrmsd.io module

spyrmsd.io.**adjacency_matrix**(*mol*) → ndarray

Adjacency matrix from OpenBabel molecule.

Parameters
mol – Molecule

Returns
Adjacency matrix of the molecule

Return type
np.ndarray

[1] W. J. Allen and R. C. Rizzo, *Implementation of the Hungarian Algorithm to Account for Ligand Symmetry and Similarity in Structure-Based Design*, J. Chem. Inf. Model. **54**, 518-529 (2014)

spyrmsd.io.**bonds**(*mol*) → List[Tuple[int, int]]

List of bonds

Parameters
mol – Molecule

Returns
List of bonds

Return type
List[Tuple[int, int]]

Notes

A bond is defined by a tuple of (0-based) indices of two atoms.

spyrmsd.io.**load**(*fname: str*)

Load molecule from file.

Parameters
fname (*str*) – File name

Return type
Molecule

spyrmsd.io.**loadall**(*fname: str*)

Load molecules from file.

Parameters
fname (*str*) – File name

Return type
List of molecules

spyrmsd.io.**numatoms**(*mol*) → int

Number of atoms.

Parameters
mol – Molecule

Returns
Number of atoms

Return type
int

spyrmsd.io.**numbonds**(*mol*) → int

Number of bonds.

Parameters
mol – Molecule

Returns
Number of bonds

Return type
int

spyrmsd.io.**to_molecule**(*mol*, *adjacency: bool = True*)

Transform molecule to *pyrmsd* molecule.

Parameters

- **mol** – Molecule
- **adjacency** (*boolean, optional*) – Flag to decide wether to build the adjacency matrix from molecule

Returns
pyrmsd molecule

Return type
pyrmsd.molecule.Molecule

spyrmsd.molecule module

class spyrmsd.molecule.**Molecule**(*atomicnums: Union[ndarray, List[int]]*, *coordinates: Union[ndarray, List[List[float]]]*, *adjacency_matrix: Optional[Union[ndarray, List[List[int]]]] = None*)

Bases: `object`

center_of_geometry() → ndarray

Center of geometry.

Returns
Center of geometry

Return type
np.ndarray

center_of_mass() → ndarray

Center of mass.

Returns
Center of mass

Return type
np.ndarray

Notes

Atomic masses are cached.

classmethod from_obabel(*obmol*, *adjacency: bool = True*)

Constructor from OpenBabel molecule.

Parameters

- **obmol** – OpenBabel molecule
- **adjacency** – Flag to compute the adjacency matrix

Returns
`spyrmsd` Molecule

Return type
spyrmsd.molecule.Molecule

classmethod from_rdkit(*rdmol*, *adjacency: bool = True*)

Constructor from RDKit molecule.

Parameters

- **rdmol** – RDKit molecule
- **adjacency** – Flag to compute the adjacency matrix

Returns

spyrmsd Molecule

Return type

spyrmsd.molecule.Molecule

rotate(*angle: float*, *axis: Union[ndarray, List[float]]*, *units: str = 'rad'*) → None

Rotate molecule.

Parameters

- **angle** (*float*) – Rotation angle
- **axis** (*np.ndarray*) – Axis of rotation (in 3D)
- **units** (*{"rad", "deg"}*) – Units of the angle (radians *rad* or degrees *deg*)

strip() → None

Strip hydrogen atoms.

to_graph()

Convert molecule to graph.

Returns

Molecular graph.

Return type

Graph

Notes

If the molecule does not have an associated adjacency matrix, a simple bond perception is used.

The molecular graph is cached per backend.

translate(*vector: Union[ndarray, List[float]]*) → None

Translate molecule.

Parameters

vector (*np.ndarray*) – Translation vector (in 3D)

spyrmsd.molecule.**coords_from_molecule**(*mol:* Molecule, *center: bool = False*) → ndarray

Atomic coordinates from molecule.

Parameters

- **mol** (*molecule.Molecule*) – Molecule
- **center** (*bool*) – Center flag

Returns

Atomic coordinates (possibly centred)

Return type

np.ndarray

Notes

Atomic coordinates are centred according to the center of geometry, not the center of mass.

spyrmsd.qcp module

spyrmsd.qcp.**K_mtx**(*M*)

Compute symmetric key matrix.

Parameters
M (*numpy.ndarray*) – Inner product between coordinate matrices

Returns
Symmetric key matrix

Return type
numpy.ndarray

Notes

The symmetric key matrix corresponds to the matrix $\mathbf{K}$.[2]

If S_{xy} is defined as

$$S_{xy} = \sum_{i}^{N} x_{B,i} y_{A,i}$$

then $\mathbf{K}$ is the 4×4 symmetric matrix given by

$$\begin{pmatrix} S_{xx} + S_{yy} + S_{zz} & S_{yz} - S_{zy} & S_{zx} - S_{xz} & S_{xy} - S_{yx} \\ & S_{xx} - S_{yy} - S_{zz} & S_{xy} + S_{yx} & S_{zx} + S_{xz} \\ & & -S_{xx} + S_{yy} - S_{zz} & S_{yz} - S_{zy} \\ & & & -S_{xx} - S_{yy} + S_{zz} \end{pmatrix}$$

spyrmsd.qcp.**M_mtx**(*A: ndarray*, *B: ndarray*) → ndarray

Compute inner product between coordinate matrices.

Parameters

- **A** (*numpy.ndarray*) – Coordinates *A*
- **B** (*numpy.ndarray*) – Coordinates *B*

Returns
Inner product of the coordinate matrices *A* and *B*

Return type
numpy.ndarray

[2] D. L. Theobald, *Rapid calculation of RMSDs using a quaternion-based characteristic polynomial*, Acta Crys. A **61**, 478-480 (2005).

Notes

The inner product of the coordinate matrices *A* and *B* corresponds to the matrix $\mathbf{M}$.[1]

If S_{xy} is defined as

$$S_{xy} = \sum_{i}^{N} x_{B,i} y_{A,i}$$

then $\mathbf{M}$ is the 3×3 matrix given by

$$\begin{pmatrix} S_{xx} & S_{xy} & S_{xz} \\ S_{yx} & S_{yy} & S_{yz} \\ S_{zx} & S_{zy} & S_{zz} \end{pmatrix}$$

spyrmsd.qcp.**coefficients**(*M: ndarray*, *K: ndarray*) → Tuple[float, float, float]

Compute quaternion polynomial coefficients.

Parameters

- **M** (*numpy.ndarray*) – Inner product between coordinate matrices
- **K** (*numpy.ndarray*) – Symmetric key matrix

Returns

Quaternion polynomial coefficients

Return type

Tuple[float, float, float]

Notes

Returns only $\mathbf{M}$- and $\mathbf{K}$-dependent coefficients are returned. $c_4 = 1$ and $c_3 = 0$ are not returned.

The $\mathbf{M}$- and $\mathbf{K}$-dependent quaternion polynomial coefficients are given by

$$c_2 = -2 \text{ tr } \left(\mathbf{M}^T \mathbf{M}\right)$$

$$c_1 = -8 \det(\mathbf{M})$$

$$c_0 = \det(\mathbf{K})$$

spyrmsd.qcp.**lambda_max**(*Ga: float*, *Gb: float*, *c2: float*, *c1: float*, *c0: float*) → float

Find largest root of the quaternion polynomial.

Parameters

- **Ga** (*float*) – Inner product of structure A
- **Gb** – Inner product of structure B
- **c2** – Coefficient c_2 of the quaternion polynomial
- **c1** – Coefficient c_1 of the quaternion polynomial
- **c0** – Coefficient c_0 of the quaternion polynomial

Returns

Largest root of the quaternion polynomial ($\lambda_{\max}$)

[1] D. L. Theobald, *Rapid calculation of RMSDs using a quaternion-based characteristic polynomial*, Acta Crys. A **61**, 478-480 (2005).

Return type
float

spyrmsd.qcp.**qcp_rmsd**(*A: ndarray*, *B: ndarray*, *atol: float = 1e-09*) → float

Compute RMSD using the quaternion polynomial method.

Parameters

- **A** (*numpy.ndarray*) – Coordinates of structure A
- **B** (*numpy.ndarray*) – Coordinates of structure B
- **atol** (*float*) – Absolute tolerance parameter (see notes)

Returns
RMSD between structures *A* and *B*

Return type
float

Raises
AssertionError – If the shape of structures *A* and *B* is different

Notes

If the structures *A* and *B* can be superimposed exactly (i.e. they differ only by center-of-mass translations and rotations), we have

$$G_a + G_b = 2\lambda_{\max}$$

This means that $s = G_a + G_b b - 2 * \lambda_{\max}$ can become negative because of numerical errors and therefore $\sqrt{s}$ fails. In order to avoid this problem, the final RMSD is set to 0 if $|s| < atol$.

spyrmsd.rmsd module

spyrmsd.rmsd.**hrmsd**(*coords1: ndarray*, *coords2: ndarray*, *atomicn1: ndarray*, *atomicn2: ndarray*, *center=False*)

Compute minimum RMSD using the Hungarian method.

Parameters

- **coords1** (*np.ndarray*) – Coordinate of molecule 1
- **coords2** (*np.ndarray*) – Coordinates of molecule 2
- **atomicn1** (*np.ndarray*) – Atomic numbers for molecule 1
- **atomicn2** (*np.ndarray*) – Atomic numbers for molecule 2

Returns
Minimum RMSD (after assignment)

Return type
float

Notes

The Hungarian algorithm is used to solve the linear assignment problem, which is a minimum weight matching of the molecular graphs (bipartite).[2]

The linear assignment problem is solved for every element separately.

spyrmsd.rmsd.**rmsd**(*coords1: ndarray*, *coords2: ndarray*, *atomicn1: ndarray*, *atomicn2: ndarray*, *center: bool = False*, *minimize: bool = False*, *atol: float = 1e-09*) → float

Compute RMSD

Parameters

- **coords1** (*np.ndarray*) – Coordinate of molecule 1
- **coords2** (*np.ndarray*) – Coordinates of molecule 2
- **atomicn1** (*np.ndarray*) – Atomic numbers for molecule 1
- **atomicn2** (*np.ndarray*) – Atomic numbers for molecule 2
- **center** (*bool*) – Center molecules at origin
- **minimize** (*bool*) – Compute minimum RMSD (with QCP method)
- **atol** (*float*) – Absolute tolerance parameter for QCP method (see `qcp_rmsd()`)

Returns
RMSD

Return type
float

Notes

When *minimize=True*, the QCP method is used.[1] The molecules are centred at the origin according to the center of geometry and superimposed in order to minimize the RMSD.

spyrmsd.rmsd.**rmsdwrapper**(*molref: Molecule*, *mols: Union[Molecule, List[Molecule]]*, *symmetry: bool = True*, *center: bool = False*, *minimize: bool = False*, *strip: bool = True*, *cache: bool = True*) → Any

Compute RMSD between two molecule.

Parameters

- **molref** (*molecule.Molecule*) – Reference molecule
- **mols** (*Union[molecule.Molecule, List[molecule.Molecule]]*) – Molecules to compare to reference molecule
- **symmetry** (*bool, optional*) – Symmetry-corrected RMSD (using graph isomorphism)
- **center** (*bool, optional*) – Center molecules at origin
- **minimize** (*bool, optional*) – Minimised RMSD (using the quaternion polynomial method)
- **strip** (*bool, optional*) – Strip hydrogen atoms

Returns
RMSDs

[2] W. J. Allen and R. C. Rizzo, *Implementation of the Hungarian Algorithm to Account for Ligand Symmetry and Similarity in Structure-Based Design*, J. Chem. Inf. Model. **54**, 518-529 (2014)

[1] D. L. Theobald, *Rapid calculation of RMSDs using a quaternion-based characteristic polynomial*, Acta Crys. A **61**, 478-480 (2005).

Return type
List[float]

spyrmsd.rmsd.**symmrmsd**(*coordsref: ndarray*, *coords: Union[ndarray, List[ndarray]]*, *apropsref: ndarray*, *aprops: ndarray*, *amref: ndarray*, *am: ndarray*, *center: bool = False*, *minimize: bool = False*, *cache: bool = True*, *atol: float = 1e-09*) → Any

Compute RMSD using graph isomorphism for multiple coordinates.

Parameters

- **coordsref** (*np.ndarray*) – Coordinate of reference molecule
- **coords** (*List[np.ndarray]*) – Coordinates of other molecule
- **apropsref** (*np.ndarray*) – Atomic properties for reference
- **aprops** (*np.ndarray*) – Atomic properties for other molecule
- **amref** (*np.ndarray*) – Adjacency matrix for reference molecule
- **am** (*np.ndarray*) – Adjacency matrix for other molecule
- **center** (*bool*) – Centering flag
- **minimize** (*bool*) – Minimum RMSD
- **cache** (*bool*) – Cache graph isomorphisms
- **atol** (*float*) – Absolute tolerance parameter for QCP (see `qcp_rmsd()`)

Returns
float – Symmetry-corrected RMSD(s) and graph isomorphisms

Return type
Union[float, List[float]]

Notes

Graph isomorphism is introduced for symmetry corrections. However, it is also useful when two molecules do not have the atoms in the same order since atom matching according to atomic numbers and the molecular connectivity is performed. If atoms are in the same order and there is no symmetry, use the *rmsd* function.

spyrmsd.utils module

spyrmsd.utils.**center**(*coordinates: ndarray*) → ndarray

Center coordinates.

Parameters
coordinates (*np.ndarray*) – Coordinates

Returns
Centred coordinates

Return type
np.ndarray

spyrmsd.utils.**center_of_geometry**(*coordinates: ndarray*) → ndarray

Center of geometry.

Parameters
coordinates (*np.ndarray*) – Coordinates

Returns
Center of geometry

Return type
np.ndarray

spyrmsd.utils.**deg_to_rad**(*angle: float*) → float

Convert angle in degrees to angle in radians.

Parameters
angle (*float*) – Angle (in degrees)

Returns
Angle (in radians)

Return type
float

spyrmsd.utils.**format**(*fname: str*) → str

Extract format extension from file name.

Parameters
fname (*str*) – File name

Returns
File extension

Return type
str

Notes

The file extension is returned without the . character, i.e. for the file *path/filename.ext* the string *ext* is returned.

If a file is compressed, the *.gz* extension is ignored.

spyrmsd.utils.**molformat**(*fname: str*) → str

Extract an OpenBabel-friendly format from file name.

Parameters
fname (*str*) – File name

Returns
File extension in an OpenBabel-friendly format

Return type
str

Notes

File types in OpenBabel do not always correspond to the file extension. This function converts the file extension to an OpenBabel file type.

The following table shows the different conversions performed by this function:

Extension	File Type
xyz	XYZ

spyrmsd.utils.**rotate**(*v: ndarray*, *angle: float*, *axis: ndarray*, *units: str = 'rad'*) → ndarray

Rotate vector.

Parameters

- **v** (*numpy.array*) – 3D vector to be rotated
- **angle** (*float*) – Angle of rotation (in *units*)
- **axis** (*numpy.array*) – 3D axis of rotation
- **units** (*{"rad", "deg"}*) – Units of *angle* (in radians *rad* or degrees *deg*)

Returns

Rotated vector

Return type

numpy.array

Raises

- **AssertionError** – If the axis of rotation is not a 3D vector
- **ValueError** – If *units* is not *rad* or *deg*

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

N

O

Q

R

S

T

U

V