ChemCoord Documentation

Release v1.3.0

Oskar Weser

Oct 03, 2017

Contents

CHAPTER 1

Features

- You can use it as a python module.
- It reliably converts from Cartesian space (xyz-files) to internal coordinates (zmat-files) **without** introducing dummy atoms. Even in the case of linearity.
- The created zmatrix is not only a transformation to internal coordinates, it is a "chemical" zmatrix. By chemical I mean, that e.g. distances are along bonds or dihedrals are defined as you draw them in chemical textbooks.
- It derived from my own work and I heavily use it during the day. So all functions are tested and tailored around the workflow in theoretical chemistry.
- The classes are safe to inherit from and you can easily costumize it for the needs of your project.

CHAPTER 2

Contents

Installation Guide

You need a working python (both python2 and 3) installation together with some standard modules. You can use for example the anaconda3 installer.

The advantage of the anaconda3 installer is that you get a lot of additional modules and programs, that make it really easy to work with python. For example Ipython and the jupyter notebooks. I highly recommend to use those.

Unix

Just type in your terminal:

```
pip install chemcoord
```

This should also resolve all dependencies automatically.

Windows

I tested neither installation nor running the module on windows. As far as I know it should work as well if you use the `pip` manager. If you get it installed and running, please report it on the Github page.

Introduction and General Structure

What you need to know

I assume that you know python.

You can use chemcoord without knowing Pandas, but it gives you a great advantage. If you invest 1h for their tutorial you will greatly increase your productivity in scientific data analysis.

It also helps to know about numpy.

Internal representation of Data

This module uses pandas DataFrames to represent cartesian and internal coordinates. (I will refer to them in lab slang as xyz and zmat)

The xyz_frame has at least four columns `['atom', 'x', 'y', 'z']`.

The zmat_frame has at least seven columns `['atom', 'b', 'bond', 'a', 'angle', 'd', 'dihedral']`.

Since they are normal pandas DataFrames you can do everything with them as long as you respect this structure. This means it is possible to append e.g. a column for the masses of each atom. Besides you can use all the capabilities of pandas.

If you want for example to get only the oxygen atoms of a xyz_frame you can use boolean slicing:

```
xyz_frame[xyz_frame['atom'] == 'O']
```

Main classes of this module

The "working horses" of this module are the `Cartesian` and the `Zmat` class. They have the methods to operate on their coordinates.

The methods of an instance of the `Cartesian` class usually return new instances of *Cartesian*. Besides all methods are **sideeffect free unless otherwise stated**.

Let's assume you have a `molecule1` and you want to cut a sphere around the origin which gives you `molecule2`:

```
molecule2 = molecule1.cut_sphere()
```

If you try this, you will see that:

- molecule2 is a `Cartesian` instance.
- molecule1 remains unchanged.

Tutorial

Just follow the link to the example notebooks.

- Cartesian
- Zmat
- Transformation
- Advanced customisation

If you want to have an interactive session, just download the following zip file, which contains all notebooks and coordinates.

Documentation

Contents:

Cartesian coordinates

Cartesian

The *Cartesian* class which is used to represent a molecule in cartesian coordinates.

Cartesian([frame, atoms, coords, index, ...])	The main class for dealing with cartesian Coordinates.

chemcoord.Cartesian

class chemcoord.**Cartesian**(*frame=None*, *atoms=None*, *coords=None*, *index=None*, *metadata=None*, *_metadata=None*)

The main class for dealing with cartesian Coordinates.

Mathematical Operations:

It supports binary operators in the logic of the scipy stack, but you need python3.x for using the matrix multiplication operator `@`.

The general rule is that mathematical operations using the binary operators `+ - * / @` and the unary operators `+ - abs` are only applied to the `['x', 'y', 'z']` columns.

Addition/Subtraction/Multiplication/Division: If you add a scalar to a Cartesian it is added elementwise onto the `['x', 'y', 'z']` columns. If you add a 3-dimensional vector, list, tuple... the first element of this vector is added elementwise to the `'x'` column of the Cartesian instance and so on. The last possibility is to add a matrix with `shape=(len(Cartesian), 3)` which is again added elementwise. The same rules are true for subtraction, division and multiplication.

Matrixmultiplication: Only leftsided multiplication with a matrix of `shape=(n, 3)`, where `n` is a natural number, is supported. The usual usecase is for example `np.diag([1, 1, -1]) @ cartesian_instance` to mirror on the x-y plane.

Indexing:

The indexing behaves like Indexing and Selecting data in Pandas. You can slice with *loc()*, *iloc()* and `Cartesian[...]`. The only question is about the return type. If the information in the columns is enough to draw a molecule, an instance of the own class (e.g. *Cartesian*) is returned. If the information in the columns is not enough to draw a molecule, there are two cases to consider:

- A `Series` instance is returned for one dimensional slices.
- A `DataFrame` instance is returned in all other cases.

This means that:

`molecule.loc[:, ['atom', 'x', 'y', 'z']]` returns a *Cartesian*.

`molecule.loc[:, ['atom', 'x']]` returns a `pandas.DataFrame`.

`molecule.loc[:, 'atom']` returns a `pandas.Series`.

Comparison:

Comparison for equality with == is supported. It behaves exactly like the equality comparison of DataFrames in pandas. Amongst other things this means that the index has to be the same and the comparison of floating point numbers is exact and not numerical. For this reason you rarely want to use ==. Usually the question is "are two given molecules chemically the same". For this comparison you have to use the function `allclose()`, which moves to the barycenter, aligns along the principal axes of inertia and compares numerically.

Chemical Methods

__init__([frame, atoms, coords, index, ...])	How to initialize a Cartesian instance.
get_bonds([self_bonding_allowed, offset, ...])	Return a dictionary representing the bonds.
restrict_bond_dict(bond_dict)	Restrict a bond dictionary to self.
get_fragment(list_of_indextuples[, ...])	Get the indices of the atoms in a fragment.
fragmentate([give_only_index, use_lookup])	Get the indices of non bonded parts in the molecule.
get_without(fragments[, use_lookup])	Return self without the specified fragments.
add_data([new_cols])	Adds a column with the requested data.
get_total_mass()	Returns the total mass in g/mol.
get_coordination_sphere(index_of_atom[, ...])	Return a Cartesian of atoms in the n-th coordination sphere.
partition_chem_env([n_sphere, use_lookup])	This function partitions the molecule into subsets of the same chemical environment.

chemcoord.Cartesian.__init__

Cartesian.**__init__**(*frame=None*, *atoms=None*, *coords=None*, *index=None*, *metadata=None*, *_metadata=None*)

How to initialize a Cartesian instance.

Parameters

- **frame** (*pd.DataFrame*) – A Dataframe with at least the columns `['atom', 'x', 'y', 'z']`. Where `'atom'` is a string for the elementsymbol.
- **atoms** (*sequence*) – A list of strings. (Elementsymbols)
- **coords** (*sequence*) – A `n_atoms * 3` array containg the positions of the atoms. Note that atoms and coords are mutually exclusive to frame. Besides atoms and coords have to be both either None or not None.

Returns A new cartesian instance.

Return type *Cartesian*

chemcoord.Cartesian.get_bonds

Cartesian.**get_bonds**(*self_bonding_allowed=False*, *offset=3*, *modified_properties=None*, *use_lookup=False*, *set_lookup=True*, *atomic_radius_data=None*)

Return a dictionary representing the bonds.

> **Warning:** This function is **not sideeffect free**, since it assigns the output to a variable `self._metadata['bond_dict']` if `set_lookup` is `True` (which is the default). This is necessary for performance reasons.

`.get_bonds()` will use or not use a lookup depending on `use_lookup`. Greatly increases performance if True, but could introduce bugs in certain situations.

Just imagine a situation where the `Cartesian` is changed manually. If you apply lateron a method e.g. `to_zmat()` that makes use of `get_bonds()` the dictionary of the bonds may not represent the actual situation anymore.

You have two possibilities to cope with this problem. Either you just re-execute `get_bonds` on your specific instance, or you change the `internally_use_lookup` option in the settings.

Please note that the internal use of the lookup variable greatly improves performance.

Parameters

- **modified_properties** (*dic*) – If you want to change the van der Vaals radius of one or more specific atoms, pass a dictionary that looks like:

```
modified_properties = {index1: 1.5}
```

  For global changes use the constants module.
- **offset** (*float*) –
- **use_lookup** (*bool*) –
- **set_lookup** (*bool*) –
- **self_bonding_allowed** (*bool*) –
- **atomic_radius_data** (*str*) – Defines which column of `constants.elements` is used. The default is `atomic_radius_cc` and can be changed with `settings['defaults']['atomic_radius_data']`. Compare with *add_data()*.

Returns Dictionary mapping from an atom index to the set of indices of atoms bonded to.

Return type dict

chemcoord.Cartesian.restrict_bond_dict

Cartesian.**restrict_bond_dict**(*bond_dict*)

Restrict a bond dictionary to self.

Parameters **bond_dict** (*dict*) – Look into *get_bonds()*, to see examples for a bond_dict.

Returns bond dictionary

chemcoord.Cartesian.get_fragment

Cartesian.**get_fragment**(*list_of_indextuples*, *give_only_index=False*, *use_lookup=None*)

Get the indices of the atoms in a fragment.

The list_of_indextuples contains all bondings from the molecule to the fragment. `[(1,3), (2,4)]` means for example that the fragment is connected over two bonds. The first bond is from atom 1 in the molecule to atom 3 in the fragment. The second bond is from atom 2 in the molecule to atom 4 in the fragment.

Parameters

- **list_of_indextuples** (*list*) –
- **give_only_index** (*bool*) – If `True` a set of indices is returned. Otherwise a new Cartesian instance.
- **use_lookup** (*bool*) – Use a lookup variable for *get_bonds()*. The default is specified in `settings['defaults']['use_lookup']`

Returns A set of indices or a new Cartesian instance.

chemcoord.Cartesian.fragmentate

Cartesian.**fragmentate**(*give_only_index=False*, *use_lookup=None*)

Get the indices of non bonded parts in the molecule.

Parameters

- **give_only_index** (*bool*) – If `True` a set of indices is returned. Otherwise a new Cartesian instance.
- **use_lookup** (*bool*) – Use a lookup variable for *get_bonds()*.
- **use_lookup** – Use a lookup variable for *get_bonds()*. The default is specified in `settings['defaults']['use_lookup']`

Returns A list of sets of indices or new Cartesian instances.
Return type list

chemcoord.Cartesian.get_without

`Cartesian.`**get_without**(*fragments*, *use_lookup=None*)
Return self without the specified fragments.

Parameters
- **fragments** – Either a list of *Cartesian* or a *Cartesian*.
- **use_lookup** (*bool*) – Use a lookup variable for *get_bonds()*. The default is specified in `settings['defaults']['use_lookup']`

Returns List containing *Cartesian*.
Return type list

chemcoord.Cartesian.add_data

`Cartesian.`**add_data**(*new_cols=None*)
Adds a column with the requested data.

If you want to see for example the mass, the colormap used in jmol and the block of the element, just use:

```
['mass', 'jmol_color', 'block']
```

The underlying `pd.DataFrame` can be accessed with `constants.elements`. To see all available keys use `constants.elements.info()`.

The data comes from the module mendeleev written by Lukasz Mentel.

Please note that I added three columns to the mendeleev data:

```
['atomic_radius_cc', 'atomic_radius_gv', 'gv_color',
    'valency']
```

The `atomic_radius_cc` is used by default by this module for determining bond lengths. The three others are taken from the MOLCAS grid viewer written by Valera Veryazov.

Parameters
- **new_cols** (*str*) – You can pass also just one value. E.g. `'mass'` is equivalent to `['mass']`. If `new_cols` is `None` all available data is returned.
- **inplace** (*bool*) –

Returns
Return type *Cartesian*

chemcoord.Cartesian.get_total_mass

`Cartesian.`**get_total_mass**()
Returns the total mass in g/mol.

Parameters **None** –

Returns
Return type float

chemcoord.Cartesian.get_coordination_sphere

Cartesian.**get_coordination_sphere**(*index_of_atom*, *n_sphere=1*, *give_only_index=False*, *only_surface=True*, *exclude=None*, *use_lookup=None*)

Return a Cartesian of atoms in the n-th coordination sphere.

Connected means that a path along covalent bonds exists.

Parameters

- **index_of_atom** (*int*) –
- **give_only_index** (*bool*) – If `True` a set of indices is returned. Otherwise a new Cartesian instance.
- **n_sphere** (*int*) – Determines the number of the coordination sphere.
- **only_surface** (*bool*) – Return only the surface of the coordination sphere.
- **exclude** (*set*) – A set of indices that should be ignored for the path finding.
- **use_lookup** (*bool*) – Use a lookup variable for *get_bonds()*. The default is specified in `settings['defaults']['use_lookup']`

Returns A set of indices or a new Cartesian instance.

chemcoord.Cartesian.partition_chem_env

Cartesian.**partition_chem_env**(*n_sphere=4*, *use_lookup=None*)

This function partitions the molecule into subsets of the same chemical environment.

A chemical environment is specified by the number of surrounding atoms of a certain kind around an atom with a certain atomic number represented by a tuple of a string and a frozenset of tuples. The `n_sphere` option determines how many branches the algorithm follows to determine the chemical environment.

Example: A carbon atom in ethane has bonds with three hydrogen (atomic number 1) and one carbon atom (atomic number 6). If `n_sphere=1` these are the only atoms we are interested in and the chemical environment is:

```
('C', frozenset([('H', 3), ('C', 1)]))
```

If `n_sphere=2` we follow every atom in the chemical enviromment of `n_sphere=1` to their direct neighbours. In the case of ethane this gives:

```
('C', frozenset([('H', 6), ('C', 1)]))
```

In the special case of ethane this is the whole molecule; in other cases you can apply this operation recursively and stop after `n_sphere` or after reaching the end of branches.

Parameters

- **n_sphere** (*int*) –
- **use_lookup** (*bool*) – Use a lookup variable for *get_bonds()*. The default is specified in `settings['defaults']['use_lookup']`

Returns

The output will look like this:

```
{ (element_symbol, frozenset([tuples])) : set([indices]) }
```

```
A dictionary mapping from a chemical environment to
the set of indices of atoms in this environment.
```

Return type dict

Manipulate

cut_cuboid([a, b, c, origin, ...])	Cut a cuboid specified by edge and radius.
cut_sphere([radius, origin, outside_sliced, ...])	Cut a sphere specified by origin and radius.
basistransform(new_basis[, old_basis, ...])	Transform the frame to a new basis.
align(other[, indices, ignore_hydrogens])	Align two Cartesians.
reindex_similar(other[, n_sphere])	Reindex `other` to be similarly indexed as `self`.
change_numbering(rename_dict[, inplace])	Return the reindexed version of Cartesian.
subs(variable, value)	Substitute a symbolic expression in `['x', 'y', 'z']`

chemcoord.Cartesian.cut_cuboid

Cartesian.**cut_cuboid**(*a=20*, *b=None*, *c=None*, *origin=None*, *outside_sliced=True*, *preserve_bonds=False*)

Cut a cuboid specified by edge and radius.

Parameters

- **a** (*float*) – Value of the a edge.
- **b** (*float*) – Value of the b edge. Takes value of a if None.
- **c** (*float*) – Value of the c edge. Takes value of a if None.
- **origin** (*list*) – Please note that you can also pass an integer. In this case it is interpreted as the index of the atom which is taken as origin.
- **outside_sliced** (*bool*) – Atoms outside/inside the sphere are cut away.
- **preserve_bonds** (*bool*) – Do not cut covalent bonds.

Returns

Return type *Cartesian*

chemcoord.Cartesian.cut_sphere

Cartesian.**cut_sphere**(*radius=15.0*, *origin=None*, *outside_sliced=True*, *preserve_bonds=False*)

Cut a sphere specified by origin and radius.

Parameters

- **radius** (*float*) –
- **origin** (*list*) – Please note that you can also pass an integer. In this case it is interpreted as the index of the atom which is taken as origin.
- **outside_sliced** (*bool*) – Atoms outside/inside the sphere are cut out.
- **preserve_bonds** (*bool*) – Do not cut covalent bonds.

Returns

Return type *Cartesian*

chemcoord.Cartesian.basistransform

Cartesian.**basistransform**(*new_basis*, *old_basis=None*, *orthonormalize=True*)

Transform the frame to a new basis.

This function transforms the cartesian coordinates from an old basis to a new one. Please note that old_basis and new_basis are supposed to have full Rank and consist of three linear independent vectors. If rotate_only is True, it is asserted, that both bases are orthonormal and right handed. Besides all involved matrices are transposed instead of inverted. In some applications this may require the function `xyz_functions.orthonormalize()` as a previous step.

Parameters
- **old_basis** (*np.array*) –
- **new_basis** (*np.array*) –
- **rotate_only** (*bool*) –

Returns The transformed molecule.

Return type *Cartesian*

chemcoord.Cartesian.align

`Cartesian.`**`align`**(*other*, *indices=None*, *ignore_hydrogens=False*)

Align two Cartesians.

Minimize the RMSD (root mean squared deviation) between `self` and `other`. Returns a tuple of copies of `self` and `other` where both are centered around their centroid and `other` is rotated unto `self`. The rotation minimises the distances between the atom pairs of same label. Uses the Kabsch algorithm implemented within `get_kabsch_rotation()`

Note: If `indices is None`, then `len(self) == len(other)` must be true and the elements in each index have to be the same.

Parameters
- **other** (`Cartesian`) –
- **indices** (*sequence*) – It is possible to specify a subset of indices that is used for the determination of the best rotation matrix:

```
[[i1, i2,...], [j1, j2,...]]
```

  If `indices` is given in this form, the rotation matrix minimises the distance between `i1` and `j1`, `i2` and `j2` and so on.
- **ignore_hydrogens** (*bool*) –

Returns

Return type tuple

chemcoord.Cartesian.reindex_similar

`Cartesian.`**`reindex_similar`**(*other*, *n_sphere=4*)

Reindex `other` to be similarly indexed as `self`.

Returns a reindexed copy of `other` that minimizes the distance for each atom to itself in the same chemical environemt from `self` to `other`. Read more about the definition of the chemical environment in *Cartesian.partition_chem_env()*

Note: It is necessary to align `self` and other before applying this method. This can be done via *align()*.

Note: It is probably necessary to improve the result using *change_numbering()*.

Parameters
- **other** (*Cartesian*) –
- **n_sphere** (*int*) – Wrapper around the argument for *partition_chem_env()*.

Returns Reindexed version of other
Return type *Cartesian*

chemcoord.Cartesian.change_numbering

Cartesian.**change_numbering**(*rename_dict*, *inplace=False*)

Return the reindexed version of Cartesian.

Parameters **rename_dict** (*dict*) – A dictionary mapping integers on integers.
Returns A renamed copy according to the dictionary passed.
Return type *Cartesian*

chemcoord.Cartesian.subs

Cartesian.**subs**(*variable*, *value*)

Substitute a symbolic expression in `['x', 'y', 'z']`

This is a wrapper around the substitution mechanism of sympy. Any symbolic expression in the columns `['x', 'y', 'z']` of `self` will be substituted with value.

Parameters
- **symb_expr** (*sympy expression*) –
- **value** –
- **perform_checks** (*bool*) – If `perform_checks is True`, it is asserted, that the resulting Zmatrix can be converted to cartesian coordinates. Dummy atoms will be inserted automatically if necessary.

Returns Cartesian with substituted symbolic expressions. If all resulting sympy expressions in a column are numbers, the column is recasted to 64bit float.
Return type *Cartesian*

Geometry

get_bond_lengths(indices)	Return the distances between given atoms.
get_angle_degrees(indices)	Return the angles between given atoms.
get_dihedral_degrees(indices[, start_row])	Return the dihedrals between given atoms.
get_barycenter()	Return the mass weighted average location.
get_inertia()	Calculate the inertia tensor and transforms along rotation axes.
get_centroid()	Return the average location.
get_distance_to([origin, other_atoms, sort])	Return a Cartesian with a column for the distance from origin.
get_shortest_distance(other)	Calculate the shortest distance between self and other

chemcoord.Cartesian.get_bond_lengths

Cartesian.**get_bond_lengths**(*indices*)

Return the distances between given atoms.

Calculates the distance between the atoms with indices `i` and `b`. The indices can be given in three ways:

- As simple list `[i, b]`
- As list of lists: `[[i1, b1], [i2, b2]...]`
- As `pd.DataFrame` where `i` is taken from the index and `b` from the respective column `'b'`.

Parameters **indices** (*list*) –
Returns Vector of angles in degrees.
Return type `numpy.ndarray`

chemcoord.Cartesian.get_angle_degrees

Cartesian.**get_angle_degrees**(*indices*)

Return the angles between given atoms.

Calculates the angle in degrees between the atoms with indices `i, b, a`. The indices can be given in three ways:

- As simple list `[i, b, a]`
- As list of lists: `[[i1, b1, a1], [i2, b2, a2]...]`
- As `pd.DataFrame` where `i` is taken from the index and `b` and `a` from the respective columns `'b'` and `'a'`.

Parameters **indices** (*list*) –
Returns Vector of angles in degrees.
Return type `numpy.ndarray`

chemcoord.Cartesian.get_dihedral_degrees

Cartesian.**get_dihedral_degrees**(*indices*, *start_row=0*)

Return the dihedrals between given atoms.

Calculates the dihedral angle in degrees between the atoms with indices `i, b, a, d`. The indices can be given in three ways:

- As simple list `[i, b, a, d]`
- As list of lists: `[[i1, b1, a1, d1], [i2, b2, a2, d2]...]`
- As `pandas.DataFrame` where `i` is taken from the index and `b`, `a` and `d``from the respective columns ``'b'`, `'a'` and `'d'`.

Parameters **indices** (*list*) –
Returns Vector of angles in degrees.
Return type `numpy.ndarray`

chemcoord.Cartesian.get_barycenter

Cartesian.**get_barycenter**()

Return the mass weighted average location.

Parameters **None** –
Returns

Return type numpy.ndarray

chemcoord.Cartesian.get_inertia

Cartesian.**get_inertia**()
Calculate the inertia tensor and transforms along rotation axes.

This function calculates the inertia tensor and returns a 4-tuple.

The unit is `amu * length-unit-of-xyz-file**2`
Parameters **None** –
Returns

The returned dictionary has four possible keys:

`transformed_Cartesian`: A *Cartesian* that is transformed to the basis spanned by the eigenvectors of the inertia tensor. The x-axis is the axis with the lowest inertia moment, the z-axis the one with the highest. Contains also a column for the mass

`diag_inertia_tensor`: A vector containing the ascendingly sorted inertia moments after diagonalization.

`inertia_tensor`: The inertia tensor in the old basis.

`eigenvectors`: The eigenvectors of the inertia tensor in the old basis. Since the inertia_tensor is hermitian, they are orthogonal and are returned as an orthonormal righthanded basis. The i-th eigenvector corresponds to the i-th eigenvalue in `diag_inertia_tensor`.
Return type dict

chemcoord.Cartesian.get_centroid

Cartesian.**get_centroid**()
Return the average location.
Parameters **None** –
Returns
Return type numpy.ndarray

chemcoord.Cartesian.get_distance_to

Cartesian.**get_distance_to**(*origin=None*, *other_atoms=None*, *sort=False*)
Return a Cartesian with a column for the distance from origin.

chemcoord.Cartesian.get_shortest_distance

Cartesian.**get_shortest_distance**(*other*)
Calculate the shortest distance between self and other
Parameters **Cartesian** – other
Returns

Returns a tuple `i, j, d` with the following meaning:

`i`: The index on self that minimises the pairwise distance.

`j`: The index on other that minimises the pairwise distance.

`d`: The distance between self and other. (float)

Return type tuple

Conversion to internal coordinates

get_zmat([construction_table, use_lookup])	Transform to internal coordinates.
get_construction_table([fragment_list, ...])	Create a construction table for a Zmatrix.
check_dihedral(construction_table)	Checks, if the dihedral defining atom is colinear.
correct_dihedral(construction_table[, ...])	Reindexe the dihedral defining atom if linear reference is used.
check_absolute_refs(construction_table)	Checks first three rows of `construction_table` for linear references
correct_absolute_refs(construction_table)	Reindexe construction_table if linear reference in first three rows present.
to_zmat(*args, **kwargs)	Deprecated, use *get_zmat()*

chemcoord.Cartesian.get_zmat

`Cartesian.`**`get_zmat`**(*construction_table=None*, *use_lookup=None*)

Transform to internal coordinates.

Transforming to internal coordinates involves basically three steps:

1. Define an order of how to build and define for each atom the used reference atoms.

2. Check for problematic local linearity. In this algorithm an angle with `170 < angle < 10` is assumed to be linear. This is not the mathematical definition, but makes it safer against "floating point noise"

3. Calculate the bond lengths, angles and dihedrals using the references defined in step 1 and 2.

In the first two steps a so called `construction_table` is created. This is basically a Zmatrix without the values for the bonds, angles and dihedrals hence containing only the information about the used references. ChemCoord uses a `pandas.DataFrame` with the columns `['b', 'a', 'd']`. Look into *get_construction_table()* for more information.

It is important to know, that calculating the construction table is a very costly step since the algoritym tries to make some guesses based on connectivity to create a "chemical" zmatrix.

If you create several zmatrices based on the same references you can obtain the construction table of a zmatrix with `Zmat_instance.loc[:, ['b', 'a', 'd']]` If you then pass the buildlist as argument to `give_zmat`, the algorithm directly starts with step 3 (which is much faster).

If a `construction_table` is passed into *get_zmat()* the check for pathological linearity is not performed! So if a `construction_table` is either manually created, or obtained from *get_construction_table()* under the option `perform_checks = False`, it is recommended to use the following methods:

- *correct_dihedral()*
- *correct_absolute_refs()*

If you want to check for problematic indices in order to solve the invalid references yourself, use the following methods:

- *check_dihedral()*
- *check_absolute_refs()*

Parameters
- **construction_table** (*pandas.DataFrame*) –
- **use_lookup** (*bool*) – Use a lookup variable for *get_bonds()*. The default is specified in `settings['defaults']['use_lookup']`

Returns A new instance of *Zmat*.
Return type *Zmat*

chemcoord.Cartesian.get_construction_table

`Cartesian.`**`get_construction_table`**(*fragment_list=None*, *use_lookup=None*, *perform_checks=True*)

Create a construction table for a Zmatrix.

A construction table is basically a Zmatrix without the values for the bond lengths, angles and dihedrals. It contains the whole information about which reference atoms are used by each atom in the Zmatrix.

This method creates a so called "chemical" construction table, which makes use of the connectivity table in this molecule.

Parameters
- **fragment_list** (*sequence*) – There are four possibilities to specify the sequence of fragments:

  1. A list of tuples is given. Each tuple contains the fragment with its corresponding construction table in the form of:

  ```
  [(frag1, c_table1), (frag2, c_table2)...]
  ```

  If the construction table of a fragment is not complete, the rest of each fragment's construction table is calculated automatically.

  2. It is possible to omit the construction tables for some or all fragments as in the following example:

  ```
  [(frag1, c_table1), frag2, (frag3, c_table3)...]
  ```

  3. If `self` contains more atoms than the union over all fragments, the rest of the molecule without the fragments is automatically prepended using *get_without()*:

  ```
  self.get_without(fragments) + fragment_list
  ```

  4. If fragment_list is `None` then fragmentation, etc. is done automatically. The fragments are then sorted by their number of atoms, in order to use the largest fragment as reference for the other ones.
- **use_lookup** (*bool*) – Use a lookup variable for *get_bonds()*. The default is specified in `settings['defaults']['use_lookup']`
- **perform_checks** (*bool*) – The checks for invalid references are performed using *correct_dihedral()* and *correct_absolute_refs()*.

Returns Construction table
Return type `pandas.DataFrame`

chemcoord.Cartesian.check_dihedral

Cartesian.**check_dihedral**(*construction_table*)
Checks, if the dihedral defining atom is colinear.

Checks for each index starting from the third row of the `construction_table`, if the reference atoms are colinear.

Parameters **construction_table** (*pd.DataFrame*) –
Returns A list of problematic indices.
Return type list

chemcoord.Cartesian.correct_dihedral

Cartesian.**correct_dihedral**(*construction_table*, *use_lookup=None*)
Reindexe the dihedral defining atom if linear reference is used.

Uses *check_dihedral()* to obtain the problematic indices.

Parameters
- **construction_table** (*pd.DataFrame*) –
- **use_lookup** (*bool*) – Use a lookup variable for *get_bonds()*. The default is specified in `settings['defaults']['use_lookup']`

Returns Appropiately renamed construction table.
Return type pd.DataFrame

chemcoord.Cartesian.check_absolute_refs

Cartesian.**check_absolute_refs**(*construction_table*)
Checks first three rows of `construction_table` for linear references

Checks for each index from first to third row of the `construction_table`, if the references are colinear. This case has to be specially treated, because the references are not only atoms (to fix internal degrees of freedom) but also points in cartesian space called absolute references. (to fix translational and rotational degrees of freedom)

Parameters **construction_table** (*pd.DataFrame*) –
Returns A list of problematic indices.
Return type list

chemcoord.Cartesian.correct_absolute_refs

Cartesian.**correct_absolute_refs**(*construction_table*)
Reindexe construction_table if linear reference in first three rows present.

Uses *check_absolute_refs()* to obtain the problematic indices.

Parameters **construction_table** (*pd.DataFrame*) –
Returns Appropiately renamed construction table.
Return type pd.DataFrame

chemcoord.Cartesian.to_zmat

Cartesian.**to_zmat**(**args*, ***kwargs*)
Deprecated, use *get_zmat()*

Symmetry

get_pointgroup([tolerance])	Returns a PointGroup object for the molecule.
get_equivalent_atoms([tolerance])	Returns sets of equivalent atoms with symmetry operations
symmetrize([max_n, tolerance, epsilon])	Returns a symmetrized molecule
get_asymmetric_unit([eq])	

chemcoord.Cartesian.get_pointgroup

Cartesian.**get_pointgroup**(*tolerance=0.3*)

Returns a PointGroup object for the molecule.

Parameters **tolerance** (*float*) – Tolerance to generate the full set of symmetry operations.

Returns *PointGroupOperations*

chemcoord.Cartesian.get_equivalent_atoms

Cartesian.**get_equivalent_atoms**(*tolerance=0.3*)

Returns sets of equivalent atoms with symmetry operations

Parameters **tolerance** (*float*) – Tolerance to generate the full set of symmetry operations.

Returns

The returned dictionary has two possible keys:

`eq_sets`: A dictionary of indices mapping to sets of indices, each key maps to indices of all equivalent atoms. The keys are guaranteed to be not equivalent.

`sym_ops`: Twofold nested dictionary. `operations[i][j]` gives the symmetry operation that maps atom `i` unto `j`.

Return type dict

chemcoord.Cartesian.symmetrize

Cartesian.**symmetrize**(*max_n=10*, *tolerance=0.3*, *epsilon=0.001*)

Returns a symmetrized molecule

The equivalent atoms obtained via *get_equivalent_atoms()* are rotated, mirrored... unto one position. Then the average position is calculated. The average position is rotated, mirrored... back with the inverse of the previous symmetry operations, which gives the symmetrized molecule. This operation is repeated iteratively `max_n` times at maximum until the difference between subsequently symmetrized structures is smaller than `epsilon`.

Parameters

- **max_n** (*int*) – Maximum number of iterations.
- **tolerance** (*float*) – Tolerance for detecting symmetry. Gets passed as Argument into `PointGroupAnalyzer`.
- **epsilon** (*float*) – If the elementwise absolute difference of two subsequently symmetrized structures is smaller epsilon, the iteration stops before `max_n` is reached.

Returns

The returned dictionary has three possible keys:

`sym_mol`: A symmetrized molecule *Cartesian*

`eq_sets`: A dictionary of indices mapping to sets of indices, each key maps to indices of all equivalent atoms. The keys are guaranteed to be not symmetry-equivalent.

`sym_ops`: Twofold nested dictionary. `operations[i][j]` gives the symmetry operation that maps atom `i` unto `j`.

Return type dict

chemcoord.Cartesian.get_asymmetric_unit

Cartesian.**get_asymmetric_unit**(*eq=None*)

IO

write_xyz(*args, **kwargs)	Deprecated, use *to_xyz()*
to_xyz([buf, sort_index, index, header, ...])	Write xyz-file
read_xyz(inputfile[, start_index, ...])	Read a file of coordinate information.
view([viewer, use_curr_dir])	View your molecule.
to_string([buf, columns, col_space, header, ...])	Render a DataFrame to a console-friendly tabular output.
to_latex([buf, columns, col_space, header, ...])	Render a DataFrame to a tabular environment table.
get_pymatgen_molecule()	Create a Molecule instance of the pymatgen library
from_pymatgen_molecule(molecule)	Create an instance of the own class from a pymatgen molecule
get_ase_atoms()	Create an Atoms instance of the ase library
from_ase_atoms(atoms)	Create an instance of the own class from an ase molecule

chemcoord.Cartesian.write_xyz

Cartesian.**write_xyz**(**args*, ***kwargs*)

Deprecated, use *to_xyz()*

chemcoord.Cartesian.to_xyz

Cartesian.**to_xyz**(*buf=None*, *sort_index=True*, *index=False*, *header=False*, *float_format=<built-in method format of str object>*, *overwrite=True*)

Write xyz-file

Parameters

- **buf** (*str*) – StringIO-like, optional buffer to write to
- **sort_index** (*bool*) – If sort_index is true, the *Cartesian* is sorted by the index before writing.

- **float_format** (*one-parameter function*) – Formatter function to apply to column's elements if they are floats. The result of this function must be a unicode string.
- **overwrite** (*bool*) – May overwrite existing files.

Returns string (or unicode, depending on data and options)
Return type formatted

chemcoord.Cartesian.read_xyz

Cartesian.**read_xyz**(*inputfile*, *start_index=0*, *get_bonds=True*, *nrows=None*, *engine=None*)

Read a file of coordinate information.

Reads xyz-files.

Parameters
- **inputfile** (*str*) –
- **start_index** (*int*) –
- **get_bonds** (*bool*) –
- **nrows** (*int*) – Number of rows of file to read. Note that the first two rows are implicitly excluded.
- **engine** (*str*) – Wrapper for argument of `pandas.read_csv()`.

Returns
Return type *Cartesian*

chemcoord.Cartesian.view

Cartesian.**view**(*viewer=None*, *use_curr_dir=False*)

View your molecule.

Note: This function writes a temporary file and opens it with an external viewer. If you modify your molecule afterwards you have to recall view in order to see the changes.

Parameters
- **viewer** (*str*) – The external viewer to use. If it is None, the default as specified in cc.settings['defaults']['viewer'] is used.
- **use_curr_dir** (*bool*) – If True, the temporary file is written to the current diretory. Otherwise it gets written to the OS dependendent temporary directory.

Returns
Return type None

chemcoord.Cartesian.to_string

Cartesian.**to_string**(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *justify=None*, *line_width=None*, *max_rows=None*, *max_cols=None*, *show_dimensions=False*)

Render a DataFrame to a console-friendly tabular output.

Wrapper around the `pandas.DataFrame.to_string()` method.

chemcoord.Cartesian.to_latex

Cartesian.**to_latex**(*buf=None*, *columns=None*, *col_space=None*, *header=True*, *index=True*, *na_rep='NaN'*, *formatters=None*, *float_format=None*, *sparsify=None*, *index_names=True*, *bold_rows=True*, *column_format=None*, *longtable=None*, *escape=None*, *encoding=None*, *decimal='.'*, *multicolumn=None*, *multicolumn_format=None*, *multirow=None*)

Render a DataFrame to a tabular environment table.

You can splice this into a LaTeX document. Requires `\usepackage{booktabs}`. Wrapper around the `pandas.DataFrame.to_latex()` method.

chemcoord.Cartesian.get_pymatgen_molecule

Cartesian.**get_pymatgen_molecule**()

Create a Molecule instance of the pymatgen library

> **Warning:** The pymatgen library is imported locally in this function and will raise an `ImportError` exception, if it is not installed.

Parameters **None** –
Returns
Return type `pymatgen.core.structure.Molecule`

chemcoord.Cartesian.from_pymatgen_molecule

Cartesian.**from_pymatgen_molecule**(*molecule*)

Create an instance of the own class from a pymatgen molecule

Parameters **molecule** (`pymatgen.core.structure.Molecule`) –
Returns
Return type *Cartesian*

chemcoord.Cartesian.get_ase_atoms

Cartesian.**get_ase_atoms**()

Create an Atoms instance of the ase library

> **Warning:** The ase library is imported locally in this function and will raise an `ImportError` exception, if it is not installed.

Parameters **None** –
Returns
Return type `ase.atoms.Atoms`

chemcoord.Cartesian.from_ase_atoms

Cartesian.**from_ase_atoms**(*atoms*)

Create an instance of the own class from an ase molecule

Parameters **molecule** (ase.atoms.Atoms) –
Returns
Return type *Cartesian*

Pandas DataFrame Wrapper

copy()	
index	Returns the index.
columns	Returns the columns.
replace([to_replace, value, inplace, limit, ...])	Replace values given in 'to_replace' with 'value'.
sort_index([axis, level, ascending, ...])	Sort object by labels (along an axis)
set_index(keys[, drop, append, inplace, ...])	Set the DataFrame index (row labels) using one or more existing columns.
append(other[, ignore_index])	Append rows of *other* to the end of this frame, returning a new object.
insert(loc, column, value[, ...])	Insert column into molecule at specified location.
sort_values(by[, axis, ascending, inplace, ...])	Sort by the values along either axis
loc	Label based indexing
iloc	Label based indexing

chemcoord.Cartesian.copy

Cartesian.**copy**()

chemcoord.Cartesian.index

Cartesian.**index**
Returns the index.

Assigning a value to it changes the index.

chemcoord.Cartesian.columns

Cartesian.**columns**
Returns the columns.

Assigning a value to it changes the columns.

chemcoord.Cartesian.replace

Cartesian.**replace**(*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*, *axis=None*)
Replace values given in 'to_replace' with 'value'.

Wrapper around the pandas.DataFrame.replace() method.

chemcoord.Cartesian.sort_index

Cartesian.**sort_index**(*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*, *by=None*)

Sort object by labels (along an axis)

Wrapper around the `pandas.DataFrame.sort_index()` method.

chemcoord.Cartesian.set_index

Cartesian.**set_index**(*keys*, *drop=True*, *append=False*, *inplace=False*, *verify_integrity=False*)

Set the DataFrame index (row labels) using one or more existing columns.

Wrapper around the `pandas.DataFrame.set_index()` method.

chemcoord.Cartesian.append

Cartesian.**append**(*other*, *ignore_index=False*)

Append rows of *other* to the end of this frame, returning a new object.

Wrapper around the `pandas.DataFrame.append()` method.

Parameters

- **other** (`Cartesian`) –
- **ignore_index** (*sequence, bool, int*) – If it is a boolean, it behaves like in the description of `pandas.DataFrame.append()`. If it is a sequence, it becomes the new index. If it is an integer, `range(ignore_index, ignore_index + len(new))` becomes the new index.

Returns

Return type *Cartesian*

chemcoord.Cartesian.insert

Cartesian.**insert**(*loc*, *column*, *value*, *allow_duplicates=False*, *inplace=False*)

Insert column into molecule at specified location.

Wrapper around the `pandas.DataFrame.insert()` method.

chemcoord.Cartesian.sort_values

Cartesian.**sort_values**(*by*, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*)

Sort by the values along either axis

Wrapper around the `pandas.DataFrame.sort_values()` method.

chemcoord.Cartesian.loc

Cartesian.**loc**

Label based indexing

The indexing behaves like Indexing and Selecting data in Pandas. You can slice with *loc()*, *iloc()* and `Cartesian[...]`. The only question is about the return type. If the information in the columns is enough to draw a molecule, an instance of the own class (e.g. *Cartesian*) is returned. If the information in the columns is not enough to draw a molecule, there are two cases to consider:

- A `Series` instance is returned for one dimensional slices.
- A `DataFrame` instance is returned in all other cases.

This means that:

`molecule.loc[:, ['atom', 'x', 'y', 'z']]` returns a *Cartesian*.

`molecule.loc[:, ['atom', 'x']]` returns a `pandas.DataFrame`.

`molecule.loc[:, 'atom']` returns a `pandas.Series`.

chemcoord.Cartesian.iloc

`Cartesian.`**`iloc`**

Label based indexing

The indexing behaves like Indexing and Selecting data in Pandas. You can slice with *loc()*, *iloc()* and `Cartesian[...]`. The only question is about the return type. If the information in the columns is enough to draw a molecule, an instance of the own class (e.g. *Cartesian*) is returned. If the information in the columns is not enough to draw a molecule, there are two cases to consider:

- A `Series` instance is returned for one dimensional slices.
- A `DataFrame` instance is returned in all other cases.

This means that:

`molecule.loc[:, ['atom', 'x', 'y', 'z']]` returns a *Cartesian*.

`molecule.loc[:, ['atom', 'x']]` returns a `pandas.DataFrame`.

`molecule.loc[:, 'atom']` returns a `pandas.Series`.

Advanced methods

_divide_et_impera([n_atoms_per_set, offset])	
_preserve_bonds(sliced_cartesian[, use_lookup])	Is called after cutting geometric shapes.

chemcoord.Cartesian._divide_et_impera

`Cartesian.`**`_divide_et_impera`**(*n_atoms_per_set=500*, *offset=3*)

chemcoord.Cartesian._preserve_bonds

`Cartesian.`**`_preserve_bonds`**(*sliced_cartesian*, *use_lookup=None*)

Is called after cutting geometric shapes.

If you want to change the rules how bonds are preserved, when applying e.g. *Cartesian.cut_sphere()* this is the function you have to modify.

It is recommended to inherit from the Cartesian class to tailor it for your project, instead of modifying the source code of ChemCoord.

Parameters
- **sliced_frame** (Cartesian) –
- **use_lookup** (*bool*) – Use a lookup variable for *get_bonds()*. The default is specified in `settings['defaults']['use_lookup']`

Returns
Return type *Cartesian*

Attributes

columns	Returns the columns.
index	Returns the index.

xyz_functions

A collection of functions operating on instances of *Cartesian*.

isclose(a, b[, align, rtol, atol])	Compare two molecules for numerical equality.
allclose(a, b[, align, rtol, atol])	Compare two molecules for numerical equality.
concat(cartesians[, ignore_index, keys])	Join list of cartesians into one molecule.
write_molden(*args, **kwargs)	Deprecated, use *to_molden()*
to_molden(cartesian_list[, buf, sort_index, ...])	Write a list of Cartesians into a molden file.
read_molden(inputfile[, start_index, get_bonds])	Read a molden file.
view(molecule[, viewer, use_curr_dir])	View your molecule or list of molecules.
dot(A, B)	Matrix multiplication between A and B

chemcoord.xyz_functions.isclose

chemcoord.xyz_functions.**isclose**(*a*, *b*, *align=False*, *rtol=1e-05*, *atol=1e-08*)
Compare two molecules for numerical equality.

Parameters
- **a** (Cartesian) –
- **b** (Cartesian) –
- **align** (*bool*) – a and b are prealigned along their principal axes of inertia and moved to their barycenters before comparing.
- **rtol** (*float*) – Relative tolerance for the numerical equality comparison look into numpy.isclose() for further explanation.
- **atol** (*float*) – Relative tolerance for the numerical equality comparison look into numpy.isclose() for further explanation.

Returns Boolean array.

Return type numpy.ndarray

chemcoord.xyz_functions.allclose

chemcoord.xyz_functions.**allclose**(*a*, *b*, *align=False*, *rtol=1e-05*, *atol=1e-08*)
Compare two molecules for numerical equality.

Parameters

- **a** (Cartesian) –
- **b** (Cartesian) –
- **align** (*bool*) – a and b are prealigned along their principal axes of inertia and moved to their barycenters before comparing.
- **rtol** (*float*) – Relative tolerance for the numerical equality comparison look into numpy.allclose() for further explanation.
- **atol** (*float*) – Relative tolerance for the numerical equality comparison look into numpy.allclose() for further explanation.

Returns

Return type bool

chemcoord.xyz_functions.concat

chemcoord.xyz_functions.**concat**(*cartesians*, *ignore_index=False*, *keys=None*)

Join list of cartesians into one molecule.

Wrapper around the pandas.concat() function. Default values are the same as in the pandas function except for verify_integrity which is set to true in case of this library.

Parameters

- **ignore_index** (*sequence, bool, int*) – If it is a boolean, it behaves like in the description of pandas.DataFrame.append(). If it is a sequence, it becomes the new index. If it is an integer, range(ignore_index, ignore_index + len(new)) becomes the new index.
- **keys** (*sequence*) – If multiple levels passed, should contain tuples. Construct hierarchical index using the passed keys as the outermost level

Returns

Return type *Cartesian*

chemcoord.xyz_functions.write_molden

chemcoord.xyz_functions.**write_molden**(**args*, ***kwargs*)

Deprecated, use *to_molden()*

chemcoord.xyz_functions.to_molden

chemcoord.xyz_functions.**to_molden**(*cartesian_list*, *buf=None*, *sort_index=True*, *overwrite=True*, *float_format=<built-in method format of str object>*)

Write a list of Cartesians into a molden file.

Note: Since it permamently writes a file, this function is strictly speaking **not sideeffect free**. The list to be written is of course not changed.

Parameters

- **cartesian_list** (*list*) –
- **buf** (*str*) – StringIO-like, optional buffer to write to
- **sort_index** (*bool*) – If sort_index is true, the Cartesian is sorted by the index before writing.
- **overwrite** (*bool*) – May overwrite existing files.
- **float_format** (*one-parameter function*) – Formatter function to apply to column's elements if they are floats. The result of this function must be a unicode string.

Returns string (or unicode, depending on data and options)

Return type formatted

chemcoord.xyz_functions.read_molden

chemcoord.xyz_functions.**read_molden**(*inputfile*, *start_index=0*, *get_bonds=True*)

Read a molden file.

Parameters

- **inputfile** (*str*) –
- **start_index** (*int*) –

Returns A list containing *Cartesian* is returned.

Return type list

chemcoord.xyz_functions.view

chemcoord.xyz_functions.**view**(*molecule*, *viewer='gv.exe'*, *use_curr_dir=False*)

View your molecule or list of molecules.

Note: This function writes a temporary file and opens it with an external viewer. If you modify your molecule afterwards you have to recall view in order to see the changes.

Parameters

- **molecule** – Can be a cartesian, or a list of cartesians.
- **viewer** (*str*) – The external viewer to use. The default is specified in settings.viewer
- **use_curr_dir** (*bool*) – If True, the temporary file is written to the current diretory. Otherwise it gets written to the OS dependendent temporary directory.

Returns

Return type None

chemcoord.xyz_functions.dot

chemcoord.xyz_functions.**dot**(*A*, *B*)
Matrix multiplication between A and B

This function is equivalent to `A @ B`, which is unfortunately not possible under python 2.x.

Parameters

- **A** (*sequence*) –
- **B** (*sequence*) –

Returns

Return type sequence

Symmetry

PointGroupOperations(sch_symbol, operations)	Defines a point group as sequence of symmetry operations.

chemcoord.PointGroupOperations

class chemcoord.**PointGroupOperations**(*sch_symbol*, *operations*, *tolerance=0.1*)
Defines a point group as sequence of symmetry operations.

Parameters

- **sch_symbol** (*str*) – Schoenflies symbol of the point group.
- **operations** (numpy.ndarray) – Initial set of symmetry operations. It is sufficient to provide only just enough operations to generate the full set of symmetries.
- **tolerance** (*float*) – Tolerance to generate the full set of symmetry operations.

AsymmetricUnitCartesian([frame, atoms, ...])	Manipulate cartesian coordinates while preserving the point group.

chemcoord.AsymmetricUnitCartesian

class chemcoord.**AsymmetricUnitCartesian**(*frame=None*, *atoms=None*, *coords=None*, *index=None*, *metadata=None*, *_metadata=None*)
Manipulate cartesian coordinates while preserving the point group.

This class has all the methods of a *Cartesian*, with one additional *get_cartesian()* method and contains only one member of each symmetry equivalence class.

get_cartesian()	Return a *Cartesian* where all members of a symmetry equivalence class are inserted back in.

chemcoord.AsymmetricUnitCartesian.get_cartesian

AsymmetricUnitCartesian.**get_cartesian**()
Return a *Cartesian* where all members of a symmetry equivalence class are inserted back in.

Parameters `None` –

Returns A new cartesian instance.

Return type *Cartesian*

Internal coordinates

Zmat

The *Zmat* class which is used to represent a molecule in non redundant, internal coordinates.

Zmat(frame[, metadata, _metadata])	The main class for dealing with internal Coordinates.

chemcoord.Zmat

class chemcoord.**Zmat**(*frame*, *metadata=None*, *_metadata=None*)

The main class for dealing with internal Coordinates.

Rotational direction:

Chemcoord uses the IUPAC definition. Note that this does not include the automatic choosing of the canonical equivalence class representation. An angle of -30° could be represented by 270°. Use *iupacify()* to choose also the IUPAC conform angle representation.

Mathematical Operations:

The general rule is that mathematical operations using the binary operators `+ - * /` and the unary operators `+ - abs` are only applied to the `['bond', 'angle', 'dihedral']` columns.

Addition/Subtraction/Multiplication/Division: The most common case is to add another Zmat instance. In this case it is tested, if the used references are the same. Afterwards the addition in the `['bond', 'angle', 'dihedral']` columns is performed. If you add a scalar to a Zmat it is added elementwise onto the `['bond', 'angle', 'dihedral']` columns. If you add a 3-dimensional vector, list, tuple... the first element of this vector is added elementwise to the `'bond'` column of the Zmat instance and so on. The third possibility is to add a matrix with `shape=(len(Zmat), 3)` which is again added elementwise. The same rules are true for subtraction, division and multiplication.

Indexing:

The indexing behaves like Indexing and Selecting data in Pandas. You can slice with *loc()*, *iloc()*, and `Zmat[...]`. The only question is about the return type. If the information in the columns is enough to draw a molecule, an instance of the own class (e.g. *Zmat*) is returned. If the information in the columns is enough to draw a molecule, an instance of the own class (e.g. *Zmat*) is returned. If the information in the columns is not enough to draw a molecule, there are two cases to consider:

- A `Series` instance is returned for one dimensional slices.
- A `DataFrame` instance is returned in all other cases.

This means that:

`molecule.loc[:, ['atom', 'b', 'bond', 'a', 'angle', 'd', 'dihedral']]` returns a *Zmat*.

`molecule.loc[:, ['atom', 'bond']]` returns a `pandas.DataFrame`.

`molecule.loc[:, 'atom']` returns a `pandas.Series`.

Comparison:

Comparison for equality with == is supported. It behaves exactly like the equality comparison of DataFrames in pandas. Amongst other things this means that the index has to be the same and the comparison of floating point numbers is exact and not numerical.

Chemical Methods

__init__(frame[, metadata, _metadata])	How to initialize a Zmat instance.
add_data([new_cols])	Adds a column with the requested data.
change_numbering([new_index])	Change numbering to a new index.
has_same_sumformula(other)	Determines if `other` has the same sumformula
get_cartesian()	Return the molecule in cartesian coordinates.
to_xyz(*args, **kwargs)	Deprecated, use *get_cartesian()*
get_total_mass()	Returns the total mass in g/mol.
subs(symb_expr, value[, perform_checks])	Substitute a symbolic expression in `['bond', 'angle', 'dihedral']`
iupacify()	Give the IUPAC conform representation.
minimize_dihedrals()	Give a representation of the dihedral with minimized absolute value.

chemcoord.Zmat.__init__

`Zmat.`**`__init__`**(*frame*, *metadata=None*, *_metadata=None*)

How to initialize a Zmat instance.

Parameters

- **`init`** (*pd.DataFrame*) – A Dataframe with at least the columns `['atom', 'b', 'bond', 'a', 'angle', 'd', 'dihedral']`. Where `'atom'` is a string for the elementsymbol.
- **`order_of_definition`** (*list like*) – Specify in which order the Zmatrix is defined. If `None` it just uses `self.index`.

Returns A new zmat instance.

Return type *Zmat*

chemcoord.Zmat.add_data

`Zmat.`**`add_data`**(*new_cols=None*)

Adds a column with the requested data.

If you want to see for example the mass, the colormap used in jmol and the block of the element, just use:

```
['mass', 'jmol_color', 'block']
```

The underlying `pd.DataFrame` can be accessed with `constants.elements`. To see all available keys use `constants.elements.info()`.

The data comes from the module mendeleev written by Lukasz Mentel.

Please note that I added three columns to the mendeleev data:

```
['atomic_radius_cc', 'atomic_radius_gv', 'gv_color',
    'valency']
```

The `atomic_radius_cc` is used by default by this module for determining bond lengths. The three others are taken from the MOLCAS grid viewer written by Valera Veryazov.

Parameters

- **new_cols** (*str*) – You can pass also just one value. E.g. `'mass'` is equivalent to `['mass']`. If `new_cols` is `None` all available data is returned.
- **inplace** (*bool*) –

Returns

Return type *Cartesian*

chemcoord.Zmat.change_numbering

`Zmat.`**change_numbering**(*new_index=None*)

Change numbering to a new index.

Changes the numbering of index and all dependent numbering (bond_with...) to a new_index.

The user has to make sure that the new_index consists of distinct elements.

Parameters **new_index** (*list*) – If None the new_index is taken from 1 to the number of atoms.

Returns Reindexed version of the zmatrix.

Return type *Zmat*

chemcoord.Zmat.has_same_sumformula

`Zmat.`**has_same_sumformula**(*other*)

Determines if `other` has the same sumformula

Parameters **other** (*molecule*) –

Returns

Return type bool

chemcoord.Zmat.get_cartesian

`Zmat.`**get_cartesian**()

Return the molecule in cartesian coordinates.

Raises an *InvalidReference* exception, if the reference of the i-th atom is undefined.

Parameters **None** –

Returns Reindexed version of the zmatrix.

Return type *Cartesian*

chemcoord.Zmat.to_xyz

Zmat.**to_xyz**(**args*, ***kwargs*)
Deprecated, use *get_cartesian()*

chemcoord.Zmat.get_total_mass

Zmat.**get_total_mass**()
Returns the total mass in g/mol.

Parameters **None** –

Returns

Return type float

chemcoord.Zmat.subs

Zmat.**subs**(*symb_expr*, *value*, *perform_checks=True*)
Substitute a symbolic expression in `['bond', 'angle', 'dihedral']`

This is a wrapper around the substitution mechanism of sympy. Any symbolic expression in the columns `['bond', 'angle', 'dihedral']` of `self` will be substituted with value.

Parameters

- **symb_expr** (*sympy expression*) –
- **value** –
- **perform_checks** (*bool*) – If `perform_checks is True`, it is asserted, that the resulting Zmatrix can be converted to cartesian coordinates. Dummy atoms will be inserted automatically if necessary.

Returns Zmatrix with substituted symbolic expressions. If all resulting sympy expressions in a column are numbers, the column is recasted to 64bit float.

Return type *Zmat*

chemcoord.Zmat.iupacify

Zmat.**iupacify**()
Give the IUPAC conform representation.

Mathematically speaking the angles in a zmatrix are representations of an equivalence class. We will denote an equivalence relation with $\sim$ and use α for an angle and δ for a dihedral angle. Then the following equations hold true.

$$\begin{aligned}(\alpha, \delta) &\sim (-\alpha, \delta + \pi) \\ \alpha &\sim \alpha \mod 2\pi \\ \delta &\sim \delta \mod 2\pi\end{aligned}$$

IUPAC defines a designated representation of these equivalence classes, by asserting:

$$\begin{aligned}0 \leq &\alpha \leq \pi \\ -\pi \leq &\delta \leq \pi\end{aligned}$$

Parameters **None** –

Returns Zmatrix with accordingly changed angles and dihedrals.

Return type *Zmat*

chemcoord.Zmat.minimize_dihedrals

Zmat.**minimize_dihedrals**()

Give a representation of the dihedral with minimized absolute value.

Mathematically speaking the angles in a zmatrix are representations of an equivalence class. We will denote an equivalence relation with $\sim$ and use α for an angle and δ for a dihedral angle. Then the following equations hold true.

$$(\alpha, \delta) \sim (-\alpha, \delta + \pi)$$
$$\alpha \sim \alpha \mod 2\pi$$
$$\delta \sim \delta \mod 2\pi$$

This function asserts:

$$-\pi \leq \delta \leq \pi$$

The main application of this function is the construction of a transforming movement from `zmat1` to `zmat2`. This is under the assumption that `zmat1` and `zmat2` are the same molecules (regarding their topology) and have the same construction table (*get_construction_table()*):

```
with cc.TestOperators(False):
    D = zm2 - zm1
    zmats1 = [zm1 + D * i / n for i in range(n)]
    zmats2 = [zm1 + D.minimize_dihedrals() * i / n for i in range(n)]
```

The movement described by `zmats1` might be too large, because going from $5°$ to $355°$ is $350°$ in this case and not $-10°$ as in `zmats2` which is the desired Δ in most cases.

Parameters **None** –

Returns Zmatrix with accordingly changed angles and dihedrals.

Return type *Zmat*

Selection of data

loc	Label based indexing for obtaining elements.
safe_loc	Label based indexing for obtaining elements and assigning values safely.
unsafe_loc	Label based indexing for obtaining elements and assigning values unsafely.
iloc	Integer position based indexing for obtaining elements.
safe_iloc	Integer position based indexing for obtaining elements and assigning values safely.
unsafe_iloc	Integer position based indexing for obtaining elements and assigning values unsafely.

chemcoord.Zmat.loc

Zmat.**loc**
Label based indexing for obtaining elements.

In the case of obtaining elements, the indexing behaves like Indexing and Selecting data in Pandas.

For assigning elements it is necessary to make a explicit decision between safe and unsafe assignments. The differences are explained in the stub page of *safe_loc()*.

chemcoord.Zmat.safe_loc

Zmat.**safe_loc**
Label based indexing for obtaining elements and assigning values safely.

In the case of obtaining elements, the indexing behaves like Indexing and Selecting data in Pandas.

chemcoord.Zmat.unsafe_loc

Zmat.**unsafe_loc**
Label based indexing for obtaining elements and assigning values unsafely.

In the case of obtaining elements, the indexing behaves like Indexing and Selecting data in Pandas.

For assigning elements it is necessary to make a explicit decision between safe and unsafe assignments. The differences are explained in the stub page of *safe_loc()*.

chemcoord.Zmat.iloc

Zmat.**iloc**
Integer position based indexing for obtaining elements.

In the case of obtaining elements, the indexing behaves like Indexing and Selecting data in Pandas.

For assigning elements it is necessary to make a explicit decision between safe and unsafe assignments. The differences are explained in the stub page of *safe_loc()*.

chemcoord.Zmat.safe_iloc

Zmat.**safe_iloc**
Integer position based indexing for obtaining elements and assigning values safely.

In the case of obtaining elements, the indexing behaves like Indexing and Selecting data in Pandas.

For assigning elements it is necessary to make a explicit decision between safe and unsafe assignments. The differences are explained in the stub page of *safe_loc()*.

chemcoord.Zmat.unsafe_iloc

Zmat.**unsafe_iloc**
Integer position based indexing for obtaining elements and assigning values unsafely.

In the case of obtaining elements, the indexing behaves like Indexing and Selecting data in Pandas.

For assigning elements it is necessary to make a explicit decision between safe and unsafe assignments. The differences are explained in the stub page of *safe_loc()*.

Pandas DataFrame Wrapper

copy()	
index	Returns the index.
columns	Returns the columns.
Zmat.sort_index([axis, level, ascending, ...])	Sort object by labels (along an axis)
insert(loc, column, value[, ...])	Insert column into molecule at specified location.
sort_values(by[, axis, ascending, kind, ...])	Sort by the values along either axis

chemcoord.Zmat.copy

Zmat.**copy**()

chemcoord.Zmat.index

Zmat.**index**
Returns the index.

Wrapper around the pandas.DataFrame.index() property.

chemcoord.Zmat.columns

Zmat.**columns**
Returns the columns.

Wrapper around the pandas.DataFrame.columns() property.

chemcoord.Zmat.sort_index

Zmat.**sort_index**(*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*, *by=None*)
Sort object by labels (along an axis)

Wrapper around the pandas.DataFrame.sort_index() method.

chemcoord.Zmat.insert

Zmat.**insert**(*loc*, *column*, *value*, *allow_duplicates=False*, *inplace=False*)
Insert column into molecule at specified location.

Wrapper around the pandas.DataFrame.insert() method.

chemcoord.Zmat.sort_values

Zmat.**sort_values**(*by*, *axis=0*, *ascending=True*, *kind='quicksort'*, *na_position='last'*)
Sort by the values along either axis

Wrapper around the `pandas.DataFrame.sort_values()` method.

IO

to_zmat([buf, upper_triangle, ...])	Write zmat-file
write(*args, **kwargs)	Deprecated, use *to_zmat()*
read_zmat(inputfile[, implicit_index])	Reads a zmat file.
to_string([buf, format_abs_ref_as, ...])	Render a DataFrame to a console-friendly tabular output.
to_latex([buf, upper_triangle])	Render a DataFrame to a tabular environment table.

chemcoord.Zmat.to_zmat

Zmat.**to_zmat**(*buf=None*, *upper_triangle=True*, *implicit_index=True*, *float_format=<built-in method format of str object>*, *overwrite=True*, *header=False*)
Write zmat-file

Parameters

- **buf** (*str*) – StringIO-like, optional buffer to write to
- **implicit_index** (*bool*) – If implicit_index is set, the zmat indexing is changed to `range(1, len(self) + 1)`. Using *change_numbering()* Besides the index is omitted while writing which means, that the index is given implicitly by the row number.
- **float_format** (*one-parameter function*) – Formatter function to apply to column's elements if they are floats. The result of this function must be a unicode string.
- **overwrite** (*bool*) – May overwrite existing files.

Returns string (or unicode, depending on data and options)

Return type formatted

chemcoord.Zmat.write

Zmat.**write**(**args*, ***kwargs*)
Deprecated, use *to_zmat()*

chemcoord.Zmat.read_zmat

Zmat.**read_zmat**(*inputfile*, *implicit_index=True*)
Reads a zmat file.

Lines beginning with # are ignored.

Parameters

- **inputfile** (*str*) –

- **implicit_index** (*bool*) – If this option is true the first column
- **to be the element symbols for the atoms.** (*has*) – The row number is used to determine the index.

Returns

Return type *Zmat*

chemcoord.Zmat.to_string

Zmat.**to_string**(*buf=None*, *format_abs_ref_as='string'*, *upper_triangle=True*, *header=True*, *index=True*, ***kwargs*)

Render a DataFrame to a console-friendly tabular output.

Wrapper around the `pandas.DataFrame.to_string()` method.

chemcoord.Zmat.to_latex

Zmat.**to_latex**(*buf=None*, *upper_triangle=True*, ***kwargs*)

Render a DataFrame to a tabular environment table.

You can splice this into a LaTeX document. Requires `\usepackage{booktabs}`. Wrapper around the `pandas.DataFrame.to_latex()` method.

Attributes

columns	Returns the columns.
index	Returns the index.
shape	Returns the shape.
dtypes	Returns the dtypes.

chemcoord.Zmat.shape

Zmat.**shape**

Returns the shape.

Wrapper around the `pandas.DataFrame.shape()` property.

chemcoord.Zmat.dtypes

Zmat.**dtypes**

Returns the dtypes.

Wrapper around the `pandas.DataFrame.dtypes()` property.

zmat_functions

A collection of functions operating on instances of *Zmat*.

DummyManipulation(dummy_manipulation_allowed)	Contextmanager that controls the behaviour of *safe_loc()* and *safe_iloc()*.

chemcoord.DummyManipulation

class chemcoord.**DummyManipulation**(*dummy_manipulation_allowed*, *cls=None*)

Contextmanager that controls the behaviour of *safe_loc()* and *safe_iloc()*.

In the following examples it is assumed, that using the assignment with *safe_loc()* would lead to an invalid reference. Then there are two possible usecases:

```
with DummyManipulation(zmat, True):
    zmat.safe_loc[...] = ...
    # This inserts required dummy atoms and removes them,
    # if they are not needed anymore.
    # Removes only dummy atoms, that were automatically inserted.

with DummyManipulation(zmat, False):
    zmat.safe_loc[...] = ...
    # This raises an exception
    # :class:`~chemcoord.exceptions.InvalidReference`.
    # which can be handled appropiately.
    # The zmat instance is unmodified, if an exception was raised.
```

__init__(*dummy_manipulation_allowed*, *cls=None*)

Configuration of settings

The current settings of chemcoord can be seen with cc.settings. This is a dictionary that can be changed in place. If it is necessary to change these settings permamently there is the possibility to write a configuration file of the current settings, that is read automatically while importing the module. The configuration file is in the INI format and can be changed with any text editor.

The possible settings and their defaults are:

['defaults']

['atomic_radius_data'] = 'atomic_radius_cc' Determines which atomic radius is used for calculating if atoms are bonded

['use_lookup_internally'] = True Look into *get_bonds()* for an explanation

['viewer'] = 'gv.exe' Which one is the default viewer used in *chemcoord.Cartesian.view()* and *chemcoord.xyz_functions.view()*.

write_configuration_file([filepath, overwrite])	Create a configuration file.
read_configuration_file([filepath])	Read the configuration file.

chemcoord.configuration.write_configuration_file

chemcoord.configuration.**write_configuration_file**(*filepath='/home/docs/.chemcoordrc'*, *overwrite=False*)

Create a configuration file.

Writes the current state of settings into a configuration file.

Note: Since a file is permamently written, this function is strictly speaking not sideeffect free.

Parameters

- **filepath** (*str*) – Where to write the file. The default is under both UNIX and Windows `~/.chemcoordrc`.
- **overwrite** (*bool*) –

Returns

Return type None

chemcoord.configuration.read_configuration_file

chemcoord.configuration.**read_configuration_file**(*filepath='/home/docs/.chemcoordrc'*)

Read the configuration file.

Note: This function changes `cc.settings` inplace and is therefore not sideeffect free.

Parameters **filepath** (*str*) – Where to read the file. The default is under both UNIX and Windows `~/.chemcoordrc`.

Returns

Return type None

Exceptions

InvalidReference([message, i, b, a, d, ...])	Raised when the i-th atom uses an invalid reference.
UndefinedCoordinateSystem([message])	
PhysicalMeaning([message])	
IllegalArgumentCombination	

chemcoord.exceptions.InvalidReference

exception chemcoord.exceptions.**InvalidReference**(*message=None*, *i=None*, *b=None*, *a=None*, *d=None*, *already_built_cartesian=None*, *zmat_after_assignment=None*)

Raised when the i-th atom uses an invalid reference.

May carry several attributes:

- `i`: Index of the atom with an invalid refernce.
- `b`, `a`, and `d`: Indices of reference atoms.
- `already_built_cartesian`: The cartesian of all atoms up to (i-1)
- `zmat_after_assignment`: Attached information if it was raised from the safe assignment methods (`Zmat.safe_loc()` and `Zmat.unsafe_loc()`).

chemcoord.exceptions.UndefinedCoordinateSystem

exception chemcoord.exceptions.**UndefinedCoordinateSystem**(*message=''*)

chemcoord.exceptions.PhysicalMeaning

exception chemcoord.exceptions.**PhysicalMeaning**(*message=''*)

chemcoord.exceptions.IllegalArgumentCombination

exception chemcoord.exceptions.**IllegalArgumentCombination**

References

Bugreports and Development

If you request new feautures or want to report bugs please open an issue on the github project page.

If you want to contribute in the development, feel free to contact me as well over the github project page.

Previous Contribution

- Main Work: Oskar Weser
- Python2 compatibility: Keld Lundgaard

License

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or

b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.

2. Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.

2. Accompany the Combined Work with a copy of the GNU GPL and this license document.

c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.

4. Do one of the following:

0. Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

   1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

   e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and

execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

Index

Symbols

A

B

C

D

F

G