CAT Documentation

Release 1.1.1.dev0

B. F. van Beek
J. Belic

Jul 11, 2023

CONTENTS

Contents:

CHAPTER

ONE

COMPOUND ATTACHMENT TOOL

CAT is a collection of tools designed for the construction of various chemical compounds. Further information is provided in the documentation.

1.1 Package installation

CAT can be installed via pip as following:

- **CAT**: `pip install nlesc-CAT --upgrade`

Note that, while not strictly necessary, it is recommended to first create a conda environment:

- Download and install miniconda for python3: miniconda (also you can install the complete anaconda version).
- Create a new virtual environment: `conda create --name CAT python`
- Activate the environment:: `conda activate CAT`

1.1.1 Input files

Running **CAT** and can be done with the following command: `init_cat my_settings.yaml`. The user merely has to provide a yaml file with the job settings, settings which can be tweaked and altered to suit ones purposes (see example1). Alternatively, **CAT** can be run like a regular python script, bypassing the command-line interface (*i.e.* `python input.py`, see example2).

An extensive description of the various available settings is available in the documentation.

1.1.2 References

- Belić, J.; van Beek, B.; Menzel, J. P.; Buda, F.; Visscher, L. Systematic Computational Design and Optimization of Light Absorbing Dyes. *J. Phys. Chem. A* **2020**, *124* (31), 6380–6388.
- van Beek, B.; Zito, J.; Visscher, L.; Infante, I. CAT: A Compound Attachment Tool for the construction of composite chemical compounds. *J. Chem. Inf. Model.* (submitted).

CHAPTER

TWO

CAT DOCUMENTATION

For a more detailed description of the **CAT** compound builder read the documentation. The documentation is divided into three parts: The basics, further details about the input cores & ligands and finally a more detailed look into the customization of the various jobs.

2.1 General Overview & Getting Started

A basic recipe for running **CAT**:

1. Create two directories named 'core' and 'ligand'. The 'core' directory should contain the input cores & the 'ligand' should contain the input ligands. The quantum dots will be exported to the 'QD' directory.

2. Customize the job settings to your liking, see CAT/examples/input_settings.yaml for an example. Note: everything under the `optional` section does **not** have to be included in the input settings. As is implied by the name, everything in `optional` is completely optional.

3. Run **CAT** with the following command: `init_cat input_settings.yaml`

4. Congratulations, you just ran **CAT**!

The default **CAT** settings, at various levels of verbosity, are provided below.

2.1.1 Default Settings

```
path: None

input_cores:
    - Cd68Se55.xyz:
        guess_bonds: False

input_ligands:
    - OC(C)=O
    - OC(CC)=O
```

2.1.2 Verbose default Settings

```
path: None

input_cores:
    - Cd68Se55.xyz:
        guess_bonds: False

input_ligands:
    - OC(C)=O
    - OC(CC)=O

optional:
    database:
        dirname: database
        read: True
        write: True
        overwrite: False
        thread_safe: False
        mol_format: (pdb, xyz)
        mongodb: False

    core:
        dirname: core
        anchor: Cl
        subset: null

    ligand:
        dirname: ligand
        optimize: True
        split: True
        anchor: null
        cosmo-rs: False

    qd:
        dirname: qd
        construct_qd: True
        optimize: False
        bulkiness: False
        activation_strain: False
        dissociate: False
```

2.1.3 Maximum verbose default Settings

```
path: None

input_cores:
    - Cd68Se55.xyz:
        guess_bonds: False

input_ligands:
```

(continues on next page)

(continued from previous page)

```
    - OC(C)=O
    - OC(CC)=O

optional:
    database:
        dirname: database
        read: (core, ligand, qd)
        write: (core, ligand, qd)
        overwrite: False
        thread_safe: False
        mol_format: (pdb, xyz)
        mongodb: False

    core:
        dirname: core
        anchor: Cl
        subset: null

    ligand:
        dirname: ligand
        split: True
        anchor: null
        cosmo-rs: False
        optimize:
            use_ff: False
            job1: null
            s1: null
            job2: null
            s2: null

    qd:
        dirname: qd
        construct_qd: True
        optimize: False
        bulkiness: False
        activation_strain: False
        dissociate:
            core_atom: Cd
            lig_count: 2
            keep_files: True
            core_core_dist: 5.0
            lig_core_dist: 5.0
            topology: {}

            job1: False
            s1: False
            job2: False
            s2: False
```

2.2 path

2.2.1 Default Settings

```
path: null
```

2.2.2 Arguments

path

Parameter

- **Type** - str or NoneType
- **Default value** – None

The path were all working directories are/will be stored. To use the current working directory, use one of the following values: None, ".", "" or "path_to_workdir".

Note: The yaml format uses null rather than None as in Python.

2.3 input_cores & input_ligands

Thia section related relates the importing and processing of cores and ligands. Ligand & cores can be imported from a wide range of different files and files types, which can roughly be divided into three categories:

1. Files containing coordinates of a single molecule: .xyz, .pdb & .mol files.
2. Python objects: plams.Molecule, rdkit.Chem.Mol & SMILES strings (str).
3. Containers with one or multiple input molecules: directories & .txt files.

In the later case, the container can consist of multiple SMILES strings or paths to .xyz, .pdb and/or .mol files. If necessary, containers are searched recursively. Both absolute and relative paths are explored.

2.3.1 Default Settings

```
input_cores:
    - Cd68Se55.xyz:
        guess_bonds: False

input_ligands:
    - OC(C)=O
    - OC(CC)=O
    - OC(CCC)=O
    - OC(CCCC)=O
```

2.3.2 Optional arguments

`.guess_bonds`

Parameter

- **Type** - `bool`
- **Default value** – `False`

Try to guess bonds and bond orders in a molecule based on the types atoms and the relative of atoms. Is set to False by default, with the exception of .xyz files.

`.column`

Parameter

- **Type** - `int`
- **Default value** – `0`

The column containing the to be imported molecules. Relevant when importing structures from .txt and .xlsx files with multiple columns. Relevant for .txt and .csv files. Numbering starts from 0.

`.row`

Parameter

- **Type** - `int`
- **Default value** – `0`

The first row in a column which contains a molecule. Useful for when, for example, the very first row contains the title of aforementioned row, in which case row = 1 would be a sensible choice. Relevant for .txt and .csv files. Numbering starts from 0.

`.canonicalize`

Parameter

- **Type** - `bool`
- **Default value** – `False` for cores and `True` for ligands

Whether the atom order of the passed molecules should be canonicalized.

`.indices`

Parameter

- **Type** - `int` or `tuple` [`int`]
- **Default value** – `None`

The behaviour of this argument depends on whether it is passed to a molecule in *input_cores* or *input_ligands*:

`input_cores`

Manually specify the atomic index of one ore more atom(s) in the core that will be replaced with ligands. If left empty, all atoms of a user-specified element (see `optional.cores.dummy`) will be replaced with ligands.

input_ligands

Manually specify the atomic index of the ligand atom that will be attached to core (implying argument_dict: *optional.ligand.split* = `False`). If two atomic indices are provided (*e.g.* `(1, 2)`), the bond between atoms `1` and `[2]` will be broken and the remaining molecule containing atom `2` is attached to the core, (implying argument_dict: *split* = `True`). Serves as an alternative to the functional group based `CAT.find_substructure()` function, which identifies the to be attached atom based on connectivity patterns (*i.e.* functional groups).

Note: Atom numbering follows the PLAMS [1, 2] convention of starting from 1 rather than 0.

2.4 Optional

There are a number of arguments which can be used to modify the functionality and behavior of the quantum dot builder. Herein an overview is provided.

Note: Inclusion of this section in the input file is not required, assuming one is content with the default settings.

2.4.1 Index

Option	Description
optional.database.dirname	The name of the directory where the database will be stored.
optional.database.read	Attempt to read results from the database before starting calculations.
optional.database.write	Export results to the database.
optional.database.overwrite	Allow previous results in the database to be overwritten.
optional.database.thread_safe	Ensure that the created workdir has a thread-safe name.
optional.database.mol_format	The file format(s) for exporting moleculair structures.
optional.database.mongodb	Options related to the MongoDB format.
optional.core.dirname	The name of the directory where all cores will be stored.
optional.core.anchor	Atomic number of symbol of the core anchor atoms.
optional.core.allignment	How the to-be attached ligands should be alligned with the core.
optional.core.subset	Settings related to the partial replacement of core anchor atoms.
optional.ligand.dirname	The name of the directory where all ligands will be stored.
optional.ligand.optimize	Optimize the geometry of the to-be attached ligands.
optional.ligand.anchor	Manually specify SMILES strings representing functional groups.
optional.ligand.split	If the ligand should be attached in its entirety to the core or not.
optional.ligand.cosmo-rs	Perform a property calculation with COSMO-RS on the ligand.
optional.ligand.cdft	Perform a conceptual DFT calculation with ADF on the ligand.
optional.ligand.cone_angle	Compute the smallest enclosing cone angle within a ligand.
optional.ligand.branch_distance	Compute the size of branches and their distance w.r.t. to the anchor within a ligand.
optional.qd.dirname	The name of the directory where all quantum dots will be stored.
optional.qd.construct_qd	Whether or not the quantum dot should actually be constructed or not.
optional.qd.optimize	Optimize the quantum dot (i.e. core + all ligands).
optional.qd.multi_ligand	A workflow for attaching multiple non-unique ligands to a single quantum dot.
optional.qd.bulkiness	Calculate the V_{bulk}, a ligand- and core-sepcific descriptor of a ligands' bulkiness.
optional.qd.activation_strain	Perform an activation strain analyses.
optional.qd.dissociate	Calculate the ligand dissociation energy.

2.4.2 Default Settings

```
optional:
    database:
        dirname: database
        read: True
        write: True
        overwrite: False
        thread_safe: False
        mol_format: (pdb, xyz)
        mongodb: False

    core:
        dirname: core
```

(continues on next page)

(continued from previous page)

```
        anchor: Cl
        allignment: surface
        subset: null

    ligand:
        dirname: ligand
        optimize: True
        anchor: null
        split: True
        cosmo-rs: False
        cdft: False
        cone_angle: False

    qd:
        dirname: qd
        construct_qd: True
        optimize: False
        activation_strain: False
        dissociate: False
        bulkiness: False
```

2.4.3 Arguments

Database

optional.**database**

All database-related settings.

Note: For *optional.database* settings to take effect the Data-CAT package has to be installed.

Example:

```
optional:
    database:
        dirname: database
        read: True
        write: True
        overwrite: False
        mol_format: (pdb, xyz)
        mongodb: False
```

optional.database.**dirname**

Parameter

- **Type** - str

- **Default Value** - `"database"`

The name of the directory where the database will be stored.

The database directory will be created (if it does not yet exist) at the path specified in *path*.

`optional.database.`**`read`**

Parameter

- **Type** - `bool`, `str` or `tuple` [`str`]
- **Default value** - `("core", "ligand", "qd")`

Attempt to read results from the database before starting calculations.

Before optimizing a structure, check if a geometry is available from previous calculations. If a match is found, use that structure and avoid any geometry (re-)optimizations. If one wants more control then the boolean can be substituted for a list of strings (*i.e.* `"core"`, `"ligand"` and/or `"qd"`), meaning that structures will be read only for a specific subset.

Example

Example #1:

```yaml
optional:
    database:
        read: (core, ligand, qd)  # This is equivalent to read: True
```

Example #2:

```yaml
optional:
    database:
        read: ligand
```

`optional.database.`**`write`**

Parameter

- **Type** - `bool`, `str` or `tuple` [`str`]
- **Default value** - `("core", "ligand", "qd")`

Export results to the database.

Previous results will **not** be overwritten unless *`optional.database.overwrite`* = `True`. If one wants more control then the boolean can be substituted for a list of strings (*i.e.* `"core"`, `"ligand"` and/or `"qd"`), meaning that structures written for a specific subset.

See *`optional.database.read`* for a similar relevant example.

`optional.database.`**`overwrite`**

Parameter

- **Type** - `bool`, `str` or `tuple` [`str`]
- **Default value** - `False`

Allow previous results in the database to be overwritten.

Only applicable if *optional.database.write* = `True`. If one wants more control then the boolean can be substituted for a list of strings (*i.e.* `"core"`, `"ligand"` and/or `"qd"`), meaning that structures written for a specific subset.

See *optional.database.read* for a similar relevant example.

`optional.database.`**`thread_safe`**

Parameter

- **Type** - `bool`
- **Default value** - `False`

Ensure that the created workdir has a thread-safe name.

Note that this disables the restarting of partially completed jobs.

`optional.database.`**`mol_format`**

Parameter

- **Type** - `bool`, `str` or `tuple` [`str`]
- **Default value** - `("pdb", "xyz")`

The file format(s) for exporting moleculair structures.

By default all structures are stored in the .hdf5 format as (partially) de-serialized .pdb files. Additional formats can be requested with this keyword. Accepted values: `"pdb"`, `"xyz"`, `"mol"` and/or `"mol2"`.

`optional.database.`**`mongodb`**

Parameter

- **Type** - `bool` or `dict`
- **Default Value** – `False`

Options related to the MongoDB format.

See also

More extensive options for this argument are provided in *The Database Class*:.

Core

`optional.`**`core`**

All settings related to the core.

Example:

```
optional:
    core:
        dirname: core
        anchor: Cl
```

(continues on next page)

(continued from previous page)

```
        allignment: surface
        subset: null
```

optional.core.**dirname**

Parameter

- **Type** - str
- **Default value** – "core"

The name of the directory where all cores will be stored.

The core directory will be created (if it does not yet exist) at the path specified in *path*.

optional.core.**anchor**

Parameter

- **Type** - str or int
- **Default value** – 17

Atomic number of symbol of the core anchor atoms.

The atomic number or atomic symbol of the atoms in the core which are to be replaced with ligands. Alternatively, anchor atoms can be manually specified with the core_indices variable.

Further customization can be achieved by passing a dictionary:

- *anchor.group*
- *anchor.group_idx*
- *anchor.group_format*
- *anchor.remove*

Note:

```
optional:
    core:
        anchor:
            group: "[H]Cl"  # Remove HCl and attach at previous Cl position
            group_idx: 1
            group_format: "SMILES"
            remove: [0, 1]
```

---

optional.core.**allignment**

Parameter

- **Type** - str
- **Default value** – "surface"

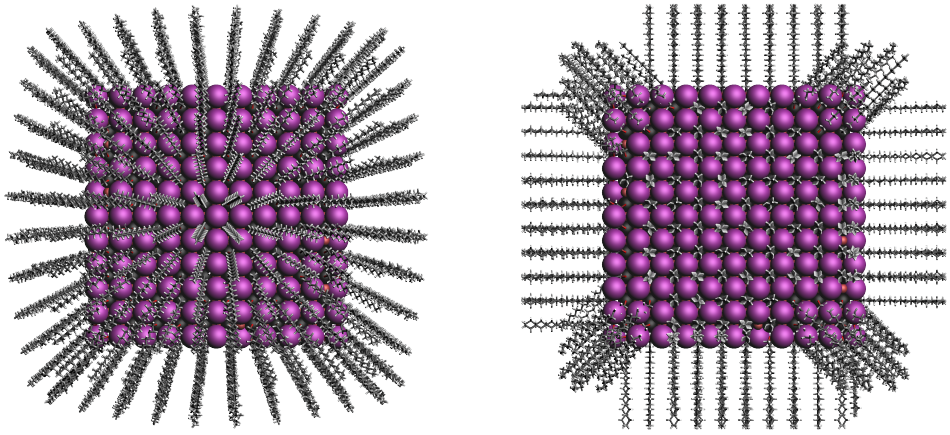
How the to-be attached ligands should be alligned with the core.

Has four allowed values:

- `"surface"`: Define the core vectors as those orthogonal to the cores surface. Not this option requires at least four core anchor atoms. The surface is herein defined by a convex hull constructed from the core.
- `"sphere"`: Define the core vectors as those drawn from the core anchor atoms to the cores center.
- `"anchor"`: Define the core vectors based on the optimal vector of its anchors. Only available in when the core contains molecular anchors, *e.g.* acetates.
- `"surface invert"`/`"surface_invert"`: The same as `"surface"`, except the core vectors are inverted.
- `"sphere invert"`/`"sphere_invert"`: The same as `"sphere"`, except the core vectors are inverted.
- `"anchor invert"`/`"anchor_invert"`: The same as `"anchor"`, except the core vectors are inverted.

Note that for a spherical core both approaches are equivalent.

Note: An example of a `"sphere"` (left) and `"surface"` (right) allignment.



`optional.core.`**`subset`**

Parameter

- **Type** - `dict`, optional
- **Default value** – `None`

Settings related to the partial replacement of core anchor atoms with ligands.

If not `None`, has access to six further keywords, the first two being the most important:

- *`subset.f`*
- *`subset.mode`*

- *subset.follow_edge*
- *subset.weight*
- *subset.randomness*
- *subset.cluster_size*

optional.core.subset.**f**

Parameter

- **Type** - float

The fraction of core anchor atoms that will actually be exchanged for ligands.

The provided value should satisfy the following condition: $0 < f \leq 1$.

Note: This argument has no value be default and must thus be provided by the user.

---

optional.core.subset.**mode**

Parameter

- **Type** - str
- **Default value** – "uniform"

Defines how the anchor atom subset, whose size is defined by the fraction f, will be generated.

Accepts one of the following values:

- "uniform": A uniform distribution; the nearest-neighbor distances between each successive anchor atom and all previous anchor atoms is maximized. can be combined with *subset.cluster_size* to create a uniform distribution of clusters of a user-specified size.
- "cluster": A clustered distribution; the nearest-neighbor distances between each successive anchor atom and all previous anchor atoms is minimized.
- "random": A random distribution.

It should be noted that all three methods converge towards the same set as f approaches 1.0.

If $\boldsymbol{D} \in \mathbb{R}_{+}^{n,n}$ is the (symmetric) distance matrix constructed from the anchor atom superset and $\boldsymbol{a} \in \mathbb{N}^{m}$ is the vector of indices which yields the anchor atom subset. The definition of element a_i is defined below for the "uniform" distribution. All elements of $\boldsymbol{a}$ are furthermore constrained to be unique.

$$*\text{argmin} a_i = \begin{cases} _{k \in \mathbb{N}} \sum_{\hat{\imath}=0}^{n} f\left(D_{k,\hat{\imath}}\right) & \text{if} \\ i = 0 & \\ _{k \in \mathbb{N}} \sum_{\hat{\imath}=0}^{i-1} f\left(D[k, a_{\hat{\imath}}]\right) & \text{if} \\ i > 0 & \end{cases} \quad \text{with} \quad f(x) = e^{-x} \tag{2.1}$$

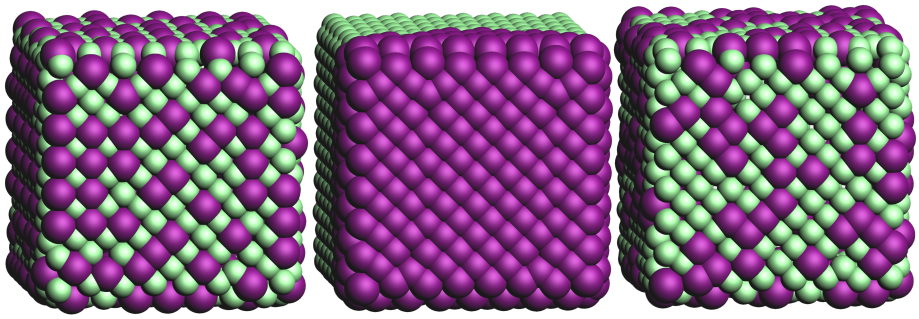
For the "cluster" distribution all argmin operations are exchanged for argmax.

The old default, the p-norm with $p = -2$, is equivalent to:

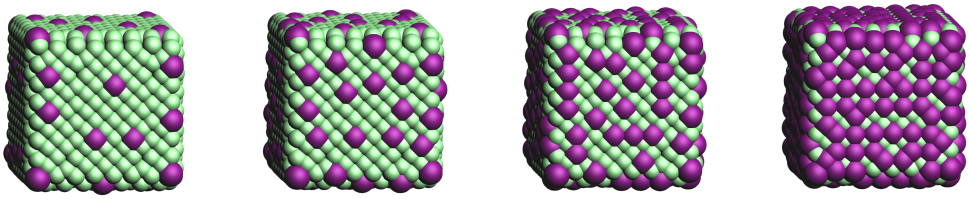
$$*\text{argmax}_{k \in \mathbb{N}} \sum_{\hat{\imath}=0}^{n} f\left(D_{k,\hat{\imath}}\right) = _{k \in \mathbb{N}} \left(\sum_{\hat{\imath}=0}^{n} |D_{k,\hat{\imath}}|^{p}\right)^{1/p} \quad \text{if} \quad f(x) = x^{-2} \tag{2.2}$$

Note that as the elements of $\boldsymbol{D}$ were defined as positive or zero-valued real numbers; operating on $\boldsymbol{D}$ is thus equivalent to operating on its absolute.

Note: An example of a `"uniform"`, `"cluster"` and `"random"` distribution with $f = 1/3$.



Note: An example of four different `"uniform"` distributions at $f = 1/16$, $f = 1/8$, $f = 1/4$ and $f = 1/2$.



`optional.core.subset.`**`follow_edge`**

Parameter

- **Type** - `bool`
- **Default value** – `False`

Construct the anchor atom distance matrix by following the shortest path along the edges of a (triangular-faced) polyhedral approximation of the core rather than the shortest path through space.

Enabling this option will result in more accurate `"uniform"` and `"cluster"` distributions at the cost of increased computational time.

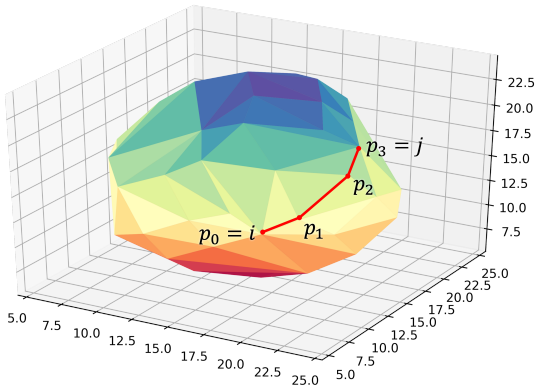
Given the matrix of Cartesian coordinates $\boldsymbol{X} \in \mathbb{R}^{n,3}$, the matching edge-distance matrix $\boldsymbol{D}^{\text{edge}} \in \mathbb{R}_{+}^{n,n}$ and the vector $\boldsymbol{p} \in \mathbb{N}^{m}$, representing a (to-be optimized) path as the indices of edge-connected vertices, then element $D_{i,j}^{\text{edge}}$ is defined as following:

$$D_{i,j}^{\text{edge}} = \min_{\boldsymbol{p} \in \mathbb{N}^m; m \in \mathbb{N}} \sum_{k=0}^{m-1} ||X_{p_k,:} - X_{p_{k+1},:}|| \quad \text{with} \quad p_0 = i \quad \text{and} \quad p_m = j \tag{2.3}$$

The polyhedron edges are constructed, after projecting all vertices on the surface of a sphere, using Qhull's `ConvexHull` algorithm (The Quickhull Algorithm for Convex Hulls). The quality of the constructed edges is proportional to the convexness of the core, more specifically: how well the vertices can be projected on a spherical surface without severely distorting the initial structure. For

example, spherical, cylindrical or cuboid cores will yield reasonably edges, while the edges resulting from torus will be extremely poor.

Note: An example of a cores' polyhedron-representation; displaying the shortest path between points i and j.



`optional.core.subset.`**`cluster_size`**

Parameter

- **Type** - `int` or `Iterable` [`int`]
- **Default value** – 1

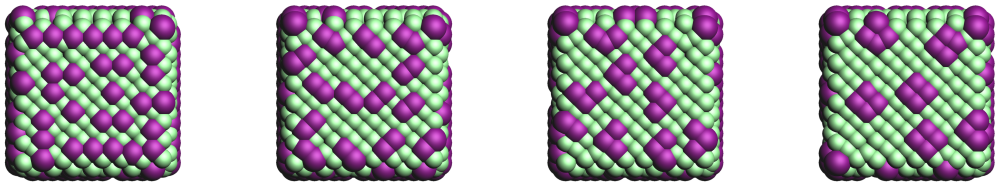
Allow for the creation of uniformly distributed clusters of size r; should be used in conjunction with *`subset.mode = "uniform"`*.

The value of r can be either a single cluster size (*e.g.* `cluster_size = 5`) or an iterable of various sizes (*e.g.* `cluster_size = [2, 3, 4]`). In the latter case the iterable will be repeated as long as necessary.

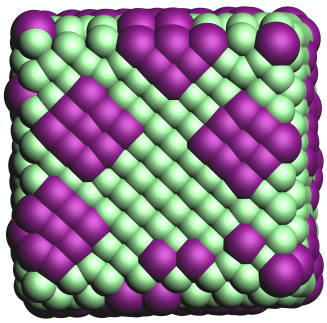
Compared to Eq (2.2) the vector of indices $\boldsymbol{a} \in \mathbb{N}^m$ is, for the purpose of book keeping, reshaped into the matrix $\boldsymbol{A} \in \mathbb{N}^{q,r}$ with $q * r = m$. All elements of $\boldsymbol{A}$ are, again, constrained to be unique.

$$
*\text{argmin} A_{i,j} = \begin{cases} k \in \mathbb{N} \sum_{\hat{i}=0}^{n} f\left(D[k, \hat{i}]\right) & \text{if} \\ i = 0 & \text{and} \\ j = 0 & \\ k \in \mathbb{N} \sum_{\hat{i}=0}^{i-1} \sum_{\hat{j}=0}^{r} f\left(D[k, A_{\hat{i},\hat{j}}]\right) & \text{if} \\ i > 0 & \text{and} \\ j = 0 & \\ k \in \mathbb{N} \dfrac{\sum_{\hat{i}=0}^{i-1} \sum_{\hat{j}=0}^{r} f\left(D[k, A_{\hat{i},\hat{j}}]\right)}{\sum_{\hat{j}=0}^{j-1} f\left(D[k, A_{i,\hat{j}}]\right)} & \\ & \text{if} \\ j > 0 & \end{cases} \tag{2.4}
$$

Note: An example of various cluster sizes (1, 2, 3 and 4) with $f = 1/4$.



Note: An example of clusters of varying size (`cluster_size = [1, 2, 9, 1]`) with $f = 1/4$.



`optional.core.subset.`**`weight`**

Parameter

- **Type** - `str`
- **Default value** – `"numpy.exp(-x)"`

The function $f(x)$ for weighting the distance.; its default value corresponds to: $f(x) = e^{-x}$.

For the old default, the p-norm with $p = -2$, one can use `weight = "x**-2"`: $f(x) = x^{-}2$.

Custom functions can be specified as long as they satisfy the following constraints:

- The function must act an variable by the name of `x`, a 2D array of positive and/or zero-valued floats ($x \in \mathbb{R}_{+}^{n,n}$).
- The function must take a single array as argument and return a new one.
- The function must be able to handle values of `numpy.nan` and `numpy.inf` without raising exceptions.
- The shape and data type of the output array should not change with respect to the input.

Modules specified in the weight function will be imported when required, illustrated here with SciPy's `expit` function: `weight = "scipy.special.expit(x)"` aka `weight = "1 / (1 + numpy.exp(-x))"`

Multi-line statements are allowed: `weight = "a = x**2; b = 5 * a; numpy.exp(b)"`. The last part of the statement is assumed to be the to-be returned value (*i.e.* `return numpy.exp(b)`).

optional.core.subset.**randomness**

Parameter

- **Type** - float, optional
- **Default value** – None

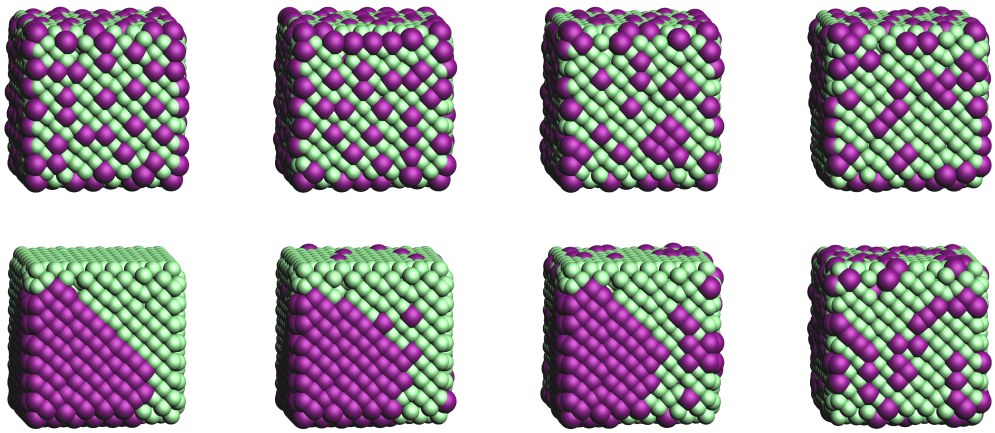
The probability that each new core anchor atom will be picked at random.

Can be used in combination with "uniform" and "cluster" to introduce a certain degree of randomness (*i.e.* entropy).

If not None, the provided value should satisfy the following condition: $0 \leq randomness \leq 1$. A value of 0 is equivalent to a "uniform" / "cluster" distribution while 1 is equivalent to "random".

Note: A demonstration of the randomness parameter for a "uniform" and "cluster" distribution at $f = 1/4$.

The randomness values are (from left to right) set to 0, $1/4$, $1/2$ and 1.



Ligand

optional.**ligand**

All settings related to the ligands.

Example:

```yaml
optional:
    ligand:
        dirname: ligand
        optimize: True
        anchor: null
        split: True
        cosmo-rs: False
```

(continues on next page)

(continued from previous page)

```
        cdft: False
        cone_angle: False
        branch_distance: False
```

optional.ligand.**dirname**

Parameter

- **Type** - `str`
- **Default value** – `"ligand"`

The name of the directory where all ligands will be stored.

The ligand directory will be created (if it does not yet exist) at the path specified in *path*.

optional.ligand.**optimize**

Parameter

- **Type** - `bool` or `dict`
- **Default value** – `True`

Optimize the geometry of the to-be attached ligands.

The ligand is split into one or multiple (more or less) linear fragments, which are subsequently optimized (RDKit UFF [1, 2, 3]) and reassembled while checking for the optimal dihedral angle. The ligand fragments are biased towards more linear conformations to minimize inter-ligand repulsion once the ligands are attached to the core.

After the conformation search a final (unconstrained) geometry optimization is performed, RDKit UFF again being the default level of theory. Custom job types and settings can, respectivelly, be specified with the `job2` and `s2` keys.

Note:

```
optional:
    ligand:
        optimize:
            job2: ADFJob
```

---

optional.ligand.**anchor**

Parameter

- **Type** - `str`, `Sequence[str]` or `dict[str, Any]`
- **Default value** – `None`

Manually specify SMILES strings representing functional groups.

For example, with *optional.ligand.anchor* = `("O[H]", "[N+].[Cl-]")` all ligands will be searched for the presence of hydroxides and ammonium chlorides.

The first atom in each SMILES string (*i.e.* the "anchor") will be used for attaching the ligand to the core, while the last atom (assuming *optional.ligand.split* = True) will be dissociated from the ligand and discarded.

If not specified, the default functional groups of **CAT** are used.

This option can alternatively be provided as `optional.ligand.functional_groups`.

Further customization can be achieved by passing dictionaries:

- *anchor.group*
- *anchor.group_idx*
- *anchor.group_format*
- *anchor.remove*
- *anchor.kind*
- *anchor.angle_offset*
- *anchor.dihedral*
- *anchor.multi_anchor_filter*

Note:

```yaml
optional:
    ligand:
        anchor:
            - group: "[H]OC(=O)C"  # Remove H and attach at the (formal)␣
↪oxyanion
              group_idx: 1
              remove: 0
            - group: "[H]OC(=O)C"  # Remove H and attach at the mean␣
↪position of both oxygens
              group_idx: [1, 3]
              remove: 0
              kind: mean
```

Note: This argument has no value be default and will thus default to SMILES strings of the default functional groups supported by **CAT**.

Note: The yaml format uses `null` rather than `None` as in Python.

`optional.ligand.anchor.group`

Parameter

- **Type** - `str`

A SMILES string representing the anchoring group.

Note: This argument has no value be default and must thus be provided by the user.

---

optional.ligand.anchor.**group_idx**

Parameter

- **Type** - `int` or `Sequence[int]`

The indices of the anchoring atom(s) in *anchor.group*.

Indices should be 0-based. These atoms will be attached to the core, the manner in which is determined by the `anchor.kind` option.

Note: This argument has no value be default and must thus be provided by the user.

---

optional.ligand.anchor.**group_format**

Parameter

- **Type** - `str`
- **Default value** – `"SMILES"`

The format used for representing *anchor.group*.

Defaults to the SMILES format. The supported formats (and matching RDKit parsers) are as following:

```
>>> import rdkit.Chem

>>> FASTA      = rdkit.Chem.MolFromFASTA
>>> HELM       = rdkit.Chem.MolFromHELM
>>> INCHI      = rdkit.Chem.MolFromInchi
>>> MOL2       = rdkit.Chem.MolFromMol2Block
>>> MOL2_FILE  = rdkit.Chem.MolFromMol2File
>>> MOL        = rdkit.Chem.MolFromMolBlock
>>> MOL_FILE   = rdkit.Chem.MolFromMolFile
>>> PDB        = rdkit.Chem.MolFromPDBBlock
>>> PDB_FILE   = rdkit.Chem.MolFromPDBFile
>>> PNG        = rdkit.Chem.MolFromPNGString
>>> PNG_FILE   = rdkit.Chem.MolFromPNGFile
>>> SVG        = rdkit.Chem.MolFromRDKitSVG
>>> SEQUENCE   = rdkit.Chem.MolFromSequence
>>> SMARTS     = rdkit.Chem.MolFromSmarts
>>> SMILES     = rdkit.Chem.MolFromSmiles
>>> TPL        = rdkit.Chem.MolFromTPLBlock
>>> TPL_FILE   = rdkit.Chem.MolFromTPLFile
```

optional.ligand.anchor.**remove**

Parameter

- **Type** - `None`, `int` or `Sequence[int]`
- **Default value** – `None`

The indices of the to-be removed atoms in *anchor.group*.

No atoms are removed when set to `None`. Indices should be 0-based. See also the *split* option.

optional.ligand.anchor.**kind**

Parameter

- **Type** - str
- **Default value** – "first"

How atoms are to-be attached when multiple anchor atoms are specified in *anchor.group_idx*.

Accepts one of the following options:

- "first": Attach the first atom to the core.
- "mean": Attach the mean position of all anchoring atoms to the core.
- "mean_translate": Attach the mean position of all anchoring atoms to the core and then translate back to the first atom.

optional.ligand.anchor.**angle_offset**

Parameter

- **Type** - None, float or str
- **Default value** – None

Manually offset the angle of the ligand vector by a given number.

The plane of rotation is defined by the first three indices in *anchor.group_idx*.

By default the angle unit is assumed to be in degrees, but if so desired one can explicitly pass the unit: angle_offset: "0.25 rad".

optional.ligand.anchor.**dihedral**

Parameter

- **Type** - None, float or str
- **Default value** – None

Manually specify the ligands vector dihedral angle, rather than optimizing it w.r.t. the inter-ligand distance.

The dihedral angle is defined by three vectors:

- The first two in dices in *anchor.group_idx*.
- The core vector(s).
- The Cartesian X-axis as defined by the core.

By default the angle unit is assumed to be in degrees, but if so desired one can explicitly pass the unit: dihedral: "0.5 rad".

optional.ligand.anchor.**multi_anchor_filter**

Parameter

- **Type** - str
- **Default value** – "ALL"

How ligands with multiple valid anchor sites are to-be treated.

Accepts one of the following options:

- "all": Construct a new ligand for each valid anchor/ligand combination.

- `"first"`: Pick only the first valid functional group, all others are ignored.
- `"raise"`: Treat a ligand as invalid if it has multiple valid anchoring sites.

`optional.ligand.`**`split`**

Parameter

- **Type** - `bool`
- **Default value** – `True`

If `False`: The ligand is to be attached to the core in its entirety .

Before	After
$NR_4{}^+$	$NR_4{}^+$
O_2CR	O_2CR
HO_2CR	HO_2CR
H_3CO_2CR	H_3CO_2CR

`True`: A proton, counterion or functional group is to be removed from the ligand before attachment to the core.

Before	After
$Cl^- + NR_4{}^+$	$NR_4{}^+$
HO_2CR	O_2CR^-
$Na^+ + O_2CR^-$	O_2CR^-
HO_2CR	O_2CR^-
H_3CO_2CR	O_2CR^-

`optional.ligand.cosmo-rs`

Parameter

- **Type** - `bool` or `dict`
- **Default value** – `False`

Perform a property calculation with COSMO-RS [4, 5, 6, 7] on the ligand.

The COSMO surfaces are by default constructed using ADF MOPAC [8, 9, 10].

The solvation energy of the ligand and its activity coefficient are calculated in the following solvents: acetone, acetonitrile, dimethyl formamide (DMF), dimethyl sulfoxide (DMSO), ethyl acetate, ethanol, *n*-hexane, toluene and water.

`optional.ligand.`**`cdft`**

Parameter

- **Type** - `bool` or `dict`
- **Default value** – `False`

Perform a conceptual DFT (CDFT) calculation with ADF on the ligand.

All global descriptors are, if installed, stored in the database. This includes the following properties:

- Electronic chemical potential (mu)
- Electronic chemical potential (mu+)

- Electronic chemical potential (mu-)
- Electronegativity (chi=-mu)
- Hardness (eta)
- Softness (S)
- Hyperhardness (gamma)
- Electrophilicity index (w=omega)
- Dissocation energy (nucleofuge)
- Dissociation energy (electrofuge)
- Electrodonating power (w-)
- Electroaccepting power(w+)
- Net Electrophilicity
- Global Dual Descriptor Deltaf+
- Global Dual Descriptor Deltaf-

This block can be furthermore customized with one or more of the following keys:

- `"keep_files"`: Whether or not to delete the ADF output afterwards.
- `"job1"`: The type of PLAMS Job used for running the calculation. The only value that should be supplied here (if any) is `"ADFJob"`.
- `"s1"`: The job Settings used for running the CDFT calculation. Can be left blank to use the default template (`nanoCAT.cdft.cdft`).

Examples

```yaml
optional:
    ligand:
        cdft: True
```

```yaml
optional:
    ligand:
        cdft:
            job1: ADFJob
            s1: ...  # Insert custom settings here
```

`optional.ligand.`**`cone_angle`**

Parameter

- **Type** - `bool` or `dict`
- **Default value** – `False`

Compute the smallest enclosing cone angle within a ligand.

The smallest enclosing cone angle is herein defined as two times the largest angle ($2 * \phi_{max}$) w.r.t. a central ligand vector, the ligand vector in turn being defined as the vector that minimizes ϕ_{max}.

Examples

```yaml
optional:
    ligand:
        cone_angle: True
```

```yaml
optional:
    ligand:
        cone_angle:
            distance: [0, 0.5, 1, 1.5, 2]
```

---

optional.ligand.cone_angle.**distance**

Parameter

- **Type** - `float` or `list[float]`
- **Default value** – `0.0`

The distance in *cone_angle* of each ligands' anchor atom w.r.t. the nanocrystal surface.

Accepts one or more distances.

optional.ligand.**branch_distance**

Parameter

- **Type** - `bool`
- **Default value** – `False`

Compute the size of branches and their distance w.r.t. to the anchor within a ligand.

QD

optional.**qd**

All settings related to the quantum dots.

Example:

```yaml
optional:
    qd:
        dirname: qd
        construct_qd: True
        optimize: False
        bulkiness: False
        activation_strain: False
        dissociate: False
```

optional.qd.**dirname**

Parameter

- **Type** - str
- **Default value** – "qd"

The name of the directory where all quantum dots will be stored.

The quantum dot directory will be created (if it does not yet exist) at the path specified in *path*.

optional.qd.**construct_qd**

Parameter

- **Type** - bool
- **Default value** – True

Whether or not the quantum dot should actually be constructed or not.

Setting this to False will still construct ligands and carry out ligand workflows, but it will not construct the actual quantum dot itself.

optional.qd.**optimize**

Parameter

- **Type** - bool or dict
- **Default value** – False

Optimize the quantum dot (i.e. core + all ligands) .

By default the calculation is performed with ADF UFF [3, 11]. The geometry of the core and ligand atoms directly attached to the core are frozen during this optimization.

optional.qd.**multi_ligand**

Parameter

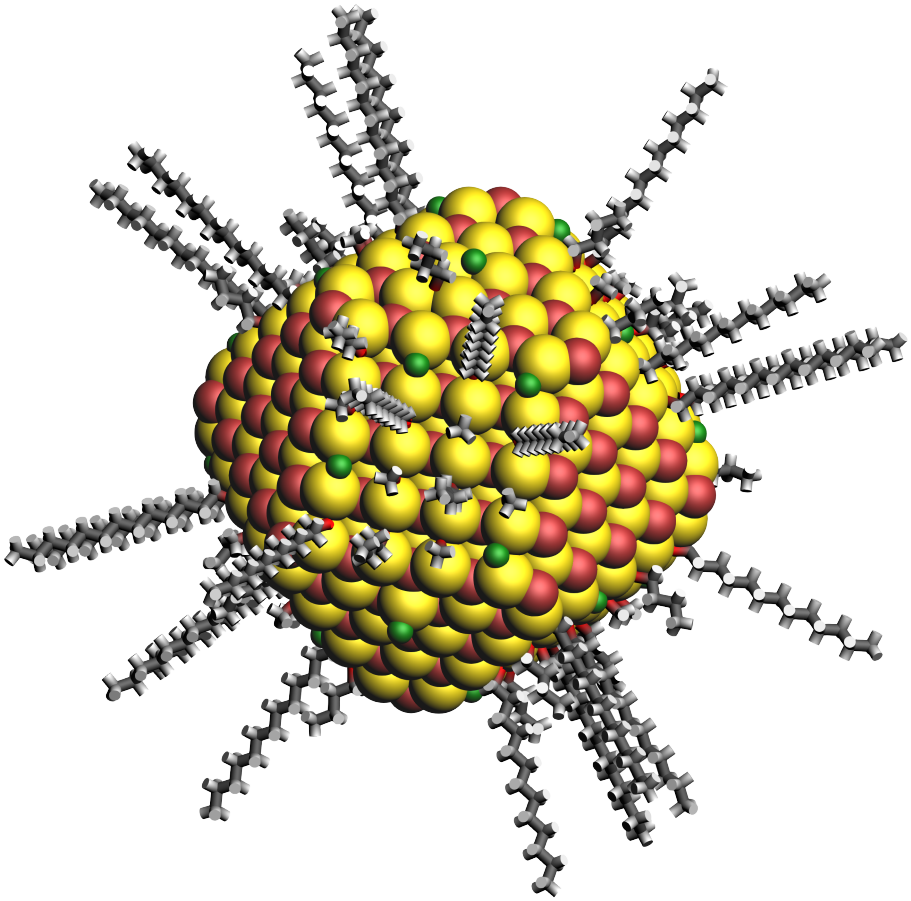
- **Type** - None or dict
- **Default value** – None

A workflow for attaching multiple non-unique ligands to a single quantum dot.

Note that this is considered a seperate workflow besides the normal ligand attachment. Consequently, these structures will *not* be passed to further workflows.

See *Multi-ligand attachment* for more details regarding the available options.

Note: An example with [O-]CCCC as main ligand and [O-]CCCCCCCCCCCC & [O-]C as additional ligands.



---

optional.qd.**bulkiness**

Parameter

- **Type** - bool or dict
- **Default value** – False

Calculate the V_{bulk}, a ligand- and core-specific descriptor of a ligands' bulkiness.

Supplying a dictionary grants access to the two additional *h_lim* and *d* sub-keys.

$$V(r_i, h_i; d, h_{lim}) = \sum_{i=1}^{n} e^{r_i} (\frac{2r_i}{d} - 1)^{+} (1 - \frac{h_i}{h_{lim}})^{+} \tag{2.5}$$

optional.qd.bulkiness.**h_lim**

Parameter

- **Type** - float or None
- **Default value** – 10.0

Default value of the h_{lim} parameter in *bulkiness*.

Set to None to disable the h_{lim}-based cutoff.

optional.qd.bulkiness.**d**

Parameter

- **Type** - float/list[float], None or "auto"
- **Default value** – "auto"

Default value of the d parameter in *bulkiness*.

Set to "auto" to automatically infer this parameters value based on the mean nearest-neighbor distance among the core anchor atoms. Set to None to disable the d-based cutoff. Supplying multiple floats will compute the bulkiness for all specified values.

optional.qd.**activation_strain**

Parameter

- **Type** - bool or dict
- **Default value** – False

Perform an activation strain analysis [12, 13, 14].

The activation strain analysis (kcal mol^{-1}) is performed on the ligands attached to the quantum dot surface with RDKit UFF [1, 2, 3].

The core is removed during this process; the analysis is thus exclusively focused on ligand deformation and inter-ligand interaction. Yields three terms:

1. dE_{strain} : The energy required to deform the ligand from their equilibrium geometry to the geometry they adopt on the quantum dot surface. This term is, by definition, destabilizing. Also known as the preparation energy (dE_{prep}).

2. dE_{int} : The mutual interaction between all deformed ligands. This term is characterized by the non-covalent interaction between ligands (UFF Lennard-Jones potential) and, depending on the inter-ligand distances, can be either stabilizing or destabilizing.

3. dE : The sum of dE_{strain} and dE_{int}. Accounts for both the destabilizing ligand deformation and (de-)stabilizing interaction between all ligands in the absence of the core.

See *Ensemble-Averaged Activation Strain Analysis* for more details.

optional.qd.**dissociate**

Parameter

- **Type** - bool or dict
- **Default value** – False

Calculate the ligand dissociation energy.

Calculate the ligand dissociation energy (BDE) of ligands attached to the surface of the core. See *Bond Dissociation Energy* for more details. The calculation consists of five distinct steps:

1. Dissociate all combinations of n ligands (Y) and an atom from the core (X) within a radius *r* from aforementioned core atom. The dissociated compound has the general structure of XY_n.

2. Optimize the geometry of XY_n at the first level of theory (1). Default: ADF MOPAC [1, 2, 3].

3. Calculate the "electronic" contribution to the BDE (ΔE) at the first level of theory (1): ADF MOPAC [1, 2, 3]. This step consists of single point calculations of the complete quantum dot, XY_n and all XY_n-dissociated quantum dots.

4. Calculate the thermochemical contribution to the BDE ($\Delta\Delta G$) at the second level of theory (2). Default: ADF UFF [4, 5]. This step consists of geometry optimizations and frequency analyses of the same compounds used for step 3.

5. $\Delta G_{tot} = \Delta E_1 + \Delta\Delta G_2 = \Delta E_1 + (\Delta G_2 - \Delta E_2)$.

See also

More extensive options for this argument are provided in *Bond Dissociation Energy*:.

2.5 Bond Dissociation Energy

Calculate the bond dissociation energy (BDE) of ligands attached to the surface of the core. The calculation consists of five distinct steps:

1. Dissociate all combinations of n ligands (Y, see `optional.qd.dissociate.lig_count`) a nd an atom from the core (X, see `optional.qd.dissociate.core_atom`) within a radius r from aforementioned core atom (see `optional.qd.dissociate.lig_core_dist` and `optional.qd.dissociate.core_core_dist`). The dissociated compound has the general structure of XY_n.

2. Optimize the geometry of XY_n at the first level of theory (1). Default: ADF MOPAC [1, 2, 3].

3. Calculate the "electronic" contribution to the BDE (ΔE) at the first level of theory (1): ADF MOPAC [1, 2, 3]. This step consists of single point calculations of the complete quantum dot, XY_n and all XY_n-dissociated quantum dots.

4. Calculate the thermalchemical contribution to the BDE ($\Delta\Delta G$) at the second level of theory (2). Default: ADF UFF [4, 5]. This step consists of geometry optimizations and frequency analyses of the same compounds used for step 3.

5. $\Delta G_{tot} = \Delta E_1 + \Delta\Delta G_2 = \Delta E_1 + (\Delta G_2 - \Delta E_2)$.

2.5.1 Default Settings

```yaml
optional:
    qd:
        dissociate:
            core_atom: Cd
            core_index: null
            lig_count: 2
            core_core_dist: 5.0  # Ångström
            lig_core_dist: 5.0  # Ångström
            lig_core_pairs: 1
            topology: {}

            keep_files: True
            job1: AMSJob
            s1: True
            job2: AMSJob
            s2: True
```

2.5.2 Arguments

optional.qd.**dissociate**

```yaml
optional:
    qd:
        dissociate:
            core_atom: Cd
            core_index: null
            lig_count: 2
            lig_pairs: 1
            core_core_dist: null  # Ångström
            lig_core_dist: 5.0  # Ångström
            topology:
                7: vertice
                8: edge
                10: face
```

optional.qd.dissociate.**core_atom**

Parameter

- **Type** - str or int

The atomic number or atomic symbol of the core atoms (X) which are to be dissociated. The core atoms are dissociated in combination with n ligands (Y, see *dissociate.lig_count*). Yields a compound with the general formula XY_n.

Atomic indices can also be manually specified with *dissociate.core_index*

If one is interested in dissociating ligands in combination with a molecular species (*e.g.* $X = NR_4{}^+$) the atomic number (or symbol) can be substituted for a SMILES string represting a poly-atomic ion (*e.g.* tetramethyl ammonium: C[N+](C)(C)C).

If a SMILES string is provided it must satisfy the following 2 requirements:

1. The SMILES string *must* contain a single charged atom; unpredictable behaviour can occur otherwise.
2. The provided structure (including its bonds) must be present in the core.

Warning: This argument has no value be default and thus *must* be provided by the user.

optional.qd.dissociate.**lig_count**

Parameter

- **Type** - int

The number of ligands, n, which is to be dissociated in combination with a single core atom (X, see *dissociate.core_atom*).

Yields a compound with the general formula XY_n.

> **Warning:** This argument has no value be default and thus *must* be provided by the user.

optional.qd.dissociate.**core_index**

Parameter

- **Type** - `int` or `list` [`int`], optional
- **Default value** – `None`

Alternative to *dissociate.lig_core_dist* and *dissociate.core_atom*. Manually specify the indices of all to-be dissociated atoms in the core. Core atoms will be dissociated in combination with the n closest ligands.

Note: Atom numbering follows the PLAMS [1, 2] convention of starting from 1 rather than 0.

Note: The yaml format uses `null` rather than `None` as in Python.

---

optional.qd.dissociate.**core_core_dist**

Parameter

- **Type** - `float` or `int`, optional
- **Default value** – `None`

The maximum to be considered distance (Ångström) between atoms in *dissociate.core_atom*. Used for determining the topology of the core atom

(see *dissociate.topology*) and whether it is exposed to the surface of the core or not. It is recommended to use a radius which encapsulates a single (complete) shell of neighbours.

If not specified (or equal to `0.0`) **CAT** will attempt to guess a suitable value based on the cores' radial distribution function.

optional.qd.dissociate.**lig_core_dist**

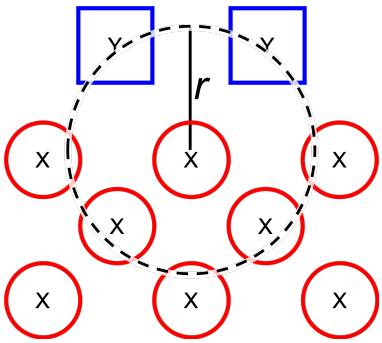
Parameter

- **Type** - `float` or `int`, optional
- **Default value** – `None`

Dissociate all combinations of a single core atom (see *dissociate.core_atom*) and the n closests ligands within a user-specified radius.

Serves as an alternative to *dissociate.lig_core_dist*, which removes a set number of combinations rather than everything withing a certain radius.

The number of ligands dissociated in combination with a single core atom is controlled by *dissociate.lig_count*.



optional.qd.dissociate.**lig_pairs**

Parameter

- **Type** - int, optional
- **Default value** – None

Dissociate a user-specified number of combinations of a single core atom (see *dissociate.core_atom*) and the n closests ligands.

Serves as an alternative to *dissociate.lig_core_dist*, removing a preset number of (closest) pairs rather than all combinations within a certain radius.

The number of ligands dissociated in combination with a single core atom is controlled by *dissociate.lig_count*.

optional.qd.dissociate.**topology**

Parameter

- **Type** - dict
- **Default value** – {}

A dictionary which translates the number neighbouring core atoms (see *dissociate.core_atom* and *dissociate.core_core_dist*) into a topology. Keys represent the number of neighbours, values represent the matching topology.

Example

Given a *dissociate.core_core_dist* of 5.0 Ångström, the following options can be interpreted as following:

```
optional:
    qd:
        dissociate:
            7: vertice
            8: edge
            10: face
```

Core atoms with 7 other neighbouring core atoms (within a radius of 5.0 Ångström) are marked as "vertice", the ones with 8 neighbours are marked as "edge" and the ones with 10 neighbours as

"face".

optional.qd.dissociate.**qd_opt**

Parameter

- **Type** - bool
- **Default value** – False

Whether to optimize the quantum dot and XY_n -dissociated quantum dot.

2.5.3 Arguments - Job Customization

optional.qd.**dissociate**

```
optional:
    qd:
        dissociate:
            keep_files: True
            job1: AMSJob
            s1: True
            job2: AMSJob
            s2: True
```

optional.qd.dissociate.**keep_files**

Parameter

- **Type** - bool
- **Default value** – True

Whether to keep or delete all BDE files after all calculations are finished.

optional.qd.dissociate.**xyn_pre_opt**

Parameter

- **Type** - bool
- **Default value** – True

Pre-optimize the XY_n fragment with UFF.

Note: Requires AMS.

optional.qd.dissociate.**job1**

Parameter

- **Type** - type, str or bool
- **Default value** – AMSJob

A type object of a Job subclass, used for calculating the "electronic" component (ΔE_1) of the bond dissociation energy. Involves single point calculations.

Alternatively, an alias can be provided for a specific job type (see *Type Aliases*).

Setting it to True will default to AMSJob, while False is equivalent to *optional.qd.dissociate* = False.

optional.qd.dissociate.**s1**

Parameter

- **Type** - dict, str or bool
- **Default value** – See below

```
s1:
    input:
        mopac:
            model: PM7
        ams:
            system:
                charge: 0
```

The job settings used for calculating the "electronic" component (ΔE_1) of the bond dissociation energy.

Alternatively, a path can be provided to .json or .yaml file containing the job settings.

Setting it to True will default to the ["MOPAC"] block in CAT/data/templates/qd.yaml, while False is equivalent to *optional.qd.dissociate* = False.

optional.qd.dissociate.**job2**

Parameter

- **Type** - type, str or bool
- **Default value** – AMSJob

A type object of a Job subclass, used for calculating the thermal component ($\Delta\Delta G_2$) of the bond dissociation energy. Involves a geometry reoptimizations and frequency analyses.

Alternatively, an alias can be provided for a specific job type (see *Type Aliases*).

Setting it to True will default to AMSJob, while False will skip the thermochemical analysis completely.

optional.qd.dissociate.**s2**

Parameter

- **Type** - dict, str or bool
- **Default value** – See below

```
s2:
    input:
        uff:
            library: uff
        ams:
            system:
                charge: 0
                bondorders:
                    _1: null
```

The job settings used for calculating the thermal component ($\Delta\Delta G_2$) of the bond dissociation energy.

Alternatively, a path can be provided to .json or .yaml file containing the job settings.

Setting it to `True` will default to the the *MOPAC* block in CAT/data/templates/qd.yaml, while `False` will skip the thermochemical analysis completely.

Index

dissociate_ligand(mol, lig_count[, ...])	Remove XY_n from **mol** with the help of the *MolDissociater* class.
MolDissociater(mol, core_idx, ligand_count)	The *MolDissociater* class; serves as an API for *dissociate_ligand()*.
MolDissociater.remove_bulk([max_vec_len])	Remove out atoms specified in *MolDissociater.core_idx* which are present in the bulk.
MolDissociater.assign_topology()	Assign a topology to all core atoms in *MolDissociater.core_idx*.
MolDissociater.get_pairs_closest(lig_idx[, ...])	Create and return the indices of each core atom and the n closest ligands.
MolDissociater.get_pairs_distance(lig_idx[, ...])	Create and return the indices of each core atom and all ligand pairs with **max_dist**.
MolDissociater.combinations(cor_lig_pairs[, ...])	Create a list with all to-be removed atom combinations.
MolDissociater.__call__(combinations)	Start the dissociation process.

API

nanoCAT.bde.dissociate_xyn.**dissociate_ligand**(*mol*, *lig_count*, *lig_core_pairs=1*, *lig_core_dist=None*, *core_atom=None*, *core_index=None*, *core_smiles=None*, *core_core_dist=None*, *topology=None*, ***kwargs*)

Remove XY_n from **mol** with the help of the *MolDissociater* class.

The dissociation process consists of 5 general steps:

- Constructing a *MolDissociater* instance for managing the dissociation workflow.
- Assigning a topology-descriptor to each atom with *MolDissociater.assign_topology()*.
- Identifying all valid core/ligand pairs using either *MolDissociater.get_pairs_closest()* or *MolDissociater.get_pairs_distance()*.
- Creating all to-be dissociated core/ligand combinations with `MolDissociater.get_combinations()`.
- Start the dissociation process by calling the earlier created *MolDissociater* instance.

Examples

```
>>> from typing import Iterator

>>> import numpy as np
>>> from scm.plams import Molecule

# Define parameters
>>> mol = Molecule(...)
>>> core_idx = [1, 2, 3, 4, 5]
>>> lig_idx = [10, 20, 30, 40]
>>> lig_count = 2

# Start the workflow
>>> dissociate = MolDissociater(mol, core_idx, lig_count)
>>> dissociate.assign_topology()
>>> pairs: np.ndarray = dissociate.get_pairs_closest(lig_idx)
>>> combinations: Iterator[tuple] = dissociate.get_combinations(pairs)

# Create the final iterator
>>> mol_iterator: Iterator[Molecule] = dissociate(cor_lig_combinations)
```

Parameters

- **mol** (*plams.Molecule*) – A molecule.
- **lig_count** (*int*) – The number of to-be dissociated ligands per core atom/molecule.
- **lig_core_pairs** (*int*, optional) – The number of to-be dissociated core/ligand pairs per core atom. Core/ligand pairs are picked based on whichever ligands are closest to each core atom. This option is irrelevant if a distance based criterium is used (see **lig_dist**).
- **lig_core_dist** (*float*, optional) – Instead of dissociating a given number of core/ligand pairs (see **lig_pairs**) dissociate all pairs within a given distance from a core atom.
- **core_index** (*int* or *Iterable* [*int*]) – An index or set of indices with all to-be dissociated core atoms. See **core_atom** to define **core_idx** based on a common atomic symbol/number.
- **core_atom** (*int* or *str*, optional) – An atomic number or symbol used for automatically defining **core_idx**. Core atoms within the bulk (rather than on the surface) are ignored.
- **core_smiles** (*str*, optional) – A SMILES string representing molecule containing **core_idx**. Provide a value here if one wants to disociate an entire molecules from the core and not just atoms.
- **core_core_dist** (*float*, optional) – A value representing the mean distance between the core atoms in **core_idx**. If `None`, guess this value based on the radial distribution function of **mol** (this is generally recomended).
- **topology** (*Mapping* [*int*, *str*], optional) – A mapping neighbouring of atom counts to a user specified topology descriptor (*e.g.* `"edge"`, `"vertice"` or `"face"`).
- ****kwargs** (*Any*) – For catching excess keyword arguments.

Returns

A generator yielding new molecules with XY_n removed.

Return type
Generator [plams.Molecule]

Raises
TypeError – Raised if **core_atom** and **core_idx** are both None or **lig_core_pairs** and **lig_core_dist** are both None.

class nanoCAT.bde.dissociate_xyn.**MolDissociater**(*mol*, *core_idx*, *ligand_count*, *max_dist=None*, *topology=None*)

The *MolDissociater* class; serves as an API for *dissociate_ligand()*.

Parameters

- **mol** (plams.Molecule) – A PLAMS molecule consisting of cores and ligands. See *MolDissociater.mol*.
- **core_idx** (int or Iterable [int]) – An iterable with (1-based) atomic indices of all core atoms valid for dissociation. See *MolDissociater.core_idx*.
- **ligand_count** (int) – The number of ligands to-be dissociation with a single atom from *MolDissociater.core_idx*. See *MolDissociater.ligand_count*.
- **max_dist** (float, optional) – The maximum distance between core atoms for them to-be considered neighbours. If None, this value will be guessed based on the radial distribution function of **mol**. See *MolDissociater.ligand_count*.
- **topology** (dict [int, str], optional) – A mapping of neighbouring atom counts to a user-specified topology descriptor. See *MolDissociater.topology*.

mol

A PLAMS molecule consisting of cores and ligands.

Type
plams.Molecule

core_idx

An iterable with (1-based) atomic indices of all core atoms valid for dissociation.

Type
int or Iterable [int]

ligand_count

The number of ligands to-be dissociation with a single atom from *MolDissociater.core_idx*.

Type
int

max_dist

The maximum distance between core atoms for them to-be considered neighbours. If None, this value will be guessed based on the radial distribution function of *MolDissociater.mol*.

Type
float, optional

topology

A mapping of neighbouring atom counts to a user-specified topology descriptor.

Type
dict [int, str], optional

MolDissociater.**remove_bulk**(*max_vec_len=0.5*)

Remove out atoms specified in *MolDissociater.core_idx* which are present in the bulk.

The function searches for all neighbouring core atoms within a radius *MolDissociater.max_dist*. Vectors are then constructed from the core atom to the mean positioon of its neighbours. Vector lengths close to 0 thus indicate that the core atom is surrounded in a (nearly) spherical pattern, *i.e.* it's located in the bulk of the material and not on the surface.

Performs in inplace update of *MolDissociater.core_idx*.

Parameters

max_vec_len (float) – The maximum length of an atom vector to-be considered part of the bulk. Atoms producing smaller values are removed from *MolDissociater.core_idx*. Units are in Angstroem.

MolDissociater.**assign_topology**()

Assign a topology to all core atoms in *MolDissociater.core_idx*.

The topology descriptor is based on:

- The number of neighbours within a radius defined by *MolDissociater.max_dist*.
- The mapping defined in *MolDissociater.topology*, which maps the number of neighbours to a user-defined topology description.

If no topology description is available for a particular neighbouring atom count, then a generic `f"{i}_neighbours"` descriptor is used (where *i* is the neighbouring atom count).

Performs an inplace update of all `Atom.properties.topology` values.

MolDissociater.**get_pairs_closest**(*lig_idx*, *n_pairs=1*)

Create and return the indices of each core atom and the n closest ligands.

Parameters

- **lig_idx** (int or Iterable [int]) – The (1-based) indices of all ligand anchor atoms.
- **n_pairs** (int) – The number of to-be returned pairs per core atom. If `n_pairs > 1` than each successive set of to-be dissociated ligands is determined by the norm of the n distances.

Returns

A 2D array with the indices of all valid ligand/core pairs.

Return type

2D numpy.ndarray [int]

MolDissociater.**get_pairs_distance**(*lig_idx*, *max_dist=5.0*)

Create and return the indices of each core atom and all ligand pairs with **max_dist**.

Parameters

- **lig_idx** (int or Iterable [int]) – The (1-based) indices of all ligand anchor atoms.
- **max_dist** (float) – The radius (Angstroem) used as cutoff.

Returns

A 2D array with the indices of all valid ligand/core pairs.

Return type

2D numpy.ndarray [int]

MolDissociater.**combinations**(*cor_lig_pairs*, *lig_mapping=None*, *core_mapping=None*)

Create a list with all to-be removed atom combinations.

Parameters

- **cor_lig_pairs** (numpy.ndarray) – An array with the indices of all core/ligand pairs.
- **lig_mapping** (Mapping, optional) – A mapping for translating (1-based) atomic indices in `cor_lig_pairs[:, 0]` to lists of (1-based) atomic indices. Used for mapping ligand anchor atoms to the rest of the to-be dissociated ligands.
- **core_mapping** (Mapping, optional) – A mapping for translating (1-based) atomic indices in `cor_lig_pairs[:, 1:]` to lists of (1-based) atomic indices. Used for mapping core atoms to the to-be dissociated sub structures.

Returns

A set of 2-tuples. The first element of each tuple is a frozenset with the (1-based) indices of all to-be removed core atoms. The second element contains a frozenset with the (1-based) indices of all to-be removed ligand atoms.

Return type

set [tuple]

MolDissociater.**__call__**(*combinations*)

Start the dissociation process.

2.6 Type Aliases

Aliases are available for a large number of job types, allowing one to pass a str instead of a type object, thus simplifying the input settings for **CAT**. Aliases are insensitive towards capitalization (or lack thereof).

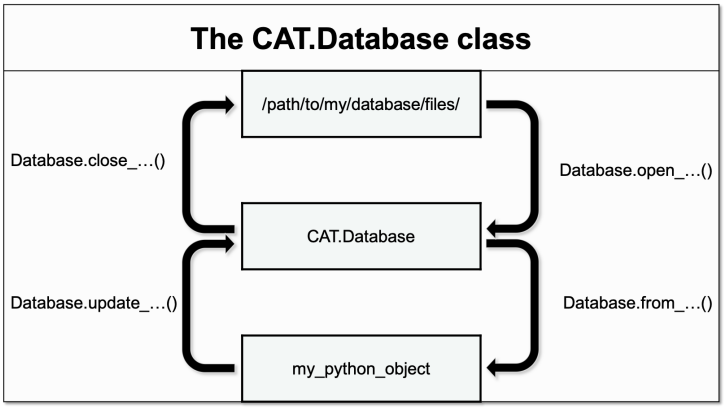
A comprehensive list of plams.Job subclasses and their respective aliases (*i.e.* str) is presented below.

2.6.1 Aliases

- ADFJob = "adf" = "adfjob"
- AMSJob = "ams" = "amsjob"
- UFFJob = "uff" = "uffjob"
- BANDJob = "band" = "bandjob"
- DFTBJob = "dftb" = "dftbjob"
- MOPACJob = "mopac" = "mopacjob"
- ReaxFFJob = "reaxff" = "reaxffjob"
- Cp2kJob = "cp2k" = "cp2kjob"
- ORCAJob = "orca" = "orcajob"
- DiracJob = "dirac" = "diracjob"
- GamessJob = "gamess" = "gamessjob"
- DFTBPlusJob = "dftbplus" = "dftbplusjob"
- CRSJob = "crs" = "cosmo-rs" = "crsjob"

2.7 The Database Class

A Class designed for the storing, retrieval and updating of results.



The methods of the Database class can be divided into three categories accoring to their functionality:

- Opening & closing the database - these methods serve as context managers for loading and unloading parts of the database from the harddrive.

  The context managers can be accessed by calling either *Database.csv_lig*, *Database.csv_qd*, or *Database.hdf5*, with the option of passing additional positional or keyword arguments.

```
>>> from dataCAT import Database

>>> database = Database()
>>> with database.csv_lig(write=False) as db:
>>>     print(repr(db))
DFProxy(ndframe=<pandas.core.frame.DataFrame at 0x7ff8e958ce80>)

>>> with database.hdf5('r') as db:
>>>     print(type(db))
<class 'h5py._hl.files.File'>
```

- Importing to the database - these methods handle the importing of new data from python objects to the Database class:

update_csv()	*update_hdf5()*	*update_mongodb()*

- Exporting from the database - these methods handle the exporting of data from the Database class to other python objects or remote locations:

from_csv()	*from_hdf5()*

2.7.1 Index

Database.dirname	Get the path+filename of the directory containing all database components.
Database.csv_lig	Get a function for constructing an *dataCAT.OpenLig* context manager.
Database.csv_qd	Get a function for constructing an *dataCAT.OpenQD* context manager.
Database.hdf5	Get a function for constructing a `h5py.File` context manager.
Database.mongodb	Get a mapping with keyword arguments for `pymongo.MongoClient`.
Database.update_mongodb([database, overwrite])	Export ligand or qd results to the MongoDB database.
Database.update_csv(df[, index, database, ...])	Update *Database.csv_lig* or *Database.csv_qd* with new settings.
Database.update_hdf5(df, index[, database, ...])	Export molecules (see the `"mol"` column in **df**) to the structure database.
Database.from_csv(df[, database, get_mol, ...])	Pull results from *Database.csv_lig* or *Database.csv_qd*.
Database.from_hdf5(index[, database, rdmol, ...])	Import structures from the hdf5 database as RDKit or PLAMS molecules.
DFProxy(ndframe)	A mutable wrapper providing a view of the underlying dataframes.
OpenLig(filename[, write])	Context manager for opening and closing the ligand database (*Database.csv_lig*).
OpenQD(filename[, write])	Context manager for opening and closing the QD database (*Database.csv_qd*).

2.7.2 API

class dataCAT.**Database**(*path=None*, *host='localhost'*, *port=27017*, ***kwargs*)

The Database class.

property dirname

Get the path+filename of the directory containing all database components.

property csv_lig

Get a function for constructing an *dataCAT.OpenLig* context manager.

Type
`Callable[..., dataCAT.OpenLig]`

property csv_qd

Get a function for constructing an *dataCAT.OpenQD* context manager.

Type
`Callable[..., dataCAT.OpenQD]`

property hdf5

Get a function for constructing a `h5py.File` context manager.

Type
`Callable[..., h5py.File]`

property mongodb

Get a mapping with keyword arguments for `pymongo.MongoClient`.

Type
`Mapping[str, Any]`, optional

update_mongodb(*database='ligand'*, *overwrite=False*)

Export ligand or qd results to the MongoDB database.

Examples

```
>>> from dataCAT import Database

>>> kwargs = dict(...)
>>> db = Database(**kwargs)

# Update from db.csv_lig
>>> db.update_mongodb('ligand')

# Update from a lig_df, a user-provided DataFrame
>>> db.update_mongodb({'ligand': lig_df})
>>> print(type(lig_df))
<class 'pandas.core.frame.DataFrame'>
```

Parameters

- **database** (`str` or `Mapping[str, pandas.DataFrame]`) – The type of database. Accepted values are `"ligand"` and `"qd"`, opening *Database.csv_lig* and *Database.csv_qd*, respectivelly. Alternativelly, a dictionary with the database name and a matching DataFrame can be passed directly.
- **overwrite** (`bool`) – Whether or not previous entries can be overwritten or not.

Return type
`None`

update_csv(*df*, *index=None*, *database='ligand'*, *columns=None*, *overwrite=False*, *job_recipe=None*, *status=None*)

Update *Database.csv_lig* or *Database.csv_qd* with new settings.

Parameters

- **df** (`pandas.DataFrame`) – A dataframe of new (potential) database entries.
- **database** (`str`) – The type of database; accepted values are `"ligand"` (*Database.csv_lig*) and `"qd"` (*Database.csv_qd*).
- **columns** (`Sequence`, optional) – Optional: A sequence of column keys in **df** which (potentially) are to be added to this instance. If `None` Add all columns.
- **overwrite** (`bool`) – Whether or not previous entries can be overwritten or not.
- **status** (`str`, optional) – A descriptor of the status of the moleculair structures. Set to `"optimized"` to treat them as optimized geometries.

Return type
`None`

update_hdf5(*df*, *index*, *database='ligand'*, *overwrite=False*, *status=None*)

Export molecules (see the "mol" column in **df**) to the structure database.

Returns a series with the *Database.hdf5* indices of all new entries.

Parameters

- **df** (pandas.DataFrame) – A dataframe of new (potential) database entries.
- **database** (str) – The type of database; accepted values are "ligand" and "qd".
- **overwrite** (bool) – Whether or not previous entries can be overwritten or not.
- **status** (str, optional) – A descriptor of the status of the moleculair structures. Set to "optimized" to treat them as optimized geometries.

Returns

A series with the indices of all new molecules in *Database.hdf5*.

Return type

pandas.Series

from_csv(*df*, *database='ligand'*, *get_mol=True*, *inplace=True*)

Pull results from *Database.csv_lig* or *Database.csv_qd*.

Performs in inplace update of **df** if **inplace** = True, thus returing None.

Parameters

- **df** (pandas.DataFrame) – A dataframe of new (potential) database entries.
- **database** (str) – The type of database; accepted values are "ligand" and "qd".
- **get_mol** (bool) – Attempt to pull preexisting molecules from the database. See the **inplace** argument for more details.
- **inplace** (bool) – If True perform an inplace update of the "mol" column in **df**. Otherwise return a new series of PLAMS molecules.

Returns

Optional: A Series of PLAMS molecules if **get_mol** = True and **inplace** = False.

Return type

pandas.Series, optional

from_hdf5(*index*, *database='ligand'*, *rdmol=True*, *mol_list=None*)

Import structures from the hdf5 database as RDKit or PLAMS molecules.

Parameters

- **index** (Sequence[int] or slice) – The indices of the to be retrieved structures.
- **database** (str) – The type of database; accepted values are "ligand" and "qd".
- **rdmol** (bool) – If True, return an RDKit molecule instead of a PLAMS molecule.

Returns

A list of PLAMS or RDKit molecules.

Return type

List[plams.Molecule] or List[rdkit.Mol]

hdf5_availability(*timeout=5.0*, *max_attempts=10*)

Check if a .hdf5 file is opened by another process; return once it is not.

If two processes attempt to simultaneously open a single hdf5 file then h5py will raise an OSError.

The purpose of this method is ensure that a .hdf5 file is actually closed, thus allowing the *Database.from_hdf5()* method to safely access **filename** without the risk of raising an OSError.

Parameters

- **timeout** (float) – Time timeout, in seconds, between subsequent attempts of opening **filename**.
- **max_attempts** (int, optional) – Optional: The maximum number attempts for opening **filename**. If the maximum number of attempts is exceeded, raise an OSError. Setting this value to None will set the number of attempts to unlimited.

Raises

OSError – Raised if **max_attempts** is exceeded.

See also:

dataCAT.functions.hdf5_availability()

This method as a function.

class dataCAT.**DFProxy**(*ndframe*)

A mutable wrapper providing a view of the underlying dataframes.

ndframe

The embedded DataFrame.

Type

pandas.DataFrame

class dataCAT.**OpenLig**(*filename*, *write=True*)

Context manager for opening and closing the ligand database (*Database.csv_lig*).

property filename

Get the name of the to-be opened file.

Type

AnyStr

property write

Get whether or not *filename* should be written to when closing the context manager.

Type

bool

class dataCAT.**OpenQD**(*filename*, *write=True*)

Context manager for opening and closing the QD database (*Database.csv_qd*).

property filename

Get the name of the to-be opened file.

Type

AnyStr

property write

Get whether or not *filename* should be written to when closing the context manager.

Type
bool

2.8 The PDBContainer Class

A module for constructing array-representations of .pdb files.

2.8.1 Index

PDBContainer(atoms, bonds, atom_count, ...)	An (immutable) class for holding array-like representations of a set of .pdb files.
PDBContainer.atoms	Get a read-only padded recarray for keeping track of all atom-related information.
PDBContainer.bonds	Get a read-only padded recarray for keeping track of all bond-related information.
PDBContainer.atom_count	Get a read-only ndarray for keeping track of the number of atoms in each molecule in *atoms*.
PDBContainer.bond_count	Get a read-only ndarray for keeping track of the number of atoms in each molecule in *bonds*.
PDBContainer.scale	Get a recarray representing an index.

PDBContainer.__init__(atoms, bonds, ...[, ...])	Initialize an instance.
PDBContainer.__getitem__(index)	Implement `self[index]`.
PDBContainer.__len__()	Implement `len(self)`.
PDBContainer.keys()	Yield the (public) attribute names in this class.
PDBContainer.values()	Yield the (public) attributes in this instance.
PDBContainer.items()	Yield the (public) attribute name/value pairs in this instance.
PDBContainer.concatenate(*args)	Concatenate n PDBContainers into a single new instance.

PDBContainer.from_molecules(mol_list[, ...])	Convert an iterable or sequence of molecules into a new *PDBContainer* instance.
PDBContainer.to_molecules([index, mol])	Create a molecule or list of molecules from this instance.
PDBContainer.to_rdkit([index, sanitize])	Create an rdkit molecule or list of rdkit molecules from this instance.
PDBContainer.create_hdf5_group(file, name, *)	Create a h5py Group for storing *dataCAT.PDBContainer* instances.
PDBContainer.validate_hdf5(group)	Validate the passed hdf5 **group**, ensuring it is compatible with *PDBContainer* instances.
PDBContainer.from_hdf5(group[, index])	Construct a new PDBContainer from the passed hdf5 **group**.
PDBContainer.to_hdf5(group, index[, ...])	Update all datasets in **group** positioned at **index** with its counterpart from **pdb**.

PDBContainer.intersection(value)	Construct a new PDBContainer by the intersection of **self** and **value**.
PDBContainer.difference(value)	Construct a new PDBContainer by the difference of **self** and **value**.
PDBContainer.symmetric_difference(value)	Construct a new PDBContainer by the symmetric difference of **self** and **value**.
PDBContainer.union(value)	Construct a new PDBContainer by the union of **self** and **value**.

2.8.2 API

class dataCAT.**PDBContainer**(*atoms*, *bonds*, *atom_count*, *bond_count*, *scale=None*, *validate=True*, *copy=True*, *index_dtype=None*)

An (immutable) class for holding array-like representions of a set of .pdb files.

The *PDBContainer* class serves as an (intermediate) container for storing .pdb files in the hdf5 format, thus facilitating the storage and interconversion between PLAMS molecules and the `h5py` interface.

The methods implemented in this class can roughly be divided into three categories:

- Molecule-interconversion: *to_molecules()*, *from_molecules()* & *to_rdkit()*.
- hdf5-interconversion: *create_hdf5_group()*, *validate_hdf5()*, *to_hdf5()* & *from_hdf5()*.
- Miscellaneous: *keys()*, *values()*, *items()*, *__getitem__()* & *__len__()*.

Examples

```
>>> import h5py
>>> from scm.plams import readpdb
>>> from dataCAT import PDBContainer

>>> mol_list [readpdb(...), ...]
>>> pdb = PDBContainer.from_molecules(mol_list)
>>> print(pdb)
PDBContainer(
    atoms      = numpy.recarray(..., shape=(23, 76), dtype=...),
    bonds      = numpy.recarray(..., shape=(23, 75), dtype=...),
    atom_count = numpy.ndarray(..., shape=(23,), dtype=int32),
    bond_count = numpy.ndarray(..., shape=(23,), dtype=int32),
    scale      = numpy.recarray(..., shape=(23,), dtype=...)
)

>>> hdf5_file = str(...)
>>> with h5py.File(hdf5_file, 'a') as f:
...     group = pdb.create_hdf5_group(f, name='ligand')
...     pdb.to_hdf5(group, None)
...
...     print('group', '=', group)
...     for name, dset in group.items():
...         print(f'group[{name!r}]', '=', dset)
group = <HDF5 group "/ligand" (5 members)>
group['atoms'] = <HDF5 dataset "atoms": shape (23, 76), type "|V46">
```

(continues on next page)

(continued from previous page)

```
group['bonds'] = <HDF5 dataset "bonds": shape (23, 75), type "|V9">
group['atom_count'] = <HDF5 dataset "atom_count": shape (23,), type "<i4">
group['bond_count'] = <HDF5 dataset "bond_count": shape (23,), type "<i4">
group['index'] = <HDF5 dataset "index": shape (23,), type "<i4">
```

---

property atoms

Get a read-only padded recarray for keeping track of all atom-related information.

See *dataCAT.dtype.ATOMS_DTYPE* for a comprehensive overview of all field names and dtypes.

Type
numpy.recarray, shape (n, m)

property bonds

Get a read-only padded recarray for keeping track of all bond-related information.

Note that all atomic indices are 1-based.

See *dataCAT.dtype.BONDS_DTYPE* for a comprehensive overview of all field names and dtypes.

Type
numpy.recarray, shape (n, k)

property atom_count

Get a read-only ndarray for keeping track of the number of atoms in each molecule in *atoms*.

Type
numpy.ndarray[int32], shape $(n,)$

property bond_count

Get a read-only ndarray for keeping track of the number of atoms in each molecule in *bonds*.

Type
numpy.ndarray[int32], shape $(n,)$

property scale

Get a recarray representing an index.

Used as dimensional scale in the h5py Group.

Type
numpy.recarray, shape $(n,)$

__init__(*atoms*, *bonds*, *atom_count*, *bond_count*, *scale=None*, *validate=True*, *copy=True*, *index_dtype=None*)

Initialize an instance.

Parameters

- **atoms** (numpy.recarray, shape (n, m)) – A padded recarray for keeping track of all atom-related information. See *PDBContainer.atoms*.
- **bonds** (numpy.recarray, shape (n, k)) – A padded recarray for keeping track of all bond-related information. See *PDBContainer.bonds*.
- **atom_count** (numpy.ndarray[int32], shape $(n,)$) – An ndarray for keeping track of the number of atoms in each molecule in **atoms**. See *PDBContainer.atom_count*.
- **bond_count** (numpy.ndarray[int32], shape $(n,)$) – An ndarray for keeping track of the number of bonds in each molecule in **bonds**. See *PDBContainer.bond_count*.

- **scale** (numpy.recarray, shape $(n,)$, optional) – A recarray representing an index. If None, use a simple numerical index (*i.e.* numpy.arange()). See *PDBContainer.scale*.

Keyword Arguments

- **validate** (bool) – If True perform more thorough validation of the input arrays. Note that this also allows the parameters to-be passed as array-like objects in addition to aforementioned ndarray or recarray instances.
- **copy** (bool) – If True, set the passed arrays as copies. Only relevant if validate = True.

Return type
None

2.8.3 API: Miscellaneous Methods

PDBContainer.**__getitem__**(*index*)

Implement self[index].

Constructs a new *PDBContainer* instance by slicing all arrays with **index**. Follows the standard NumPy broadcasting rules: if an integer or slice is passed then a shallow copy is returned; otherwise a deep copy will be created.

Examples

```
>>> from dataCAT import PDBContainer

>>> pdb = PDBContainer(...)
>>> print(pdb)
PDBContainer(
    atoms      = numpy.recarray(..., shape=(23, 76), dtype=...),
    bonds      = numpy.recarray(..., shape=(23, 75), dtype=...),
    atom_count = numpy.ndarray(..., shape=(23,), dtype=int32),
    bond_count = numpy.ndarray(..., shape=(23,), dtype=int32),
    scale      = numpy.recarray(..., shape=(23,), dtype=...)
)

>>> pdb[0]
PDBContainer(
    atoms      = numpy.recarray(..., shape=(1, 76), dtype=...),
    bonds      = numpy.recarray(..., shape=(1, 75), dtype=...),
    atom_count = numpy.ndarray(..., shape=(1,), dtype=int32),
    bond_count = numpy.ndarray(..., shape=(1,), dtype=int32),
    scale      = numpy.recarray(..., shape=(1,), dtype=...)
)

>>> pdb[:10]
PDBContainer(
    atoms      = numpy.recarray(..., shape=(10, 76), dtype=...),
    bonds      = numpy.recarray(..., shape=(10, 75), dtype=...),
    atom_count = numpy.ndarray(..., shape=(10,), dtype=int32),
    bond_count = numpy.ndarray(..., shape=(10,), dtype=int32),
    scale      = numpy.recarray(..., shape=(10,), dtype=...)
```

(continues on next page)

(continued from previous page)

```
)

>>> pdb[[0, 5, 7, 9, 10]]
PDBContainer(
    atoms      = numpy.recarray(..., shape=(5, 76), dtype=...),
    bonds      = numpy.recarray(..., shape=(5, 75), dtype=...),
    atom_count = numpy.ndarray(..., shape=(5,), dtype=int32),
    bond_count = numpy.ndarray(..., shape=(5,), dtype=int32),
    scale      = numpy.recarray(..., shape=(5,), dtype=...)
)
```

Parameters
index (int, Sequence[int] or slice) – An object for slicing arrays along `axis=0`.

Returns
A shallow or deep copy of a slice of this instance.

Return type
dataCAT.PDBContainer

PDBContainer.**__len__**()

Implement `len(self)`.

Returns
Returns the length of the arrays embedded within this instance (which are all of the same length).

Return type
int

classmethod PDBContainer.**keys**()

Yield the (public) attribute names in this class.

Examples

```
>>> from dataCAT import PDBContainer

>>> for name in PDBContainer.keys():
...     print(name)
atoms
bonds
atom_count
bond_count
scale
```

Yields
str – The names of all attributes in this class.

PDBContainer.**values**()

Yield the (public) attributes in this instance.

Examples

```
>>> from dataCAT import PDBContainer

>>> pdb = PDBContainer(...)
>>> for value in pdb.values():
...     print(object.__repr__(value))
<numpy.recarray object at ...>
<numpy.recarray object at ...>
<numpy.ndarray object at ...>
<numpy.ndarray object at ...>
<numpy.recarray object at ...>
```

Yields

str – The values of all attributes in this instance.

PDBContainer.**items**()

Yield the (public) attribute name/value pairs in this instance.

Examples

```
>>> from dataCAT import PDBContainer

>>> pdb = PDBContainer(...)
>>> for name, value in pdb.items():
...     print(name, '=', object.__repr__(value))
atoms = <numpy.recarray object at ...>
bonds = <numpy.recarray object at ...>
atom_count = <numpy.ndarray object at ...>
bond_count = <numpy.ndarray object at ...>
scale = <numpy.recarray object at ...>
```

Yields

str and numpy.ndarray / numpy.recarray – The names and values of all attributes in this instance.

PDBContainer.**concatenate**(**args*)

Concatenate n PDBContainers into a single new instance.

Examples

```
>>> from dataCAT import PDBContainer

>>> pdb1 = PDBContainer(...)
>>> pdb2 = PDBContainer(...)
>>> pdb3 = PDBContainer(...)
>>> print(len(pdb1), len(pdb2), len(pdb3))
23 23 23

>>> pdb_new = pdb1.concatenate(pdb2, pdb3)
>>> print(pdb_new)
```

(continues on next page)

(continued from previous page)

```
PDBContainer(
    atoms      = numpy.recarray(..., shape=(69, 76), dtype=...),
    bonds      = numpy.recarray(..., shape=(69, 75), dtype=...),
    atom_count = numpy.ndarray(..., shape=(69,), dtype=int32),
    bond_count = numpy.ndarray(..., shape=(69,), dtype=int32),
    scale      = numpy.recarray(..., shape=(69,), dtype=...)
)
```

Parameters

***args** (*PDBContainer*) – One or more PDBContainers.

Returns

A new PDBContainer cosntructed by concatenating **self** and **args**.

Return type

PDBContainer

2.8.4 API: Object Interconversion

classmethod PDBContainer.**from_molecules**(*mol_list*, *min_atom=0*, *min_bond=0*, *scale=None*)

Convert an iterable or sequence of molecules into a new *PDBContainer* instance.

Examples

```
>>> from typing import List
>>> from dataCAT import PDBContainer
>>> from scm.plams import readpdb, Molecule

>>> mol_list: List[Molecule] = [readpdb(...), ...]
>>> PDBContainer.from_molecules(mol_list)
PDBContainer(
    atoms      = numpy.recarray(..., shape=(23, 76), dtype=...),
    bonds      = numpy.recarray(..., shape=(23, 75), dtype=...),
    atom_count = numpy.ndarray(..., shape=(23,), dtype=int32),
    bond_count = numpy.ndarray(..., shape=(23,), dtype=int32),
    scale      = numpy.recarray(..., shape=(23,), dtype=...)
)
```

Parameters

- **mol_list** (Iterable[Molecule]) – An iterable consisting of PLAMS molecules.
- **min_atom** (int) – The minimum number of atoms which *PDBContainer.atoms* should accomodate.
- **min_bond** (int) – The minimum number of bonds which *PDBContainer.bonds* should accomodate.
- **scale** (*array-like, optional*) – An array-like object representing an user-specified index. Defaults to a simple range index if None (see numpy.arange()).

Returns
A pdb container.

Return type
dataCAT.PDBContainer

PDBContainer.**to_molecules**(*index=None*, *mol=None*)

Create a molecule or list of molecules from this instance.

Examples

An example where one or more new molecules are created.

```
>>> from dataCAT import PDBContainer
>>> from scm.plams import Molecule

>>> pdb = PDBContainer(...)

# Create a single new molecule from `pdb`
>>> pdb.to_molecules(index=0)
<scm.plams.mol.molecule.Molecule object at ...>

# Create three new molecules from `pdb`
>>> pdb.to_molecules(index=[0, 1])
[<scm.plams.mol.molecule.Molecule object at ...>,
 <scm.plams.mol.molecule.Molecule object at ...>]
```

An example where one or more existing molecules are updated in-place.

```
# Update `mol` with the info from `pdb`
>>> mol = Molecule(...)  # doctest: +SKIP
>>> mol_new = pdb.to_molecules(index=2, mol=mol)
>>> mol is mol_new
True

# Update all molecules in `mol_list` with info from `pdb`
>>> mol_list = [Molecule(...), Molecule(...), Molecule(...)]  # doctest: +SKIP
>>> mol_list_new = pdb.to_molecules(index=range(3), mol=mol_list)
>>> for m, m_new in zip(mol_list, mol_list_new):
...     print(m is m_new)
True
True
True
```

Parameters

- **index** (`int`, `Sequence[int]` or `slice`, optional) – An object for slicing the arrays embedded within this instance. Follows the standard numpy broadcasting rules (*e.g.* `self.atoms[index]`). If a scalar is provided (*e.g.* an integer) then a single molecule will be returned. If a sequence, range, slice, *etc.* is provided then a list of molecules will be returned.
- **mol** (`Molecule` or `Iterable[Molecule]`, optional) – A molecule or list of molecules. If one or molecules are provided here then they will be updated in-place.

Returns
A molecule or list of molecules, depending on whether or not **index** is a scalar or sequence / slice. Note that if `mol is not None`, then the-be returned molecules won't be copies.

Return type
`Molecule` or `List[Molecule]`

`PDBContainer.`**`to_rdkit`**(*index=None*, *sanitize=True*)

Create an rdkit molecule or list of rdkit molecules from this instance.

Examples

An example where one or more new molecules are created.

```
>>> from dataCAT import PDBContainer
>>> from rdkit.Chem import Mol

>>> pdb = PDBContainer(...)

# Create a single new molecule from `pdb`
>>> pdb.to_rdkit(index=0)
<rdkit.Chem.rdchem.Mol object at ...>

# Create three new molecules from `pdb`
>>> pdb.to_rdkit(index=[0, 1])
[<rdkit.Chem.rdchem.Mol object at ...>,
 <rdkit.Chem.rdchem.Mol object at ...>]
```

Parameters

- **index** (`int`, `Sequence[int]` or `slice`, optional) – An object for slicing the arrays embedded within this instance. Follows the standard numpy broadcasting rules (*e.g.* `self.atoms[index]`). If a scalar is provided (*e.g.* an integer) then a single molecule will be returned. If a sequence, range, slice, *etc.* is provided then a list of molecules will be returned.
- **sanitize** (*bool*) – Whether to sanitize the molecule before returning or not.

Returns
A molecule or list of molecules, depending on whether or not **index** is a scalar or sequence / slice.

Return type
`Mol` or `list[Mol]`

`classmethod` `PDBContainer.`**`create_hdf5_group`**(*file*, *name*, ***, *scale=None*, *scale_dtype=None*, ***kwargs*)

Create a h5py Group for storing *dataCAT.PDBContainer* instances.

Notes

The **scale** and **scale_dtype** parameters are mutually exclusive.

Parameters

- **file** (h5py.File or h5py.Group) – The h5py File or Group where the new Group will be created.
- **name** (str) – The name of the to-be created Group.

Keyword Arguments

- **scale** (h5py.Dataset, keyword-only) – A pre-existing dataset serving as dimensional scale. See **scale_dtype** to create a new instead instead.
- **scale_dtype** (*dtype-like, keyword-only*) – The datatype of the to-be created dimensional scale. See **scale** to use a pre-existing dataset for this purpose.
- ****kwargs** (Any) – Further keyword arguments for the creation of each dataset. Arguments already specified by default are: `name`, `shape`, `maxshape` and `dtype`.

Returns
The newly created Group.

Return type
h5py.Group

classmethod PDBContainer.**validate_hdf5**(*group*)

Validate the passed hdf5 **group**, ensuring it is compatible with *PDBContainer* instances.

An AssertionError will be raise if **group** does not validate.

This method is called automatically when an exception is raised by *to_hdf5()* or *from_hdf5()*.

Parameters
group (h5py.Group) – The to-be validated hdf5 Group.

Raises
AssertionError – Raised if the validation process fails.

classmethod PDBContainer.**from_hdf5**(*group*, *index=None*)

Construct a new PDBContainer from the passed hdf5 **group**.

Parameters

- **group** (h5py.Group) – The to-be read h5py group.
- **index** (int, Sequence[int] or slice, optional) – An object for slicing all datasets in **group**.

Returns
A new PDBContainer constructed from **group**.

Return type
dataCAT.PDBContainer

PDBContainer.**to_hdf5**(*group*, *index*, *update_scale=True*)

Update all datasets in **group** positioned at **index** with its counterpart from **pdb**.

Follows the standard broadcasting rules as employed by h5py.

Important: If **index** is passed as a sequence of integers then, contrary to NumPy, they *will* have to be sorted.

Parameters

- **group** (h5py.Group) – The to-be updated h5py group.

- **index** (int, Sequence[int] or slice) – An object for slicing all datasets in **group**. Note that, contrary to numpy, if a sequence of integers is provided then they'll have to ordered.
- **update_scale** (bool) – If True, also export *PDBContainer.scale* to the dimensional scale in the passed **group**.

2.8.5 API: Set Operations

PDBContainer.**intersection**(*value*)

Construct a new PDBContainer by the intersection of **self** and **value**.

Examples

An example where one or more new molecules are created.

```
>>> from dataCAT import PDBContainer

>>> pdb = PDBContainer(...)
>>> print(pdb.scale)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22]

>>> pdb_new = pdb.intersection(range(4))
>>> print(pdb_new.scale)
[0 1 2 3]
```

Parameters

value (*PDBContainer* or array-like) – Another PDBContainer or an array-like object representing *PDBContainer.scale*. Note that both **value** and **self.scale** should consist of unique elements.

Returns

A new instance by intersecting *self.scale* and **value**.

Return type

PDBContainer

See also:

set.intersection

Return the intersection of two sets as a new set.

PDBContainer.**difference**(*value*)

Construct a new PDBContainer by the difference of **self** and **value**.

Examples

```
>>> from dataCAT import PDBContainer

>>> pdb = PDBContainer(...)
>>> print(pdb.scale)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22]
```

(continues on next page)

(continued from previous page)

```
>>> pdb_new = pdb.difference(range(10, 30))
>>> print(pdb_new.scale)
[0 1 2 3 4 5 6 7 8 9]
```

Parameters

value (*PDBContainer* or array-like) – Another PDBContainer or an array-like object representing *PDBContainer.scale*. Note that both **value** and **self.scale** should consist of unique elements.

Returns

A new instance as the difference of *self.scale* and **value**.

Return type

PDBContainer

See also:

set.difference

Return the difference of two or more sets as a new set.

PDBContainer.**symmetric_difference**(*value*)

Construct a new PDBContainer by the symmetric difference of **self** and **value**.

Examples

```
>>> from dataCAT import PDBContainer

>>> pdb = PDBContainer(...)
>>> pdb2 = PDBContainer(..., scale=range(10, 30))

>>> print(pdb.scale)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22]

>>> pdb_new = pdb.symmetric_difference(pdb2)
>>> print(pdb_new.scale)
[ 0  1  2  3  4  5  6  7  8  9 23 24 25 26 27 28 29]
```

Parameters

value (*PDBContainer*) – Another PDBContainer. Note that both **value.scale** and **self.scale** should consist of unique elements.

Returns

A new instance as the symmetric difference of *self.scale* and **value**.

Return type

PDBContainer

See also:

set.symmetric_difference

Return the symmetric difference of two sets as a new set.

PDBContainer.**union**(*value*)

Construct a new PDBContainer by the union of **self** and **value**.

Examples

```
>>> from dataCAT import PDBContainer

>>> pdb = PDBContainer(...)
>>> pdb2 = PDBContainer(..., scale=range(10, 30))

>>> print(pdb.scale)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22]

>>> pdb_new = pdb.union(pdb2)
>>> print(pdb_new.scale)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29]
```

Parameters

value (*PDBContainer*) – Another PDBContainer. Note that both **value** and **self.scale** should consist of unique elements.

Returns

A new instance as the union of `self.index` and **value**.

Return type

PDBContainer

See also:

`set.union`

Return the union of sets as a new set.

2.9 Data Types

A module with various data-types used throughout **Data-CAT**.

2.9.1 Index

ATOMS_DTYPE	The datatype of *PDBContainer.atoms*
BONDS_DTYPE	The datatype of *PDBContainer.bonds*
ATOM_COUNT_DTYPE	The datatype of *PDBContainer.atom_count*
BOND_COUNT_DTYPE	The datatype of *PDBContainer.bond_count*
LIG_IDX_DTYPE	The datatype of `PDBContainer.index` as used by the ligand database
QD_IDX_DTYPE	The datatype of `PDBContainer.index` as used by the QD database
BACKUP_IDX_DTYPE	The default datatype of `PDBContainer.index`
DT_DTYPE	The datatype of the `"date"` dataset created by *create_hdf5_log()*
VERSION_DTYPE	The datatype of the `"version"` dataset created by *create_hdf5_log()*
INDEX_DTYPE	The datatype of the `"index"` dataset created by *create_hdf5_log()*
MSG_DTYPE	The datatype of the `"message"` dataset created by *create_hdf5_log()*

2.9.2 API

dataCAT.dtype.**ATOMS_DTYPE : numpy.dtype = ...**

The datatype of *PDBContainer.atoms*

.

Most field names are based on to their, identically named, counterpart as produced by `readpdb()`, the data in question being stored in the `Atom.properties.pdb_info` block.

There are six exception to this general rule:

- `x`, `y` & `z`: Based on `Atom.x`, `Atom.y` and `Atom.z`.
- `symbol`: Based on `Atom.symbol`.
- `charge`: Based on `Atom.properties.charge`.
- `charge_float`: Based on `Atom.properties.charge_float`.

```
>>> from dataCAT.dtype import ATOMS_DTYPE

>>> print(repr(ATOMS_DTYPE))
dtype([('IsHeteroAtom', '?'),
       ('SerialNumber', '<i2'),
       ('Name', 'S4'),
       ('ResidueName', 'S3'),
       ('ChainId', 'S1'),
       ('ResidueNumber', '<i2'),
       ('x', '<f4'),
       ('y', '<f4'),
       ('z', '<f4'),
       ('Occupancy', '<f4'),
       ('TempFactor', '<f4'),
```

(continues on next page)

(continued from previous page)

```
       ('symbol', 'S4'),
       ('charge', 'i1'),
       ('charge_float', '<f8')])
```

dataCAT.dtype.**BONDS_DTYPE : numpy.dtype = ...**

The datatype of *PDBContainer.bonds*

.

Field names are based on to their, identically named, counterpart in `plams.Bond`.

```
>>> from dataCAT.dtype import BONDS_DTYPE

>>> print(repr(BONDS_DTYPE))
dtype([('atom1', '<i4'),
       ('atom2', '<i4'),
       ('order', 'i1')])
```

dataCAT.dtype.**ATOM_COUNT_DTYPE : numpy.dtype = ...**

The datatype of *PDBContainer.atom_count*

.

```
>>> from dataCAT.dtype import ATOM_COUNT_DTYPE

>>> print(repr(ATOM_COUNT_DTYPE))
dtype('int32')
```

dataCAT.dtype.**BOND_COUNT_DTYPE : numpy.dtype = ...**

The datatype of *PDBContainer.bond_count*

.

```
>>> from dataCAT.dtype import BOND_COUNT_DTYPE

>>> print(repr(BOND_COUNT_DTYPE))
dtype('int32')
```

dataCAT.dtype.**LIG_IDX_DTYPE : numpy.dtype = ...**

The datatype of `PDBContainer.index` as used by the ligand database

.

```
>>> import h5py
>>> from dataCAT.dtype import LIG_IDX_DTYPE

>>> print(repr(LIG_IDX_DTYPE))
dtype([('ligand', 'O'),
       ('ligand anchor', 'O')])

>>> h5py.check_string_dtype(LIG_IDX_DTYPE.fields['ligand'][0])
string_info(encoding='ascii', length=None)

>>> h5py.check_string_dtype(LIG_IDX_DTYPE.fields['ligand anchor'][0])
string_info(encoding='ascii', length=None)
```

dataCAT.dtype.**QD_IDX_DTYPE : numpy.dtype = ...**

The datatype of `PDBContainer.index` as used by the QD database

.

```
>>> import h5py
>>> from dataCAT.dtype import QD_IDX_DTYPE

>>> print(repr(QD_IDX_DTYPE))
dtype([('core', 'O'),
       ('core anchor', 'O'),
       ('ligand', 'O'),
       ('ligand anchor', 'O')])

>>> h5py.check_string_dtype(QD_IDX_DTYPE.fields['core'][0])
string_info(encoding='ascii', length=None)

>>> h5py.check_string_dtype(QD_IDX_DTYPE.fields['core anchor'][0])
string_info(encoding='ascii', length=None)

>>> h5py.check_string_dtype(QD_IDX_DTYPE.fields['ligand'][0])
string_info(encoding='ascii', length=None)

>>> h5py.check_string_dtype(QD_IDX_DTYPE.fields['ligand anchor'][0])
string_info(encoding='ascii', length=None)
```

dataCAT.dtype.**BACKUP_IDX_DTYPE : numpy.dtype = ...**

The default datatype of `PDBContainer.index`

.

```
>>> from dataCAT.dtype import BACKUP_IDX_DTYPE

>>> print(repr(BACKUP_IDX_DTYPE))
dtype('int32')
```

dataCAT.dtype.**DT_DTYPE : numpy.dtype = ...**

The datatype of the "`date`" dataset created by *create_hdf5_log()*

.

Field names are based on their, identically named, counterpart in the `datetime` class.

```
>>> from dataCAT.dtype import DT_DTYPE

>>> print(repr(DT_DTYPE))
dtype([('year', '<i2'),
       ('month', 'i1'),
       ('day', 'i1'),
       ('hour', 'i1'),
       ('minute', 'i1'),
       ('second', 'i1'),
       ('microsecond', '<i4')])
```

dataCAT.dtype.**VERSION_DTYPE : numpy.dtype = ...**

The datatype of the "`version`" dataset created by *create_hdf5_log()*

.

Field names are based on their, identically named, counterpart in the `nanoutils.VersionInfo` namedtuple.

```
>>> from dataCAT.dtype import VERSION_DTYPE

>>> print(repr(VERSION_DTYPE))
dtype([('major', 'i1'),
       ('minor', 'i1'),
       ('micro', 'i1')])
```

dataCAT.dtype.**INDEX_DTYPE : numpy.dtype = ...**

The datatype of the "index" dataset created by *create_hdf5_log()*

.

Used for representing a ragged array of 32-bit integers.

```
>>> import h5py
>>> from dataCAT.dtype import INDEX_DTYPE

>>> print(repr(INDEX_DTYPE))
dtype('O')

>>> h5py.check_vlen_dtype(INDEX_DTYPE)
dtype('int32')
```

dataCAT.dtype.**MSG_DTYPE : numpy.dtype = ...**

The datatype of the "message" dataset created by *create_hdf5_log()*

.

Used for representing variable-length ASCII strings.

```
>>> import h5py
>>> from dataCAT.dtype import MSG_DTYPE

>>> print(repr(MSG_DTYPE))
dtype('O')

>>> h5py.check_string_dtype(MSG_DTYPE)
string_info(encoding='ascii', length=None)
```

dataCAT.dtype.**FORMULA_DTYPE : numpy.dtype = ...**

The datatype of the "/ligand/properties/formula" dataset.

.

Used for representing variable-length ASCII strings.

```
>>> import h5py
>>> from dataCAT.dtype import FORMULA_DTYPE

>>> print(repr(FORMULA_DTYPE))
dtype('O')
```

(continues on next page)

(continued from previous page)

```
>>> h5py.check_string_dtype(FORMULA_DTYPE)
string_info(encoding='ascii', length=None)
```

dataCAT.dtype.**LIG_COUNT_DTYPE : numpy.dtype = ...**

The datatype of the "/qd/properties/ligand count" dataset.

.

```
>>> from dataCAT.dtype import LIG_COUNT_DTYPE

>>> print(repr(LIG_COUNT_DTYPE))
dtype('int32')
```

2.10 HDF5 Access Logging

A module related to logging and hdf5.

2.10.1 Index

create_hdf5_log(file[, n_entries, ...])	Create a hdf5 group for logging database modifications.
update_hdf5_log(group, index[, message, ...])	Add a new entry to the hdf5 logger in **file**.
reset_hdf5_log(group[, version_values])	Clear and reset the passed `logger` Group.
log_to_dataframe(group)	Export the log embedded within **file** to a Pandas DataFrame.

2.10.2 API

dataCAT.**create_hdf5_log**(*file*, *n_entries=100*, *clear_when_full=False*, *version_names=array([b'CAT', b'Nano-CAT', b'Data-CAT'], dtype='|S8')*, *version_values=array([(1, 1, 1), (0, 7, 2), (0, 7, 2)], dtype=[('major', 'i1'), ('minor', 'i1'), ('micro', 'i1')])*, ***kwargs*)

Create a hdf5 group for logging database modifications.

The logger Group consists of four main datasets:

- "`date`": Denotes dates and times for when the database is modified.
- "`version`": Denotes user-specified package versions for when the database is modified.
- "`version_names`" : See the **version_names** parameter.
- "`message`": Holds user-specified modification messages.
- "`index`": Denotes indices of which elements in the database were modified.

Examples

```
>>> import h5py
>>> from dataCAT import create_hdf5_log
```

(continues on next page)

(continued from previous page)

```
>>> hdf5_file = str(...)
>>> with h5py.File(hdf5_file, 'a') as f:
...     group = create_hdf5_log(f)
...
...     print('group', '=', group)
...     for name, dset in group.items():
...         print(f'group[{name!r}]', '=', dset)
group = <HDF5 group "/logger" (5 members)>
group['date'] = <HDF5 dataset "date": shape (100,), type "|V11">
group['version'] = <HDF5 dataset "version": shape (100, 3), type "|V3">
group['version_names'] = <HDF5 dataset "version_names": shape (3,), type "|S8">
group['message'] = <HDF5 dataset "message": shape (100,), type "|O">
group['index'] = <HDF5 dataset "index": shape (100,), type "|O">
```

Parameters

- **file** (h5py.File or h5py.Group) – The File or Group where the logger should be created.
- **n_entries** (int) – The initial number of entries in each to-be created dataset. In addition, everytime the datasets run out of available slots their length will be increased by this number (assuming clear_when_full = False).
- **clear_when_full** (bool) – If True, delete the logger and create a new one whenever it is full. Increase the size of each dataset by **n_entries** otherwise.
- **version_names** (Sequence[str or bytes]) – A sequence consisting of strings and/or bytes representing the names of the to-be stored package versions. Should be of the same length as **version_values**.
- **version_values** (Sequence[Tuple[int, int, int]]) – A sequence with 3-tuples, each tuple representing a package version associated with its respective counterpart in **version_names**.
- ****kwargs** (Any) – Further keyword arguments for the h5py create_dataset() function.

Returns

The newly created "logger" group.

Return type

h5py.Group

dataCAT.**update_hdf5_log**(*group*, *index*, *message=None*, *version_values=array([(1, 1, 1), (0, 7, 2), (0, 7, 2)], dtype=[('major', 'i1'), ('minor', 'i1'), ('micro', 'i1')])*)

Add a new entry to the hdf5 logger in **file**.

Examples

```
>>> from datetime import datetime

>>> import h5py
>>> from dataCAT import update_hdf5_log

>>> hdf5_file = str(...)
```

(continues on next page)

(continued from previous page)

```
>>> with h5py.File(hdf5_file, 'r+') as f:
...     group = f['ligand/logger']
...
...     n = group.attrs['n']
...     date_before = group['date'][n]
...     index_before = group['index'][n]
...
...     update_hdf5_log(group, index=[0, 1, 2, 3], message='append')
...     date_after = group['date'][n]
...     index_after = group['index'][n]

>>> print(index_before, index_after, sep='\n')
[]
[0 1 2 3]

>>> print(date_before, date_after, sep='\n')
(0, 0, 0, 0, 0, 0, 0)
(2020, 6, 24, 16, 33, 7, 959888)
```

Parameters

- **group** (h5py.Group) – The `logger` Group.
- **idx** (numpy.ndarray) – A numpy array with the indices of (to-be logged) updated elements.
- **version_values** (Sequence[Tuple[int, int, int]]) – A sequence with 3-tuples representing to-be updated package versions.

Return type

None

dataCAT.**reset_hdf5_log**(*group*, *version_values=array([(1, 1, 1), (0, 7, 2), (0, 7, 2)], dtype=[('major', 'i1'), ('minor', 'i1'), ('micro', 'i1')])*)

Clear and reset the passed `logger` Group.

Examples

```
>>> import h5py
>>> from dataCAT import reset_hdf5_log

>>> hdf5_file = str(...)

>>> with h5py.File(hdf5_file, 'r+') as f:
...     group = f['ligand/logger']
...     print('before:')
...     print(group.attrs['n'])
...
...     group = reset_hdf5_log(group)
...     print('\nafter:')
...     print(group.attrs['n'])
before:
2
```

(continues on next page)

(continued from previous page)

```
after:
0
```

Parameters

- **group** (h5py.File or h5py.Group) – The logger Group.
- **version_values** (Sequence[Tuple[int, int, int]]) – A sequence with 3-tuples representing to-be updated package versions.

Returns
The newly (re-)created "logger" group.

Return type
h5py.Group

dataCAT.**log_to_dataframe**(*group*)

Export the log embedded within **file** to a Pandas DataFrame.

Examples

```
>>> import h5py
>>> from dataCAT import log_to_dataframe

>>> hdf5_file = str(...)

>>> with h5py.File(hdf5_file, 'r') as f:
...     group = f['ligand/logger']
...     df = log_to_dataframe(group)
...     print(df)
                            CAT              ... Data-CAT message              ␣
↪index
                          major minor micro  ...    micro
date                                         ...
2020-06-24 15:28:09.861074     0     9     6  ...        1  update                ␣
↪[0]
2020-06-24 15:56:18.971201     0     9     6  ...        1  append  [1, 2, 3, 4, 5,␣
↪6]

[2 rows x 11 columns]
```

Parameters
group (h5py.Group) – The logger Group.

Returns
A DataFrame containing the content of file["logger"].

Return type
pandas.DataFrame

2.11 HDF5 Property Storage

A module for storing quantum mechanical properties in hdf5 format.

2.11.1 Index

create_prop_group(file, scale)	Create a group for holding user-specified properties.
create_prop_dset(group, name[, dtype, ...])	Construct a new dataset for holding a user-defined molecular property.
update_prop_dset(dset, data[, index])	Update **dset** at position **index** with **data**.
validate_prop_group(group)	Validate the passed hdf5 **group**, ensuring it is compatible with *create_prop_group()* and *create_prop_group()*.
index_to_pandas(dset[, fields])	Construct an MultiIndex from the passed `index` dataset.
prop_to_dataframe(dset[, dtype])	Convert the passed property Dataset into a DataFrame.

2.11.2 API

dataCAT.**create_prop_group**(*file*, *scale*)

Create a group for holding user-specified properties.

```
>>> import h5py
>>> from dataCAT import create_prop_group

>>> hdf5_file = str(...)
>>> with h5py.File(hdf5_file, 'r+') as f:
...     scale = f.create_dataset('index', data=np.arange(10))
...     scale.make_scale('index')
...
...     group = create_prop_group(f, scale=scale)
...     print('group', '=', group)
group = <HDF5 group "/properties" (0 members)>
```

Parameters

- **file** (h5py.File or h5py.Group) – The File or Group where the new "properties" group should be created.
- **scale** (h5py.DataSet) – The dimensional scale which will be attached to all property datasets created by *dataCAT.create_prop_dset()*.

Returns

The newly created group.

Return type

h5py.Group

dataCAT.**create_prop_dset**(*group*, *name*, *dtype=None*, *prop_names=None*, ***kwargs*)

Construct a new dataset for holding a user-defined molecular property.

Examples

In the example below a new dataset is created for storing solvation energies in water, methanol and ethanol.

```
>>> import h5py
>>> from dataCAT import create_prop_dset

>>> hdf5_file = str(...)

>>> with h5py.File(hdf5_file, 'r+') as f:
...     group = f['properties']
...     prop_names = ['water', 'methanol', 'ethanol']
...
...     dset = create_prop_dset(group, 'E_solv', prop_names=prop_names)
...     dset_names = group['E_solv_names']
...
...     print('group', '=', group)
...     print('group["E_solv"]', '=', dset)
...     print('group["E_solv_names"]', '=', dset_names)
group = <HDF5 group "/properties" (2 members)>
group["E_solv"] = <HDF5 dataset "E_solv": shape (10, 3), type "<f4">
group["E_solv_names"] = <HDF5 dataset "E_solv_names": shape (3,), type "|S8">
```

Parameters

- **group** (h5py.Group) – The "properties" group where the new dataset will be created.
- **name** (str) – The name of the new dataset.
- **prop_names** (Sequence[str], optional) – The names of each row in the to-be created dataset. Used for defining the length of the second axis and will be used as a dimensional scale for aforementioned axis. If None, create a 1D dataset (with no columns) instead.
- **dtype** (*dtype-like*) – The data type of the to-be created dataset.
- ****kwargs** (Any) – Further keyword arguments for the h5py create_dataset() method.

Returns

The newly created dataset.

Return type

h5py.Dataset

dataCAT.**update_prop_dset**(*dset*, *data*, *index=None*)

Update **dset** at position **index** with **data**.

Parameters

- **dset** (h5py.Dataset) – The to-be updated h5py dataset.
- **data** (numpy.ndarray) – An array containing the to-be added data.
- **index** (slice or numpy.ndarray, optional) – The indices of all to-be updated elements in **dset**. **index** either should be of the same length as **data**.

Return type

None

dataCAT.**validate_prop_group**(*group*)

Validate the passed hdf5 **group**, ensuring it is compatible with *create_prop_group()* and *create_prop_group()*.

This method is called automatically when an exception is raised by *update_prop_dset()*.

Parameters
group (h5py.Group) – The to-be validated hdf5 Group.

Raises
AssertionError – Raised if the validation process fails.

dataCAT.**index_to_pandas**(*dset*, *fields=None*)

Construct an MultiIndex from the passed `index` dataset.

Examples

```
>>> from dataCAT import index_to_pandas
>>> import h5py

>>> filename = str(...)

# Convert the entire dataset
>>> with h5py.File(filename, "r") as f:
...     dset: h5py.Dataset = f["ligand"]["index"]
...     index_to_pandas(dset)
MultiIndex([('O=C=O', 'O1'),
            ('O=C=O', 'O3'),
            ( 'CCCO', 'O4')],
           names=['ligand', 'ligand anchor'])

# Convert a subset of fields
>>> with h5py.File(filename, "r") as f:
...     dset = f["ligand"]["index"]
...     index_to_pandas(dset, fields=["ligand"])
MultiIndex([('O=C=O',),
            ('O=C=O',),
            ( 'CCCO',)],
           names=['ligand'])
```

Parameters

- **dset** (h5py.Dataset) – The relevant `index` dataset.
- **fields** (Sequence[str]) – The names of the `index` fields that are to-be included in the returned MultiIndex. If None, include all fields.

Returns
A multi-index constructed from the passed dataset.

Return type
pandas.MultiIndex

dataCAT.**prop_to_dataframe**(*dset*, *dtype=None*)

Convert the passed property Dataset into a DataFrame.

Examples

```
>>> import h5py
>>> from dataCAT import prop_to_dataframe

>>> hdf5_file = str(...)

>>> with h5py.File(hdf5_file, 'r') as f:
...     dset = f['ligand/properties/E_solv']
...     df = prop_to_dataframe(dset)
...     print(df)
E_solv_names             water  methanol   ethanol
ligand ligand anchor
O=C=O  O1            -0.918837 -0.151129 -0.177396
       O3            -0.221182 -0.261591 -0.712906
CCCO   O4            -0.314799 -0.784353 -0.190898
```

Parameters

- **dset** (h5py.Dataset) – The property-containing Dataset of interest.
- **dtype** (*dtype-like, optional*) – The data type of the to-be returned DataFrame. Use None to default to the data type of **dset**.

Returns
A DataFrame constructed from the passed **dset**.

Return type
pandas.DataFrame

2.12 Context Managers

Various context managers for manipulating molecules.

2.12.1 Index

as_array.AsArray(mol)	A context manager for temporary interconverting between PLAMS molecules and NumPy arrays.
mol_split_cm.SplitMol(mol, bond_list[, cap_type])	A context manager for temporary splitting a single molecule into multiple components.
remove_atoms_cm.RemoveAtoms(mol, atoms)	A context manager for temporary removing a set of atoms from a molecule.

2.12.2 API

class CAT.attachment.as_array.**AsArray**(*mol*)

A context manager for temporary interconverting between PLAMS molecules and NumPy arrays.

Examples

```
>>> from scm.plams import Molecule

# Create a H2 example molecule
>>> h1 = Atom(symbol='H', coords=(0.0, 0.0, 0.0))
>>> h2 = Atom(symbol='H', coords=(1.0, 0.0, 0.0))
>>> mol = Molecule()
>>> mol.add_atom(h1)
>>> mol.add_atom(h2)

>>> print(mol)
  Atoms:
    1         H      0.000000      0.000000      0.000000
    2         H      1.000000      0.000000      0.000000


# Example: Translate the molecule along the Cartesian Z-axis by 5 Angstroem
>>> with AsArray(mol) as xyz:
...     xyz[:, 2] += 5

>>> print(mol)
  Atoms:
    1         H      0.000000      0.000000      5.000000
    2         H      1.000000      0.000000      5.000000
```

Parameters

mol (plams.Molecule or Iterable [plams.Atom]) – An iterable consisting of PLAMS atoms. See *AsArray.mol*.

mol

A PLAMS molecule or a sequence of PLAMS atoms.

Type

plams.Molecule or Sequence [plams.Atom]

_xyz

A 2D array with the Cartesian coordinates of **mol**. Empty by default; this value is set internally by the AsArray.__enter__() method.

Type

$n * 3$ numpy.ndarray [float], optional

class CAT.attachment.mol_split_cm.**SplitMol**(*mol*, *bond_list*, *cap_type='H'*)

A context manager for temporary splitting a single molecule into multiple components.

The context manager splits the provided molecule into multiple components, capping all broken bonds in the process. The exact amount of fragments depends on the number of specified bonds.

These moleculair fragments are returned upon opening the context manager and merged back into the initial molecule once the context manager is closed. While opened, the initial molecule is cleared of all atoms and bonds, while the same hapens to the moleculair fragments upon closing.

Examples

```
>>> from scm.plams import Molecule, Bond, from_smiles

>>> mol: Molecule = from_smiles('CC')  # Ethane
>>> bond: Bond = mol[1, 2]

# A backup of all bonds and atoms
>>> bonds_backup = mol.bonds.copy()
>>> atoms_backup = mol.atoms.copy()

# The context manager is opened; the bond is removed and the molecule is fragmented
>>> with SplitMol(mol, bond) as fragment_tuple:
...     for fragment in fragment_tuple:
...         fancy_operation(fragment)
...
...     print(
...         mol.bonds == bonds_backup,
...         mol.atoms == atoms_backup,
...         bond in mol.bonds
...     )
False False False

# The context manager is closed; all atoms and bonds have been restored
>>> print(
...     mol.bonds == bonds_backup,
...     mol.atoms == atoms_backup,
...     bond in mol.bonds
... )
True True True
```

Parameters

- **mol** (plams.Molecule) – A PLAMS molecule. See *SplitMol.mol*.
- **bond_list** (plams.Bond or Iterable [plams.Bond]) – An iterable consisting of PLAMS bonds. All bonds must be part of **mol**. See *SplitMol.bonds*.
- **cap_type** (str, int or plams.Atom) – An atomic number or symbol of the atom type used for capping the to-be split molecule. See *SplitMol.cap_type*.

mol

A PLAMS molecule.

Type
plams.Molecule

bonds

A set of PLAMS bonds.

Type
set [plams.Bond]

cap_type

An atomic symbol of the atom type used for capping the to-be split molecule.

Type
str

_at_pairs

A list of dictionaries. Each dictionary contains two atoms as keys (see `SplitMol.bond_list`) and their respective capping atom as values. Used for reassembling *SplitMol.mol* once the context manager is closed. Set internally by `SplitMol.__enter__()`.

Type
list [dict [plams.Atom, plams.Atom]], optional

_vars_backup

A backup of all instance variables of *SplitMol.mol*. Set internally by `SplitMol.__enter__()`.

Type
dict [str, Any], optional

_tmp_mol_list

A list of PLAMS molecules obtained by splitting *SplitMol.mol*. Set internally by `SplitMol.__enter__()`.

Type
tuple [plams.Molecule], optional

Raises
MoleculeError – Raised when one attempts to access or manipulate the instance variables of *SplitMol.mol* when the context manager is opened.

class CAT.attachment.remove_atoms_cm.**RemoveAtoms**(*mol*, *atoms*)

A context manager for temporary removing a set of atoms from a molecule.

The *relative* ordering of the to-be removed atoms (and matching bonds), as specified in **atoms**, is preserved during the removal and reattachment process. Note that reattaching will (re-)append the removed atoms/bonds, a process which is thus likelly to affect the *absolute* ordering of atoms/bonds within the entire molecule.

Examples

```
>>> from scm.plams import Molecule, Atom, from_smiles

>>> mol: Molecule = from_smiles('CO')
>>> atom1: Atom = mol[1]
>>> atom2: Atom = mol[2]

>>> atom_set = {atom1, atom2}
>>> with RemoveAtoms(mol, atom_set):
...     print(atom1 in mol, atom2 in mol)
False False

>>> print(atom1 in mol, atom2 in mol)
True True
```

Parameters

- **mol** (plams.Molecule) – A PLAMS molecule. See *RemoveAtoms.mol*.
- **atoms** (plams.Atom or Iterable [plams.Atom]) – A PLAMS atom or an iterable consisting of unique PLAMS atoms. All supplied atoms should belong to **mol**. See *RemoveAtoms.atoms*.

mol

A PLAMS molecule.

Type
plams.Molecule

atoms

A sequence of PLAMS atoms belonging to *RemoveAtoms.mol*. Setting a value will convert it into a sequence of atoms.

Type
Sequence [plams.Atom]

_bonds

A ordered dictionary of PLAMS bonds connected to one or more atoms in *RemoveAtoms.atoms*. All values are None, the dictionary serving as an improvised OrderedSet. Set to None until RemoveAtoms.__enter__() is called.

Type
OrderedDict [plams.Bond, None]

2.13 Ensemble-Averaged Activation Strain Analysis

$$\Delta\overline{E} = \Delta\overline{E}_{\text{strain}} + \Delta\overline{E}_{\text{int}} \tag{2.6}$$

Herein we describe an Ensemble-Averaged extension of the activation/strain analysis (ASA; also known as the distortion/interaction model), wherein the ASA is utilized for the analyses of entire molecular dynamics trajectories. The implementation utilizes CHARMM-style forcefields for the calculation of all energy terms.

Note: Throughout this document an overline will be used to distinguish between "normal" and ensemble-averaged quantities: *e.g.* E_{strain} versus $\overline{E}_{\text{strain}}$.

2.13.1 Strain/Distortion

The ensemble averaged strain $\Delta\overline{E}_{\text{strain}}$ represents the distortion of all ligands with respect to their equilibrium geometry. Given an MD trajectory with m iterations and n ligands per quantum dot, the energy is averaged over all m MD iterations and summed over all n ligands.

The magnitude of this term is determined by all covalent and non-covalent intra-ligand interactions. As this term quantifies the deviation of a ligand from its equilibrium geometry, it is, by definition, always positive.

$$\Delta E_{\text{strain}} = E_{\text{lig-pert}} - E_{\text{lig-eq}} \quad \Rightarrow \quad \Delta\overline{E}_{\text{strain}} = \frac{1}{m}\sum_{i=0}^{m}\sum_{j=0}^{n} E_{\text{lig-pert}}(i,j) - E_{\text{lig-eq}} \tag{2.7}$$

$$\Delta E_{\text{strain}} = \Delta V_{\text{bond}} + \Delta V_{\text{angle}} + \Delta V_{\text{Urey-Bradley}} + \Delta V_{\text{dihedral}} + \Delta V_{\text{improper}} + \Delta V_{\text{Lennard-Jones}} + \Delta V_{\text{elstat}} \tag{2.8}$$

$E_{\text{lig-eq}}$ is herein the total energy of a (single) ligand at its equilibrium geometry, while $E_{\text{lig-pert}}(i,j)$ is the total energy of the (perturbed) ligand j at MD iteration i.

2.13.2 Interaction

The ensemble averaged interaction $\Delta\overline{E}_{\text{int}}$ represents the mutual interaction between all ligands in a molecule. The interaction is, again, averaged over all MD iterations and summed over all ligand-pairs.

The magnitude of this term is determined by all non-covalent inter-ligand interactions and can be either positive (dominated by Pauli and/or Coulombic repulsion) or negative (dominated by dispersion and/or Coulombic attraction).

$$\Delta E_{\text{int}} = \sum_{j=0}^{n}\sum_{kj}^{n} \Delta E_{\text{lig-int}}(j,k) \quad \Rightarrow \quad \Delta\overline{E}_{\text{int}} = \frac{1}{m}\sum_{i=0}^{m}\sum_{j=0}^{n}\sum_{kj}^{n} \Delta E_{\text{lig-int}}(i,j,k) \tag{2.9}$$

$$\Delta E_{\text{int}} = \Delta V_{\text{Lennard-Jones}} + \Delta V_{\text{elstat}} \tag{2.10}$$

$\Delta E_{\text{lig-int}}(i,j,k)$ represents the pair-wise interactions between ligands j and k at MD iteration i. Double counting is avoided by ensuring that $k > j$.

Note: In order to avoid the substantial Coulombic repulsion between negatively charged ligands, its parameters are substituted with those from its neutral (*i.e.* protonated) counterpart. This correction is applied, exclusively, for the calculation of $\Delta E_{\text{lig-int}}$.

2.13.3 Total Energy

The total (ensemble-averaged) energy is the sum of $\Delta\overline{E}_{\text{strain}}$ and $\Delta\overline{E}_{\text{int}}$. Note that the energy is associated with a set of n ligands, *i.e.* the distortion and mutual interaction between all n ligands. Division by n will thus yield the averaged energy per ligand per MD iteration.

$$\Delta\overline{E} = \Delta\overline{E}_{\text{strain}} + \Delta\overline{E}_{\text{int}} = \frac{1}{m}\sum_{i=0}^{m} \Delta E_{\text{strain}}(i) + \Delta E_{\text{int}}(i) \tag{2.11}$$

2.13.4 Examples

An example input script using the `Cd68Se55` core and `OC(=O)CC` ligand.

The *activation_strain.md* key enables the MD-ASA procedure; *activation_strain.use_ff* ensures that the user-specified forcefield is used during the construction of the MD trajectory.

```
path: ...

input_cores:
    - Cd68Se55.xyz:
        guess_bonds: False

input_ligands:
    - OC(=O)CC

optional:
    core:
        dummy: Cl

    ligand:
        optimize: True
        split: True

    qd:
        activation_strain:
            use_ff: True
            md: True
            job1: Cp2kJob

    forcefield:
        charge:
            keys: [input, force_eval, mm, forcefield, charge]
            Cd: 0.9768
            Se: -0.9768
            O2D2: -0.4704
            C2O3: 0.4524
        epsilon:
            unit: kjmol
            keys: [input, force_eval, mm, forcefield, nonbonded, lennard-jones]
            Cd Cd: 0.3101
            Se Se: 0.4266
            Cd Se: 1.5225
            Cd O2D2: 1.8340
            Se O2D2: 1.6135
        sigma:
            unit: nm
            keys: [input, force_eval, mm, forcefield, nonbonded, lennard-jones]
            Cd Cd: 0.1234
            Se Se: 0.4852
            Cd Se: 0.2940
            Cd O2D2: 0.2471
            Se O2D2: 0.3526
```

2.13.5 activation_strain

optional.qd.**activation_strain**

All settings related to the activation strain analyses.

Example:

```
optional:
    qd:
        activation_strain:
            use_ff: True
            md: True
            iter_start: 500
            dump_csv: False

            el_scale14: 1.0
            lj_scale14: 1.0

            distance_upper_bound: "inf"
            k: 20
            shift_cutoff: True

            job1: cp2kjob
            s1: ...


    forcefield:
        ...
```

optional.qd.activation_strain.**use_ff**

Parameter

- **Type** - bool
- **Default value** – False

Utilize the parameters supplied in the optional.forcefield block.

optional.qd.activation_strain.**md**

Parameter

- **Type** - bool
- **Default value** – False

Perform an ensemble-averaged activation strain analysis.

If True, perform the analysis along an entire molecular dynamics trajectory. If False, only use a single geometry instead.

optional.qd.activation_strain.**iter_start**

Parameter

- **Type** - int
- **Default value** – 500

The MD iteration at which the ASA will be started.

All preceding iteration are disgarded, treated as pre-equilibration steps. Note that this refers to the iteration is specified in the .xyz file. For example, if a geometry is written to the .xyz file very 10 iterations (as is the default), then `iter_start=500` is equivalent to MD iteration 5000.

optional.qd.activation_strain.**dump_csv**

Parameter

- **Type** - bool
- **Default value** – False

Dump a set of .csv files containing all potential energies gathered over the course of the MD simulation.

For each quantum dot two files are created in the `.../qd/asa/` directory, one containing the potentials over the course of the MD simulation (`.qd.csv`) and for the optimized ligand (`.lig.csv`).

optional.qd.activation_strain.**el_scale14**

Parameter

- **Type** - float
- **Default value** – 1.0

Scaling factor to apply to all 1,4-nonbonded electrostatic interactions.

Serves the same purpose as the cp2k EI_SCALE14 keyword.

optional.qd.activation_strain.**lj_scale14**

Parameter

- **Type** - float
- **Default value** – 1.0

Scaling factor to apply to all 1,4-nonbonded Lennard-Jones interactions.

Serves the same purpose as the cp2k VDW_SCALE14 keyword.

optional.qd.activation_strain.**distance_upper_bound**

Parameter

- **Type** - float or str
- **Default value** – "inf"

Consider only atom-pairs within this distance for calculating inter-ligand interactions.

Units are in Angstrom. Using `"inf"` will default to the full, untruncated, distance matrix.

optional.qd.activation_strain.**k**

Parameter

- **Type** - int
- **Default value** – 20

The (maximum) number of to-be considered distances per atom.

Only relevant when *distance_upper_bound != "inf"*.

optional.qd.activation_strain.**shift_cutoff**

Parameter

- **Type** - bool
- **Default value** – True

Add a constant to all electrostatic and Lennard-Jones potentials such that the potential is zero at the *distance upper bound*.

Serves the same purpose as the cp2k SHIFT_CUTOFF keyword. Only relevant when *distance_upper_bound != "inf"*.

optional.qd.activation_strain.**job1**

Parameter

- **Type** - type or str
- **Default value** – Cp2kJob

A type object of a Job subclass, used for performing the activation strain analysis.

Should be set to Cp2kJob if *activation_strain.md = True*.

optional.qd.activation_strain.**s1**

Parameter

- **Type** - dict, str or bool
- **Default value** – See below

```yaml
s1:
    input:
        motion:
            print:
                trajectory:
                    each:
                        md: 10
            md:
                ensemble: NVT
                temperature: 300.0
                timestep: 1.0
                steps: 15000
                thermostat:
                    type: CSVR
                    csvr:
                        timecon: 1250
```

(continues on next page)

(continued from previous page)

```
        force_eval:
            method: FIST
            mm:
                forcefield:
                    ei_scale14: 1.0
                    vdw_scale14: 1.0
                    ignore_missing_critical_params: ''
                    parmtype: CHM
                    parm_file_name: null
                    do_nonbonded: ''
                    shift_cutoff: .TRUE.
                    spline:
                        emax_spline: 10e10
                        r0_nb: 0.2
                poisson:
                    periodic: NONE
                    ewald:
                        ewald_type: NONE
            subsys:
                cell:
                    abc: '[angstrom] 100.0 100.0 100.0'
                    periodic: NONE
                topology:
                    conn_file_format: PSF
                    conn_file_name: null
                    coord_file_format: 'OFF'
                    center_coordinates:
                        center_point: 0.0 0.0 0.0

        global:
            print_level: low
            project: cp2k
            run_type: MD
```

The job settings used for calculating the performing the ASA.

Alternatively, a path can be provided to .json or .yaml file containing the job settings.

The default settings above are specifically for the ensemble-averaged ASA (*activation_strain.md = True*.).

2.14 Recipes

2.14.1 nanoCAT.recipes.mark_surface

A recipe for identifying surface-atom subsets.

Index

replace_surface(mol, symbol[, symbol_new, ...])	A workflow for identifying all surface atoms in **mol** and replacing a subset of them.

API

nanoCAT.recipes.**replace_surface**(*mol*, *symbol*, *symbol_new='Cl'*, *nth_shell=0*, *f=0.5*, *mode='uniform'*, *displacement_factor=0.5*, ***kwargs*)

A workflow for identifying all surface atoms in **mol** and replacing a subset of them.

Consists of three distinct steps:

1. Identifying which atoms, with a user-specified atomic **symbol**, are located on the surface of **mol** rather than in the bulk.
2. Define a subset of the newly identified surface atoms using one of CAT's distribution algorithms.
3. Create and return a molecule where the atom subset defined in step 2 has its atomic symbols replaced with **symbol_new**.

Examples

Replace 75% of all surface `"Cl"` atoms with `"I"`.

```
>>> from scm.plams import Molecule
>>> from CAT.recipes import replace_surface

>>> mol = Molecule(...)  # Read an .xyz file
>>> mol_new = replace_surface(mol, symbol='Cl', symbol_new='I', f=0.75)
>>> mol_new.write(...)  # Write an .xyz file
```

The same as above, except this time the new `"I"` atoms are all deleted.

```
>>> from scm.plams import Molecule
>>> from CAT.recipes import replace_surface

>>> mol = Molecule(...)  # Read an .xyz file
>>> mol_new = replace_surface(mol, symbol='Cl', symbol_new='I', f=0.75)

>>> del_atom = [at for at in mol_new if at.symbol == 'I']
>>> for at in del_atom:
...     mol_new.delete_atom(at)
>>> mol_new.write(...)  # Write an .xyz file
```

Parameters

- **mol** (`Molecule`) – The input molecule.
- **symbol** (`str` or `int`) – An atomic symbol or number defining the super-set of the surface atoms.
- **symbol_new** (`str` or `int`) – An atomic symbol or number which will be assigned to the new surface-atom subset.

- **nth_shell** (int or Iterable [int]) – One or more integers denoting along which shell-surface(s) to search. For example, if `symbol = "Cd"` then `nth_shell = 0` represents the surface, `nth_shell = 1` is the first sub-surface `"Cd"` shell and `nth_shell = 2` is the second sub-surface `"Cd"` shell. Using `nth_shell = [1, 2]` will search along both the first and second `"Cd"` sub-surface shells. Note that a `Zscm.plams.core.errors.MoleculeError` will be raised if the specified **nth_shell** is larger than the actual number of available sub-surface shells.
- **f** (float) – The fraction of surface atoms whose atom types will be replaced with **symbol_new**. Must obey the following condition: $0 < f \leq 1$.
- **mode** (str) – How the subset of surface atoms will be generated. Accepts one of the following values:
  - `"random"`: A random distribution.
  - `"uniform"`: A uniform distribution; maximizes the nearest-neighbor distance.
  - `"cluster"`: A clustered distribution; minimizes the nearest-neighbor distance.
- **displacement_factor** (float) – The smoothing factor n for constructing a convex hull; should obey $0 <= n <= 1$. Represents the degree of displacement of all atoms with respect to a spherical surface; $n = 1$ is a complete projection while $n = 0$ means no displacement at all.

  A non-zero value is generally recomended here, as the herein utilized ConvexHull class requires an adequate degree of surface-convexness, lest it fails to properly identify all valid surface points.
- ****kwargs** (Any) – Further keyword arguments for `distribute_idx()`.

Returns
A new Molecule with a subset of its surface atoms replaced with **symbol_new**.

Return type
Molecule

See also:

distribute_idx()
Create a new distribution of atomic indices from **idx** of length `f * len(idx)`.

identify_surface()
Take a molecule and identify which atoms are located on the surface, rather than in the bulk.

identify_surface_ch()
Identify the surface of a molecule using a convex hull-based approach.

2.14.2 nanoCAT.recipes.bulk

A short recipe for accessing the ligand-bulkiness workflow.

Index

bulk_workflow(smiles_list[, anchor, ...])	Start the CAT ligand bulkiness workflow with an iterable of smiles strings.
fast_bulk_workflow(smiles_list[, anchor, ...])	Start the ligand fast-bulkiness workflow with an iterable of smiles strings.

API

nanoCAT.recipes.**bulk_workflow**(*smiles_list*, *anchor='O(C=O)[H]'*, ***, *anchor_condition=None*, *diameter=4.5*, *height_lim=10.0*, *optimize=True*)

Start the CAT ligand bulkiness workflow with an iterable of smiles strings.

Examples

```
>>> from CAT.recipes import bulk_workflow

>>> smiles_list = [...]
>>> mol_list, bulk_array = bulk_workflow(smiles_list, optimize=True)
```

Parameters

- **smiles_list** (Iterable[str]) – An iterable of SMILES strings.
- **anchor** (str) – A SMILES string representation of an anchor group such as "O(C=O)[H]". The first atom will be marked as anchor atom while the last will be dissociated. Used for filtering molecules in **smiles_list**.
- **anchor_condition** (Callable[[int], bool], optional) – If not None, filter ligands based on the number of identified functional groups. For example, anchor_condition = lambda n: n == 1 will only accept ligands with a single **anchor** group, anchor_condition = lambda n: n >= 3 requires three or more anchors and anchor_condition = lambda n: n < 2 requires fewer than two anchors.
- **diameter** (float, optional) – The lattice spacing, *i.e.* the average nearest-neighbor distance between the anchor atoms of all ligads. Set to None to ignore the lattice spacing. Units should be in Angstrom.
- **height_lim** (float, optional) – A cutoff above which all atoms are ignored. Set to None to ignore the height cutoff. Units should be in Angstrom.
- **optimize** (bool) – Enable or disable the ligand geometry optimization.

Returns

A list of plams Molecules and a matching array of V_{bulk} values.

Return type

list[plams.Molecule] and np.ndarray[np.float64]

nanoCAT.recipes.**fast_bulk_workflow**(*smiles_list*, *anchor='O(C=O)[H]'*, ***, *anchor_condition=None*, *diameter=4.5*, *height_lim=10.0*, *func=<ufunc 'exp'>*)

Start the ligand fast-bulkiness workflow with an iterable of smiles strings.

Examples

```
>>> from CAT.recipes import fast_bulk_workflow

>>> smiles_list = [...]
>>> mol_list, bulk_array = fast_bulk_workflow(smiles_list, optimize=True)
```

Parameters

- **smiles_list** (Iterable[str]) – An iterable of SMILES strings.
- **anchor** (str) – A SMILES string representation of an anchor group such as "O(C=O)[H]". The first atom will be marked as anchor atom while the last will be dissociated. Used for filtering molecules in **smiles_list**.
- **anchor_condition** (Callable[[int], bool], optional) – If not None, filter ligands based on the number of identified functional groups. For example, anchor_condition = lambda n: n == 1 will only accept ligands with a single **anchor** group, anchor_condition = lambda n: n >= 3 requires three or more anchors and anchor_condition = lambda n: n < 2 requires fewer than two anchors.
- **diameter** (float, optional) – The lattice spacing, *i.e.* the average nearest-neighbor distance between the anchor atoms of all ligads. Set to None to ignore the lattice spacing. Units should be in Angstrom.
- **height_lim** (float, optional) – A cutoff above which all atoms are ignored. Set to None to ignore the height cutoff. Units should be in Angstrom.
- **func** (Callable[[np.float64], Any]) – A function for weighting each radial distance. Defaults to np.exp.

Returns

A list of plams Molecules and a matching array of V_{bulk} values.

Return type

list[plams.Molecule] and np.ndarray[np.float64]

Raises

RuntimeWarning – Issued if an exception is encountered when constructing or traversing one of the molecular graphs. The corresponding bulkiness value will be set to nan in such case.

2.14.3 nanoCAT.recipes.surface_dissociation

A recipe for dissociation specific sets of surface atoms.

Index

dissociate_surface(mol, idx[, symbol, ...])	A workflow for dissociating $(XY_n)_{\leq m}$ compounds from the surface of **mol**.
dissociate_bulk(mol, symbol_x[, symbol_y, ...])	A workflow for removing XY-based compounds from the bulk of **mol**.
row_accumulator(iterable)	Return a generator which accumulates elements along the nested elements of **iterable**.

API

nanoCAT.recipes.**dissociate_surface**(*mol*, *idx*, *symbol='Cl'*, *lig_count=1*, *k=4*, *displacement_factor=0.5*, ***kwargs*)

A workflow for dissociating $(XY_n)_{\leq m}$ compounds from the surface of **mol**.

The workflow consists of four distinct steps:

1. Identify which atoms Y, as specified by **symbol**, are located on the surface of **mol**.
2. Identify which surface atoms are neighbors of X, the latter being defined by **idx**.
3. Identify which pairs of $n * m$ neighboring surface atoms are furthest removed from each other. n is defined by **lig_count** and m, if applicable, by the index along axis 1 of **idx**.
4. Yield $(XY_n)_{\leq m}$ molecules constructed from **mol**.

Note: The indices supplied in **idx** will, when applicable, be sorted along its last axis.

Examples

```
>>> from pathlib import Path

>>> import numpy as np

>>> from scm.plams import Molecule
>>> from CAT.recipes import dissociate_surface, row_accumulator

>>> base_path = Path(...)
>>> mol = Molecule(base_path / 'mol.xyz')

# The indices of, e.g., Cs-pairs
>>> idx = np.array([
...     [1, 3],
...     [4, 5],
...     [6, 10],
...     [15, 12],
...     [99, 105],
...     [20, 4]
... ])

# Convert 1- to 0-based indices by substracting 1 from idx
```

(continues on next page)

(continued from previous page)

```
>>> mol_generator = dissociate_surface(mol, idx-1, symbol='Cl', lig_count=1)

# Note: The indices in idx are (always) be sorted along axis 1
>>> iterator = zip(row_accumulator(np.sort(idx, axis=1)), mol_generator)
>>> for i, mol in iterator:
...     mol.write(base_path / f'output{i}.xyz')
```

Parameters

- **mol** (Molecule) – The input molecule.
- **idx** (array-like, dimensions: ≤ 2) – An array of indices denoting to-be dissociated atoms (*i.e.* X); its elements will, if applicable, be sorted along the last axis. If a 2D array is provided then all elements along axis 1 will be dissociated in a cumulative manner. m is herein defined as the index along axis 1.
- **symbol** (str or int) – An atomic symbol or number defining the super-set of the atoms to-be dissociated in combination with **idx** (*i.e.* Y).
- **lig_count** (int) – The number of atoms specified in **symbol** to-be dissociated in combination with a single atom from **idx** (*i.e.* n).
- **k** (int) – The number of atoms specified in **symbol** which are surrounding a single atom in **idx**. Must obey the following condition: $k \geq 1$.
- **displacement_factor** (float) – The smoothing factor n for constructing a convex hull; should obey $0 <= n <= 1$. Represents the degree of displacement of all atoms with respect to a spherical surface; $n = 1$ is a complete projection while $n = 0$ means no displacement at all.

  A non-zero value is generally recomended here, as the herein utilized ConvexHull class requires an adequate degree of surface-convexness, lest it fails to properly identify all valid surface points.
- ****kwargs** (Any) – Further keyword arguments for *brute_uniform_idx()*.

Yields

Molecule – Yields new $(XY_n)_m$-dissociated molecules.

See also:

brute_uniform_idx()
Brute force approach to creating uniform or clustered distributions.

identify_surface()
Take a molecule and identify which atoms are located on the surface, rather than in the bulk.

identify_surface_ch()
Identify the surface of a molecule using a convex hull-based approach.

dissociate_ligand()
Remove XY_n from **mol** with the help of the *MolDissociater* class.

nanoCAT.recipes.**dissociate_bulk**(*mol*, *symbol_x*, *symbol_y=None*, *count_x=1*, *count_y=1*, *n_pairs=1*, *k=4*, *r_max=None*, *mode='uniform'*, ***kwargs*)

A workflow for removing XY-based compounds from the bulk of **mol**.

Examples

```
>>> from scm.plams import Molecule
>>> from CAT.recipes import dissociate_bulk

>>> mol: Molecule = ...

# Remove two PbBr2 pairs in a system where
# each lead atom is surrounded by 6 bromides
>>> mol_out1 = dissociate_bulk(
...     mol, symbol_x="Pb", symbol_y="Br", count_y=2, n_pairs=2, k=6
... )

# The same as before, expect all potential bromides are
# identified based on a radius, rather than a fixed number
>>> mol_out2 = dissociate_bulk(
...     mol, symbol_x="Pb", symbol_y="Br", count_y=2, n_pairs=2, r_max=5.0
... )

# Convert a fraction to a number of pairs
>>> f = 0.5
>>> count_x = 2
>>> symbol_x = "Pb"
>>> n_pairs = int(f * sum(at.symbol == symbol_x for at in mol) / count_x)

>>> mol_out3 = dissociate_bulk(
...     mol, symbol_x="Pb", symbol_y="Br", count_y=2, n_pairs=n_pairs, k=6
... )
```

Parameters

- **mol** (Molecule) – The input molecule.
- **symbol_x** (str or int) – The atomic symbol or number of the central (to-be dissociated) atom(s) X.
- **symbol_y** (str or int, optional) – The atomic symbol or number of the surrounding (to-be dissociated) atom(s) Y. If None, do not dissociate any atoms Y.
- **count_x** (int) – The number of central atoms X per individual to-be dissociated cluster.
- **count_y** (int) – The number of surrounding atoms Y per individual to-be dissociated cluster.
- **n_pairs** (int) – The number of to-be removed XY fragments.
- **k** (int, optional) – The total number of Y candidates surrounding each atom X. This value should be smaller than or equal to **count_y**. See the **r_max** parameter for a radius-based approach; note that both parameters are not mutually exclusive.
- **r_max** (int, optional) – The radius used for searching for Y candidates surrounding each atom X. See **k** parameter to use a fixed number of nearest neighbors; note that both parameters are not mutually exclusive.
- **mode** (str) – How the subset of to-be removed atoms X should be generated. Accepts one of the following values:

- "random": A random distribution.
- "uniform": A uniform distribution; the distance between each successive atom and all previous points is maximized.
- "cluster": A clustered distribution; the distance between each successive atom and all previous points is minmized.

Keyword Arguments
****kwargs** (Any) – Further keyword arguments for *CAT.distribution.distribute_idx()*.

Returns
The molecule with $n_{\text{pair}} * XY$ fragments removed.

Return type
Molecule

nanoCAT.recipes.**row_accumulator**(*iterable*)

Return a generator which accumulates elements along the nested elements of **iterable**.

Examples

```
>>> iterable = [[1, 3],
...             [4, 5],
...             [6, 10]]

>>> for i in row_accumulator(iterable):
...     print(repr(i))
'_1'
'_1_3'
'_4'
'_4_5'
'_6'
'_6_10'
```

Parameters
iterable (Iterable [Iterable [Any]]) – A nested iterable.

Yields
str – The accumulated nested elements of **iterable** as strings.

2.14.4 nanoCAT.recipes.charges

A short recipe for calculating and rescaling ligand charges.

Index

get_lig_charge(ligand, desired_charge[, ...])	Calculate and rescale the **ligand** charges using MATCH.

API

nanoCAT.recipes.**get_lig_charge**(*ligand*, *desired_charge*, *ligand_idx=None*, *invert_idx=False*, *settings=None*, *path=None*, *folder=None*)

Calculate and rescale the **ligand** charges using MATCH.

The atomic charges in **ligand_idx** wil be altered such that the molecular charge of **ligand** is equal to **desired_charge**.

Examples

```
>>> import pandas as pd
>>> from scm.plams import Molecule

>>> from CAT.recipes import get_lig_charge

>>> ligand = Molecule(...)
>>> desired_charge = 0.66
>>> ligand_idx = 0, 1, 2, 3, 4

>>> charge_series: pd.Series = get_lig_charge(
...     ligand, desired_charge, ligand_idx
... )

>>> charge_series.sum() == desired_charge
True
```

Parameters

- **ligand** (Molecule) – The input ligand.
- **desired_charge** (float) – The desired molecular charge of the ligand.
- **ligand_idx** (int or Iterable [int], optional) – An integer or iterable of integers representing atomic indices. The charges of these atoms will be rescaled; all others will be frozen with respect to the MATCH output. Setting this value to None means that *all* atomic charges are considered variable. Indices should be 0-based.
- **invert_idx** (bool) – If True invert **ligand_idx**, *i.e.* all atoms specified therein are now threated as constants and the rest as variables, rather than the other way around.
- **settings** (Settings, optional) – The input settings for MatchJob. Will default to the "top_all36_cgenff_new" forcefield if not specified.
- **path** (str or PathLike, optional) – The path to the PLAMS workdir as passed to init(). Will default to the current working directory if None.
- **folder** (str or PathLike, optional) – The name of the to-be created to the PLAMS working directory as passed to init(). Will default to "plams_workdir" if None.

Returns
A Series with the atom types of **ligand** as keys and atomic charges as values.

Return type
`pd.Series` [`str`, `float`]

See also:

`MatchJob`

A
class:~*scm.plams.core.basejob.Job* subclass for interfacing with MATCH: Multipurpose Atom-Typer for CHARMM.

2.14.5 nanoCAT.recipes.coordination_number

A recipe for calculating atomic coordination numbers.

Index

get_coordination_number(mol[, shell, d_outer])	Take a molecule and identify the coordination number of each atom.
coordination_outer(dist, d_outer, length)	Calculate the coordination number relative to the outer shell.

API

nanoCAT.recipes.**get_coordination_number**(*mol*, *shell='inner'*, *d_outer=None*)

Take a molecule and identify the coordination number of each atom.

The function first compute the pair distance between all reference atoms in **mol**. The number of first neighbors, defined as all atoms within a threshold radius **d_inner** is then count for each atom. The threshold radius can be changed to a desired value **d_outer** (in angstrom) to obtain higher coordination numbers associated to outer coordination shells. The function finally groups the (1-based) indices of all atoms in **mol** according to their atomic symbols and coordination numbers.

Parameters

- **mol** (array-like [`float`], shape $(n, 3)$) – An array-like object with the Cartesian coordinates of the molecule.
- **shell** (`str`) – The coordination shell to be considered. Only `'inner'` or `'outer'` values are accepted. The default, `'inner'`, refers to the first coordination shell.
- **d_outer** (`float`, optional) – The threshold radius for defining which atoms are considered as neighbors. The default, `None`, is accepted only if `shell` is `'inner'`

Returns
A nested dictionary `{'Cd': {8: [0, 1, 2, 3, 4, ...], ...}, ...}` containing lists of (1-based) indices refered to the atoms in **mol** having a given atomic symbol (*e.g.* `'Cd'`) and coordination number (*e.g.* `8`).

Return type
`dict`

Raises

- **TypeError** – Raised if no threshold radius is defined for the outer coordination shell.
- **ValueError** – Raised if a wrong value is attributed to `shell`.

See also:

guess_core_core_dist()
Estimate a value for **d_inner** based on the radial distribution function of **mol**. Can also be used to estimate **d_outer** as the distance between the atom pairs ('A', 'B').

nanoCAT.recipes.**coordination_outer**(*dist*, *d_outer*, *length*)

Calculate the coordination number relative to the outer shell.

A module for multiple compound attachment and export of the .xyz files.

2.14.6 Index

add_ligands(core_dir, ligand_dir[, ...])	Add ligand(s) to one core.
export_dyes(mol_list[, new_dir, err_dir, ...])	Exports molecular coordinates to .xyz files.
sa_scores(mols[, filename])	Calculate the synthetic accessibility score for all molecules in **mols**.

2.14.7 API

CAT.dye.addlig.**add_ligands**(*core_dir*, *ligand_dir*, *min_dist=1.2*, *n=1*, *symmetry=()*)

Add ligand(s) to one core.

Parameters

- **core_dir** (*str*) – Name of directory where core coordinates are located
- **ligand_dir** (*str*) – Name of directory where ligands coordinates are located
- **min_dist** (*float*) – Criterion for the minimal interatomic distances
- **n** (*int*) – Number of substitutions
- **symmetry** (*tuple[str]*) – Keywords for substitution symmetry for deleting equivalent molecules

Returns
New structures that are containg core and lingad fragments

Return type
Iterator[Molecule]

CAT.dye.addlig.**export_dyes**(*mol_list*, *new_dir='new_molecules'*, *err_dir='err_molecules'*, *min_dist=1.2*)

Exports molecular coordinates to .xyz files.

Parameters

- **mol_list** (*Iterable[Molecule]*) – List of molecules
- **new_dir** (*str*) – Name of the durectory to place new structures
- **err_dir** (*str*) – Name of the directory for molecules that do not fullfill min_dist criterion

- **min_dist** (*float*) – Criterion for the minimal interatomic distances

CAT.dye.addlig.**sa_scores**(*mols*, *filename=None*)

Calculate the synthetic accessibility score for all molecules in **mols**.

2.14.8 nanoCAT.recipes.multi_ligand_job

Estimate forcefield parameters using MATCH and then run a MM calculation with CP2K.

Examples

```
>>> from qmflows import Settings
>>> from qmflows.templates import geometry
>>> from qmflows.packages import Result
>>> from scm.plams import Molecule

>>> from CAT.recipes import multi_ligand_job

>>> mol = Molecule(...)
>>> psf = str(...)

# Example input settings for a geometry optimization
>>> settings = Settings()
>>> settings.specific.cp2k += geometry.specific.cp2k_mm
>>> settings.charge = {
...     'param': 'charge',
...     'Cd': 2,
...     'Se': -2
... }
>>> settings.lennard_jones = {
...     'param': ('epsilon', 'sigma'),
...     'unit': ('kcalmol', 'angstrom'),
...     'Cd Cd': (1, 1),
...     'Se Se': (2, 2),
...     'Se Se': (3, 3)
... }

>>> results: Result = multi_ligand_job(mol, psf, settings)
```

param mol
The input molecule.

type mol
Molecule

param psf
A PSFContainer or path to a .psf file.

type psf
PSFContainer or path-like

param settings
The QMFlows-style CP2K input settings.

type settings
Settings

param path
The path to the PLAMS working directory.

type path
path-like, optional

param folder
The name of the PLAMS working directory.

type folder
path-like, optional

param **kwargs
Further keyword arguments for `qmflows.cp2k_mm()`.

type **kwargs
Any

returns
The results of the `CP2KMM` calculation.

rtype
`CP2KMM_Result`

See also:

`FOX.recipes.generate_psf2()`
Generate a `PSFContainer` instance for **qd** with multiple different **ligands**.

`qmflows.cp2k_mm()`
An instance of `CP2KMM`; used for running classical forcefield calculations with CP2K.

10.1002/jcc.21963
MATCH: An atom-typing toolset for molecular mechanics force fields, J.D. Yesselman, D.J. Price, J.L. Knight and C.L. Brooks III, J. Comput. Chem., 2011.

2.14.9 nanoCAT.recipes.mol_filter

Recipes for filtering molecules.

Index

get_mol_length(mol, atom)	Return the distance between **atom** and the atom in **mol** which it is furthest removed from.
filter_mol(mol_list, data, filter)	Filter **mol_list** and **data** based on elements from **mol_list**.
filter_data(mol_list, data, filter)	Filter **mol_list** and **data** based on elements from **data**.

API

nanoCAT.recipes.**get_mol_length**(*mol*, *atom*)

Return the distance between **atom** and the atom in **mol** which it is furthest removed from.

Examples

Use the a molecules length for filtering a list of molecules:

```
>>> from CAT.recipes import get_mol_length, filter_mol
>>> from scm.plams import Molecule

>>> mol_list = [Molecule(...), ...]
>>> data = [...]
>>> filter = lambda mol: get_mol_length(mol, mol.properties.get('anchor')) < 10

>>> mol_dict = filter_mol(mol_list, data, filter=filter)
```

Parameters

- **mol** (Molecule or numpy.ndarray) – A PLAMS molecule or a 2D numpy array with a molecules Cartesian coordinates.
- **atom** (Atom or numpy.ndarray) – A PLAMS atom or a 1D numpy array with an atoms Cartesian coordinates.

Returns

The largest distance between **atom** and all other atoms **mol**.

Return type

float

See also:

filter_mol()

Filter **mol_list** and **data** based on elements from **mol_list**.

nanoCAT.recipes.**filter_mol**(*mol_list*, *data*, *filter*)

Filter **mol_list** and **data** based on elements from **mol_list**.

Examples

```
>>> from scm.plams import Molecule
>>> from CAT.recipes import filter_mol

>>> mol_list = [Molecule(...), ...]
>>> data = [...]
>>> mol_dict1 = filter_mol(mol_list, data, filter=lambda n: n < 10)

>>> prop1 = [...]
>>> prop2 = [...]
>>> prop3 = [...]
>>> multi_data = zip([prop1, prop2, prop3])
```

(continues on next page)

(continued from previous page)

```
>>> mol_dict2 = filter_mol(mol_list, multi_data, filter=lambda n: n < 10)

>>> keys = mol_dict1.keys()
>>> values = mol_dict1.values()
>>> mol_dict3 = filter_mol(keys, values, filter=lambda n: n < 5)
```

Parameters

- **mol_list** (Iterable [Molecule]) – An iterable of the, to-be filtered, PLAMS molecules.
- **data** (Iterable[T]) – An iterable which will be assigned as values to the to-be returned dict. These parameters will be filtered in conjunction with **mol_list**. Note that **mol_list** and **data** *should* be of the same length.
- **filter** (Callable[[Molecule], bool]) – A callable for filtering the distance vector. An example would be `lambda n: max(n) < 10`.

Returns
A dictionary with all (filtered) molecules as keys and elements from **data** as values.

Return type
dict [Molecule, T]

See also:

filter_data()
Filter **mol_list** and **data** based on elements from **data**.

nanoCAT.recipes.**filter_data**(*mol_list*, *data*, *filter*)

Filter **mol_list** and **data** based on elements from **data**.

Examples

See *filter_mol()* for a number of input examples.

Parameters

- **mol_list** (Iterable [Molecule]) – An iterable of the, to-be filtered, PLAMS molecules.
- **data** (Iterable[T]) – An iterable which will be assigned as values to the to-be returned dict. These parameters will be filtered in conjunction with **mol_list**. Note that **mol_list** and **data** *should* be of the same length.
- **filter** (Callable[[T], bool]) – A callable for filtering the elements of **data**. An example would be `lambda n: n < 10`.

Returns
A dictionary with all (filtered) molecules as keys and elements from **data** as values.

Return type
dict [Molecule, T]

See also:

filter_mol()
Filter **mol_list** and **data** based on elements from **mol_list**.

2.14.10 nanoCAT.recipes.cdft_utils

Recipes for running conceptual dft calculations.

Index

run_jobs(mol, *settings[, job_type, ...])	Run multiple jobs in succession.
get_global_descriptors(results)	Extract a dictionary with all ADF conceptual DFT global descriptors from **results**.
cdft	Automatic multi-level dictionary.

API

nanoCAT.recipes.**run_jobs**(*mol*, **settings*, *job_type=<function adf(self, settings, mol, job_name='', validate_output=True, **kwargs)>*, *job_name=None*, *path=None*, *folder=None*, ***kwargs*)

Run multiple jobs in succession.

Examples

```
>>> from scm.plams import Molecule
>>> from qmflows import Settings
>>> from qmflows.templates import geometry
>>> from qmflows.utils import InitRestart
>>> from qmflows.packages.SCM import ADF_Result

>>> from CAT.recipes import run_jobs, cdft

>>> mol = Molecule(...)

>>> settings_opt = Settings(...)
>>> settings_opt += geometry
>>> settings_cdft = Settings(...)
>>> settings_cdft += cdft

>>> result: ADF_Result = run_jobs(mol, settings_opt, settings_cdft)
```

Parameters

- **mol** (Molecule) – The input molecule.
- ***settings** (Mapping) – One or more input settings. A single job will be run for each provided settings object. The output molecule of each job will be passed on to the next one.
- **job_type** (Package) – A QMFlows package instance.

- **job_name** (str, optional) – The name basename of the job. The name will be append with ".{i}", where {i} is the number of the job.
- **path** (str or PathLike, optional) – The path to the working directory.
- **folder** (str or PathLike, optional) – The name of the working directory.
- ****kwargs** (Any) – Further keyword arguments for **job_type** and the noodles job runner.

Returns
A QMFlows Result object as constructed by the last calculation. The exact type depends on the passed **job_type**.

Return type
Result

See also:

noodles.run.threading.sqlite3.run_parallel()
Run a workflow in parallel threads, storing results in a Sqlite3 database.

nanoCAT.recipes.**get_global_descriptors**(*results*)

Extract a dictionary with all ADF conceptual DFT global descriptors from **results**.

Examples

```
>>> import pandas as pd
>>> from scm.plams import ADFResults
>>> from CAT.recipes import get_global_descriptors

>>> results = ADFResults(...)

>>> series: pd.Series = get_global_descriptors(results)
>>> print(dct)
Electronic chemical potential (mu)     -0.113
Electronegativity (chi=-mu)             0.113
Hardness (eta)                          0.090
Softness (S)                           11.154
Hyperhardness (gamma)                  -0.161
Electrophilicity index (w=omega)        0.071
Dissocation energy (nucleofuge)         0.084
Dissociation energy (electrofuge)       6.243
Electrodonating power (w-)              0.205
Electroaccepting power(w+)              0.092
Net Electrophilicity                    0.297
Global Dual Descriptor Deltaf+          0.297
Global Dual Descriptor Deltaf-         -0.297
Electronic chemical potential (mu+)    -0.068
Electronic chemical potential (mu-)    -0.158
Name: global descriptors, dtype: float64
```

Parameters
results (plams.ADFResults or qmflows.ADF_Result) – A PLAMS Results or QMFlows Result instance of an ADF calculation.

Returns
A Series with all ADF global decsriptors as extracted from **results**.

Return type
pandas.Series

nanoCAT.recipes.**cdft = qmflows.Settings(...)**

A QMFlows-style template for conceptual DFT calculations.

```
specific:
    adf:
        symmetry: nosym
        conceptualdft:
            enabled: yes
            analysislevel: extended
            electronegativity: yes
            domains:
                enabled: yes
        qtaim:
            enabled: yes
            analysislevel: extended
            energy: yes
        basis:
            core: none
            type: DZP
        xc:
            libxc:
                CAM-B3LYP
        numericalquality: good
```

2.14.11 nanoCAT.recipes.entropy

A recipe for calculating the rotational and translational entropy.

Index

get_entropy(mol[, temp])	Calculate the translational of the passsed molecule.

API

nanoCAT.recipes.**get_entropy**(*mol*, *temp=298.15*)

Calculate the translational of the passsed molecule.

Parameters

- **mol** (Molecule) – A PLAMS molecule.
- **temp** (float) – The temperature in Kelvin.

Returns
Two floats respectively representing the translational and rotational entropy. Units are in kcal/mol/K

Return type
`float` & `float`

2.14.12 nanoCAT.recipes.fast_sigma

A recipe for calculating specific COSMO-RS properties using the fast-sigma approximation.

Index

run_fast_sigma(input_smiles, solvents, *[, ...])	Perform (fast-sigma) COSMO-RS property calculations on the passed SMILES and solvents.
get_compkf(smiles[, directory, name])	Estimate the sigma profile of a SMILES string using the COSMO-RS fast-sigma method.
read_csv(file, *[, columns])	Read the passed .csv file as produced by *run_fast_sigma()*.
sanitize_smiles_df(df[, column_levels, ...])	Sanitize the passed dataframe, canonicalizing the SMILES in its index, converting the columns into a multiIndex and removing duplicate entries.

API

nanoCAT.recipes.**run_fast_sigma**(*input_smiles*, *solvents*, *, *output_dir='crs'*, *ams_dir=None*, *chunk_size=100*, *processes=None*, *return_df=False*, *log_options=mappingproxy({'file': 5, 'stdout': 3, 'time': True, 'date': False})*)

Perform (fast-sigma) COSMO-RS property calculations on the passed SMILES and solvents.

The output is exported to the `cosmo-rs.csv` file.

Includes the following properties:

- LogP
- Activety Coefficient
- Solvation Energy
- Formula
- Molar Mass
- Nring
- boilingpoint
- criticalpressure
- criticaltemp
- criticalvol
- density
- dielectricconstant
- entropygas
- flashpoint

- gidealgas
- hcombust
- hformstd
- hfusion
- hidealgas
- hsublimation
- meltingpoint
- molarvol
- parachor
- solubilityparam
- tpt
- vdwarea
- vdwvol
- vaporpressure

Jobs are performed in parallel, with chunks of a given size being distributed to a user-specified number of processes and subsequently cashed. After all COSMO-RS calculations have been performed, the temporary .csv files are concatenated into `cosmo-rs.csv`.

Examples

```
>>> import os
>>> import pandas as pd
>>> from nanoCAT.recipes import run_fast_sigma

>>> output_dir: str = ...
>>> smiles_list = ["CO[H]", "CCO[H]", "CCCO[H]"]
>>> solvent_dict = {
...     "water": "$AMSRESOURCES/ADFCRS/Water.coskf",
...     "octanol": "$AMSRESOURCES/ADFCRS/1-Octanol.coskf",
... }

>>> run_fast_sigma(smiles_list, solvent_dict, output_dir=output_dir)

>>> csv_file = os.path.join(output_dir, "cosmo-rs.csv")
>>> pd.read_csv(csv_file, header=[0, 1], index_col=0)
property Activity Coefficient             ... Solvation Energy
solvent               octanol      water  ...          octanol     water
smiles                                    ...
CO[H]                1.045891   4.954782  ...        -2.977354 -3.274420
CCO[H]               0.980956  12.735228  ...        -4.184214 -3.883986
CCCO[H]              0.905952  47.502557  ...        -4.907177 -3.779867

[3 rows x 8 columns]
```

Parameters

- **input_smiles** (Iterable[str]) – The input SMILES strings.
- **solvents** (Mapping[str, path-like]) – A mapping with solvent-names as keys and paths to their respective .coskf files as values.

Keyword Arguments

- **output_dir** (path-like object) – The directory wherein the .csv files will be stored. A new directory will be created if it does not yet exist.
- **plams_dir** (path-like, optional) – The directory wherein all COSMO-RS computations will be performed. If None, use a temporary directory inside **output_dir**.
- **chunk_size** (int) – The (maximum) number of entries to-be stored in a single .csv file.
- **processes** (int, optional) – The number of worker processes to use. If None, use the number returned by os.cpu_count().
- **return_df** (bool) – If True, return a dataframe with the content of cosmo-rs.csv.
- **log_options** (Mapping[str, Any]) – Alternative settings for plams.config.log. See the PLAMS documentation for more details.

nanoCAT.recipes.**get_compkf**(*smiles*, *directory=None*, *name=None*)

Estimate the sigma profile of a SMILES string using the COSMO-RS fast-sigma method.

See the COSMO-RS docs for more details.

Parameters

- **smiles** (str) – The SMILES string of the molecule of interest.
- **directory** (str, optional) – The directory wherein the resulting .compkf file should be stored. If None, use the current working directory.
- **name** (str) – The name of the to-be created .compkf file (excluding extensions). If None, use **smiles**.

Returns

The absolute path to the created .compkf file. None will be returned if an error is raised by AMS.

Return type

str, optional

nanoCAT.recipes.**read_csv**(*file*, *, *columns=None*, ***kwargs*)

Read the passed .csv file as produced by *run_fast_sigma()*.

Examples

```
>>> from nanoCAT.recipes import read_csv

>>> file: str = ...

>>> columns1 = ["molarvol", "gidealgas", "Activity Coefficient"]
>>> read_csv(file, usecols=columns1)
property  molarvol  gidealgas Activity Coefficient
solvent        NaN        NaN              octanol     water
smiles
CCCO[H]   0.905952  47.502557          -153.788589  0.078152
```

(continues on next page)

(continued from previous page)

```
CCO[H]    0.980956  12.735228          -161.094955  0.061220
CO[H]     1.045891   4.954782                  NaN       NaN

>>> columns2 = [("Solvation Energy", "water")]
>>> read_csv(file, usecols=columns2)
property Solvation Energy
solvent             water
smiles
CCCO[H]         -3.779867
CCO[H]          -3.883986
CO[H]           -3.274420
```

Parameters

- **file** (path-like object) – The name of the to-be opened .csv file.
- **columns** (*key or sequence of keys, optional*) – The to-be read columns. Note that any passed value must be a valid dataframe (multiindex) key.
- ****kwargs** (Any) – Further keyword arguments for `pd.read_csv`.

See also:

pd.read_csv
Read a comma-separated values (csv) file into DataFrame.

nanoCAT.recipes.**sanitize_smiles_df**(*df*, *column_levels=2*, *column_padding=None*)

Sanitize the passed dataframe, canonicalizing the SMILES in its index, converting the columns into a multiIndex and removing duplicate entries.

Examples

```
>>> import pandas as pd
>>> from nanoCAT.recipes import sanitize_smiles_df

>>> df: pd.DataFrame = ...
>>> print(df)
         a
smiles
CCCO[H]  1
CCO[H]   2
CO[H]    3

>>> sanitize_smiles_df(df)
         a
       NaN
smiles
CCCO     1
CCO      2
CO       3
```

Parameters

- **df** (pd.DataFrame) – The dataframe in question. The dataframes' index should consist of smiles strings.
- **column_levels** (int) – The number of multiindex column levels that should be in the to-be returned dataframe.
- **column_padding** (Hashable) – The object used as padding for the multiindex levels (where appropiate).

Returns
The newly sanitized dataframe. Returns either the initially passed dataframe or a copy thereof.

Return type
pd.DataFrame

2.15 Multi-ligand attachment

optional.qd.**multi_ligand**

All settings related to the multi-ligand attachment procedure.

Example:

```
optional:
    qd:
        multi_ligand:
            ligands:
                - OCCC
                - OCCCCCCC
                - OCCCCCCCCCCCC
            anchor:
                - F
                - Br
                - I
```

optional.qd.multi_ligand.**ligands**

Parameter

- **Type** - list [str]

SMILES strings of to-be attached ligands.

Note that these ligands will be attached *in addition* to whichever ligands are specified in *input_cores & input_ligands*.

Note: This argument has no value be default and must thus be provided by the user.

optional.qd.multi_ligand.**anchor**

Parameter

- **Type** - `list` [`str` or `int`]

Atomic number of symbol of the core anchor atoms.

The first anchor atom will be assigned to the first ligand in *multi_ligand.ligands*, the second anchor atom to the second ligand, *etc.*. The list's length should consequently be of the same length as *multi_ligand.ligands*.

Works analogous to *optional.core.anchor*.

This optiona can alternatively be provided as `optional.qd.multi_ligand.dummy`.

Note: This argument has no value be default and must thus be provided by the user.

2.16 Subset Generation

Functions for creating distributions of atomic indices (*i.e.* core anchor atoms).

2.16.1 Index

uniform_idx(dist[, operation, cluster_size, ...])	Yield the column-indices of **dist** which yield a uniform or clustered distribution.
distribute_idx(core, idx, f[, mode])	Create a new distribution of atomic indices from **idx** of length `f * len(idx)`.

2.16.2 API

`CAT.distribution.`**uniform_idx**(*dist*, *operation='min'*, *cluster_size=1*, *start=None*, *randomness=None*, *weight=<function <lambda>>*)

Yield the column-indices of **dist** which yield a uniform or clustered distribution.

Given the (symmetric) distance matrix $\boldsymbol{D} \in \mathbb{R}^{n,n}$ and the vector $\boldsymbol{a} \in \mathbb{N}^{m}$ (representing a subset of indices in D), then the i'th element a_i is defined below. All elements of $\boldsymbol{a}$ are furthermore constrained to be unique. $f(x)$ is herein a, as of yet unspecified, function for weighting each individual distance.

Following the convention used in python, the $\boldsymbol{X}[0:3, 1:5]$ notation is herein used to denote the submatrix created by intersecting rows 0 up to (but not including) 3 and columns 1 up to (but not including) 5.

$$*\text{argmin} a_i = \begin{cases} _{k \in \mathbb{N}} \sum f\big(\boldsymbol{D}_{k,:}\big) & \text{if} \\ i = 0 & \\ _{k \in \mathbb{N}} \sum f\big(\boldsymbol{D}[k, \boldsymbol{a}[0:i]]\big) & \text{if} \\ i > 0 & \end{cases}$$

Default weighting function: $f(x) = e^{-x}$.

The row in D corresponding to a_0 can alternatively be specified by **start**.

The argmin operation can be exchanged for argmax by setting **operation** to `"max"`, thus yielding a clustered- rather than uniform-distribution.

The **cluster_size** parameter allows for the creation of uniformly distributed clusters of size r. Herein the vector of indices, $\boldsymbol{a} \in \mathbb{N}^m$ is for the purpose of book keeping reshaped into the matrix $\boldsymbol{A} \in \mathbb{N}^{q,r}$ with $q * r = m$. All elements of $\boldsymbol{A}$ are, again, constrained to be unique.

$$*\text{argmin} A_{i,j} = \begin{cases} \underset{k \in \mathbb{N}}{\sum} f\big(\boldsymbol{D}_{k,:}\big) & \text{if} \\ i = 0;\ j = 0 & \\ \underset{k \in \mathbb{N}}{\sum} f\big(\boldsymbol{D}[k; \boldsymbol{A}[0:i, 0:r]\big) & \text{if} \\ i > 0;\ j = 0 & \\ \underset{k \in \mathbb{N}}{\dfrac{\sum f\big(\boldsymbol{D}[k, \boldsymbol{A}[0:i, 0:r]\big)}{\sum f\big(\boldsymbol{D}[k, \boldsymbol{A}[i, 0:j]\big)}} & \text{if} \\ j > 0 & \end{cases}$$

Examples

```
>>> import numpy as np
>>> from CAT.distribution import uniform_idx

>>> dist: np.ndarray = np.random.rand(10, 10)

>>> out1 = uniform_idx(dist)
>>> idx_ar1 = np.fromiter(out1, dtype=np.intp)

>>> out2 = uniform_idx(dist, operation="min")
>>> out3 = uniform_idx(dist, cluster_size=5)
>>> out4 = uniform_idx(dist, cluster_size=[1, 1, 1, 1, 2, 2, 4])
>>> out5 = uniform_idx(dist, start=5)
>>> out6 = uniform_idx(dist, randomness=0.75)
>>> out7 = uniform_idx(dist, weight=lambda x: x**-1)
```

Parameters

- **dist** (numpy.ndarray [float], shape (n, n)) – A symmetric 2D NumPy array ($D_{i,j} = D_{j,i}$) representing the distance matrix D.
- **operation** (str) – Whether to use argmin() or argmax(). Accepted values are "min" and "max".
- **cluster_size** (int or Iterable [int]) – An integer or iterable of integers representing the size of clusters. Used in conjunction with operation = "max" for creating a uniform distribution of clusters. cluster_size = 1 is equivalent to a normal uniform distribution.

  Providing **cluster_size** as an iterable of integers will create clusters of varying, user-specified, sizes. For example, cluster_size = range(1, 4) will continuesly create clusters of sizes 1, 2 and 3. The iteration process is repeated until all atoms represented by **dist** are exhausted.
- **start** (int, optional) – The index of the starting row in **dist**. If None, start in whichever row contains the global minimum ($*\text{argmin}_{k \in \mathbb{N}} ||\boldsymbol{D}_{k,:}||_p$) or maximum ($*\text{argmax}_{k \in \mathbb{N}} ||\boldsymbol{D}_{k,:}||_p$). See **operation**.
- **randomness** (float, optional) – If not None, represents the probability that a random index will be yielded rather than obeying **operation**. Should obey the following condition: $0 \leq randomness \leq 1$.

- **weight** (Callable) – A callable for applying weights to the distance; default: e^{-x}. The callable should take an array as argument and return a new array, *e.g.* `numpy.exp()`.

Yields

int – Yield the column-indices specified in $\boldsymbol{d}$.

CAT.distribution.**distribute_idx**(*core*, *idx*, *f*, *mode='uniform'*, ***kwargs*)

Create a new distribution of atomic indices from **idx** of length `f * len(idx)`.

Parameters

- **core** (array-like [float], shape $(m, 3)$) – A 2D array-like object (such as a `Molecule` instance) consisting of Cartesian coordinates.
- **idx** (int or Iterable [int], shape $(i,)$) – An integer or iterable of unique integers representing the 0-based indices of all anchor atoms in **core**.
- **f** (float) – A float obeying the following condition: $0.0 < f \leq 1.0$. Represents the fraction of **idx** that will be returned.
- **mode** (str) – How the subset of to-be returned indices will be generated. Accepts one of the following values:
  - `"random"`: A random distribution.
  - `"uniform"`: A uniform distribution; the distance between each successive atom and all previous points is maximized.
  - `"cluster"`: A clustered distribution; the distance between each successive atom and all previous points is minmized.
- ****kwargs** (Any) – Further keyword arguments for the **mode**-specific functions.

Returns

A 1D array of atomic indices. If **idx** has i elements, then the length of the returned list is equal to $\max(1, f * i)$.

Return type

numpy.ndarray [int], shape $(f * i,)$

See also:

uniform_idx()

Yield the column-indices of **dist** which yield a uniform or clustered distribution.

cluster_idx()

Return the column-indices of **dist** which yield a clustered distribution.

2.17 identify_surface

2.17.1 nanoCAT.bde.identify_surface

A module for identifying which atoms are located on the surface, rather than in the bulk

Index

identify_surface(mol[, max_dist, tolerance, ...])	Take a molecule and identify which atoms are located on the surface, rather than in the bulk.
identify_surface_ch(mol[, n, invert])	Identify the surface of **mol** using a convex hull-based approach.

API

nanoCAT.bde.identify_surface.**identify_surface**(*mol*, *max_dist=None*, *tolerance=0.5*, *compare_func=<built-in function gt>*)

Take a molecule and identify which atoms are located on the surface, rather than in the bulk.

The function compares the position of all reference atoms in **mol** with its direct neighbors, the latter being defined as all atoms within a radius **max_dist**. The distance is then calculated between the reference atoms and the mean-position of its direct neighbours. A length of 0 means that the atom is surrounded in a spherical symmetric manner, *i.e.* it must be located in the bulk. Deviations from 0 conversely imply that an atom is located on the surface.

Parameters

- **mol** (array-like [float], shape $(n, 3)$) – An array-like object with the Cartesian coordinates of the molecule.
- **max_dist** (float, optional) – The radius for defining which atoms constitute as neighbors. If None, estimate this value using the radial distribution function of **mol**.
- **tolerance** (float) – The tolerance for considering atoms part of the surface. A higher value will impose stricter criteria, which might be necasary as the local symmetry of **mol** becomes less pronounced. Should be in the same units as the coordinates of **mol**.
- **compare_func** (Callable) – The function for evaluating the direct-neighbor distance. The default, __gt__(), is equivalent to identifying the surface, while *e.g.* __lt__() identifies the bulk.

Returns

The (0-based) indices of all atoms in **mol** located on the surface.

Return type

numpy.ndarray [int], shape $(n,)$

Raises

ValueError – Raised if no atom-pairs are found within the distance **max_dist**. Implies that either the user-specified or guessed value is too small.

See also:

guess_core_core_dist()

Estimate a value for **max_dist** based on the radial distribution function of **mol**.

nanoCAT.bde.identify_surface.**identify_surface_ch**(*mol*, *n=0.5*, *invert=False*)

Identify the surface of **mol** using a convex hull-based approach.

A convex hull represents the smallest set of points enclosing itself, thus defining a surface.

Parameters

- **mol** (array-like [float], shape $(n, 3)$) – A 2D array-like object of Cartesian coordinates representing a polyhedron. The supplied polyhedron should be convex in shape.
- **n** (float) – Smoothing factor for constructing a convex hull. Should obey $0 <= n <= 1$. Represents the degree of displacement of all atoms to a spherical surface; $n = 1$ is a complete projection while $n = 0$ means no displacement at all.

  A non-zero value is generally recomended here, as the herein utilized ConvexHull class requires an adequate degree of surface-convexness, lest it fails to properly identify all valid surface points.
- **invert** (bool) – If True, return the indices of all atoms in the bulk rather than on the surface.

Returns
The (0-based) indices of all atoms in **mol** located on the surface.

Return type
numpy.ndarray [int], shape $(n,)$

See also:

ConvexHull
Convex hulls in N dimensions.

2.18 distribution_brute

Functions for creating distributions of atomic indices using brute-force approaches.

2.18.1 Index

brute_uniform_idx(mol, idx[, n, operation, ...])	Brute force approach to creating uniform or clustered distributions.

2.18.2 API

CAT.attachment.distribution_brute.**brute_uniform_idx**(*mol*, *idx*, *n=2*, *operation='min'*, *weight=<function <lambda>>*)

Brute force approach to creating uniform or clustered distributions.

Explores, and evaluates, all valid combinations of size n constructed from the k atoms in **neighbor** closest to each atom in **center**.

The combination where the n atoms are closest (`operation = 'max'`) or furthest removed from each other (`operation = 'min'`) is returned.

Parameters

- **mol** (array-like [float], shape $(m, 3)$) – An array-like object with Cartesian coordinate representing a collection of central atoms.
- **idx** (array-like [int], shape (l, p)) – An array-like object with indices in **mol**. Combinations will be explored and evaluated along axis `-1` of the passed array.
- **n** (int) – The number of to-be returned opposing atoms. Should be larger than or equal to 1.

- **operation** (str) – Whether to evaluate the weighted distance using argmin() or argmax(). Accepted values are "min" and "max".
- **weight** (Callable) – A callable for applying weights to the distance; default: e^{-x}. The callable should take an array as argument and return a new array, *e.g.* numpy.exp().

Returns
An array with indices of opposing atoms.

Return type
numpy.ndarray [int], shape (m, n)

See also:

uniform_idx()
Yield the column-indices of **dist** which yield a uniform or clustered distribution.

2.19 guess_core_dist

2.19.1 nanoCAT.bde.guess_core_dist

A module for estimating ideal values for ["optional"]["qd"]["bde"]["core_core_dist"] .

Index

guess_core_core_dist(mol[, atom, dr, r_max, ...])	Guess a value for the ["optional"]["qd"]["bde"]["core_core_dist"] parameter in **CAT**.

API

nanoCAT.bde.guess_core_dist.**guess_core_core_dist**(*mol*, *atom=None*, *dr=0.1*, *r_max=8.0*, *window_length=21*, *polyorder=7*)

Guess a value for the ["optional"]["qd"]["bde"]["core_core_dist"] parameter in **CAT**.

The estimation procedure involves finding the first minimum in the radial distribution function (RDF) of **mol**. After smoothing the RDF wth a Savitzky-Golay filer, the gradient of the RDF is explored (starting from the RDFs' global maximum) until a stationary point is found with a positive second derivative (*i.e.* a minimum).

Examples

```
>>> from scm.plams import Molecule
>>> from nanoCAT.bde.guess_core_dist import guess_core_core_dist

>>> atom1 = 'Cl'  # equivalent to ('Cl', 'Cl')
>>> atom2 = 'Cl', 'Br'

>>> mol = Molecule(...)

>>> guess_core_core_dist(mol, atom1)
>>> guess_core_core_dist(mol, atom2)
```

Parameters

- **mol** (array-like [float], shape $(n, 3)$) – A molecule.
- **atom** (str or int, optional) – An atomic number or symbol for defining an atom subset within **mol**. The RDF is constructed for this subset. Providing a 2-tuple will construct the RDF between these 2 atom subsets.
- **dr** (float) – The RDF integration step-size in Angstrom, *i.e.* the distance between concentric spheres.
- **r_max** (float) – The maximum to be evaluated interatomic distance in the RDF.
- **window_length** (int) – The length of the filter window (*i.e.* the number of coefficients) for the Savitzky-Golay filter.
- **polyorder** (int) – The order of the polynomial used to fit the samples for the Savitzky-Golay filter.

Returns
The interatomic radius of the first RDF minimum (following the first maximum).

Return type
float

Raises

- **MoleculeError** – Raised if **atom** is not in **mol**.
- **ValueError** – Raised if no minimum is found in the smoothed RDF.

See also:

savgol_filter()
Apply a Savitzky-Golay filter to an array.

2.20 Importing Quantum Dots

WiP: Import pre-built quantum dots rather than constructing them from scratch.

2.20.1 Default Settings

```
input_qd:
    - Cd68Se55_ethoxide.xyz:
        ligand_smiles: '[O-]CC'
        ligand_anchor: '[O-]'
```

2.20.2 Arguments

`ligand_smiles`

Parameter

- **Type** - `str`
- **Default value** – `None`

A SMILES string representing the ligand. The provided SMILES string will be used for identifying the core and all ligands.

> **Warning:** This argument has no value be default and thus *must* be provided by the user.

`ligand_anchor`

Parameter

- **Type** - `str`
- **Default value** – `None`

A SMILES string representing the achor functional group of the ligand. If the provided SMILES string consists of multiple atoms (*e.g.* a carboxylate: `"[O-]C=O"`), than the first atom will be treated as anchor (`"[O-]"`).

> **Warning:** This argument has no value be default and thus *must* be provided by the user.

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

M

N

O

P

Q

R

S

T

U

V

W

X