Contents:
**AUTOMATED FORCEFIELD OPTIMIZATION EXTENSION 0.7.4**

**Auto-FOX** is a library for analyzing potential energy surfaces (PESs) and using the resulting PES descriptors for constructing forcefield parameters. Further details are provided in the documentation.

### 1.1 Currently implemented

This package is a work in progress; the following functionalities are currently implemented:

- The MultiMolecule class, a class designed for handling and processing potential energy surfaces. (1)
- A multi-XYZ reader. (2)
- A radial and angular distribution generator (RDF & ADF). (3)
- A root mean squared displacement generator (RMSD). (4)
- A root mean squared fluctuation generator (RMSF). (5)
- Tools for describing shell structures in, *e.g.*, nanocrystals or dissolved solutes. (6)
- A Monte Carlo forcefield parameter optimizer. (7)

### 1.2 Using Auto-FOX

- An input file with some basic examples is provided in the FOX.examples directory.
- An example MD trajectory of a CdSe quantum dot is included in the FOX.data directory.
  - The absolute path + filename of aforementioned trajectory can be retrieved as following:

  ```python
  from FOX import example_xyz
  ```

- Further examples and more detailed descriptions are available in the documentation.
1.3 Installation

1.3.1 Anaconda environments

- While not a strictly required, it strongly recommended to use the virtual environments of Anaconda.
  - Available as either Miniconda or the complete Anaconda package.
- Anaconda comes with a built-in installer; more detailed installation instructions are available for a wide range of OSs.
  - See the Anaconda documentation.
- Anaconda environments can be created, enabled and disabled by, respectively, typing:
  - Create environment: `conda create --name FOX python=3.7`
  - Enable environment: `conda activate FOX`
  - Disable environment: `conda deactivate`

1.3.2 Installing Auto-FOX

- If using Conda, enable the environment: `conda activate FOX`
- Install Auto-FOX with PyPi: `pip install git+https://github.com/nlesc-nano/auto-FOX@master --upgrade`
- Congratulations, Auto-FOX is now installed and ready for use!

1.3.3 Optional dependencies

- Use of the FOX.monte_carlo module requires h5py. Note: h5py is not distributed via PyPi:
  - Anaconda: `conda install --name FOX -y -c conda-forge h5py`
- The plotting of data produced by Auto-FOX requires Matplotlib. Matplotlib is distributed by both PyPi and Anaconda:
  - Anaconda: `conda install --name FOX -y -c conda-forge matplotlib`
  - PyPi: `pip install matplotlib`
- Construction of the angular distribution function in parallel requires DASK.
  - Anaconda: `conda install --name FOX -y -c conda-forge dask`
2.1 Radial & Angular Distribution Function

Radial and angular distribution function (RDF & ADF) generators have been implemented in the `MultiMolecule` class. The radial distribution function, or pair correlation function, describes how the particle density in a system varies as a function of distance from a reference particle. The herein implemented function is designed for constructing RDFs between all possible (user-defined) atom-pairs.

\[
g(r) = \frac{V}{N_a \cdot N_b} \sum_{i=1}^{N_a} \sum_{j=1}^{N_b} \langle \text{placeholder} \rangle
\]

Given a trajectory, `mol`, stored as a `MultiMolecule` instance, the RDF can be calculated with the following command: `rdf = mol.init_rdf(atom_subset=None, low_mem=False)`. The resulting `rdf` is a Pandas dataframe, an object which is effectively a hybrid between a dictionary and a NumPy array.

A slower, but more memory efficient, method of RDF construction can be enabled with `low_mem=True`, causing the script to only store the distance matrix of a single molecule in memory at once. If `low_mem=False`, all distance matrices are stored in memory simultaneously, speeding up the calculation but also introducing an additional linear scaling of memory with respect to the number of molecules. Note: Due to larger size of angle matrices it is recommended to use `low_mem=False` when generating ADFs.

Below is an example RDF and ADF of a CdSe quantum dot pacified with formate ligands. The RDF is printed for all possible combinations of cadmium, selenium and oxygen (Cd_Cd, Cd_Se, Cd_O, Se_Se, Se_O and O_O).

```python
>>> from FOX import MultiMolecule, example_xyz

>>> mol = MultiMolecule.from_xyz(example_xyz)
# Default weight: np.exp(-r)
>>> rdf = mol.init_rdf(atom_subset=('Cd', 'Se', 'O'))
>>> adf = mol.init_adf(r_max=8, weight=None, atom_subset=('Cd', 'Se'))
>>> adf_weighted = mol.init_adf(r_max=8, atom_subset=('Cd', 'Se'))

>>> rdf.plot(title='RDF')
>>> adf.plot(title='ADF')
>>> adf_weighted.plot(title='Distance-weighted ADF')
```
2.1. Radial & Angular Distribution Function
2.1.1 API

`MultiMolecule.init_rdf(mol_subset=None, atom_subset=None, dr=0.05, r_max=12.0, mem_level=2)`

Initialize the calculation of radial distribution functions (RDFs).

RDFs are calculated for all possible atom-pairs in `atom_subset` and returned as a dataframe.

**Parameters**

- **mol_subset (slice)** – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all \( m \) molecules in this instance if `None`.
- **atom_subset (Sequence)** – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all \( n \) atoms per molecule in this instance if `None`.
- **dr (float)** – The integration step-size in Ångström, *i.e.* the distance between concentric spheres.
- **r_max (float)** – The maximum to be evaluated interatomic distance in Ångström.
- **mem_level (int)** – Set the level of to-be consumed memory and, by extension, the execution speed. Given a molecule subset of size \( m \), atom subsets of (up to) size \( n \) and the resulting RDF with \( p \) points (\( p = r_{\text{max}} / \text{dr} \)), the `mem_level` values can be interpreted as following:
  - 0: Slow; memory scaling: \( n^2 \)
  - 1: Medium; memory scaling: \( n^2 + m \cdot p \)
  - 2: Fast; memory scaling: \( n^2 \cdot m \)

**Returns** A dataframe of radial distribution functions, averaged over all conformations in `xyz_array`. Keys are of the form: `at_symbol1 + ' ' + at_symbol2` (*e.g.* "Cd Cd"). Radii are used as index.

**Return type** `pd.DataFrame`

`MultiMolecule.init_adf(mol_subset=None, atom_subset=None, r_max=8.0, weight=<function neg_exp>)`

Initialize the calculation of distance-weighted angular distribution functions (ADFs).

ADFs are calculated for all possible atom-pairs in `atom_subset` and returned as a dataframe.

**Parameters**

- **mol_subset (slice)** – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all \( m \) molecules in this instance if `None`.
- **atom_subset (Sequence)** – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all \( n \) atoms per molecule in this instance if `None`.
- **r_max (float or str)** – The maximum inter-atomic distance (in Angstrom) for which angles are constructed. The distance cutoff can be disabled by settings this value to `np.inf`, "np.inf" or "inf".
- **weight (Callable[[np.ndarray], np.ndarray], optional)** – A callable for creating a weighting factor from inter-atomic distances. The callable should take an array as input and return an array. Given an angle \( \phi_{ijk} \), to the distance \( r_{ijk} \) is defined as \( \text{max}(|r_{ij}, r_{jk}|) \). Set to `None` to disable distance weighting.

**Returns** A dataframe of angular distribution functions, averaged over all conformations in this instance.
**Return type** *pd.DataFrame*

**Note:** Disabling the distance cutoff is strongly recommended (i.e. it is faster) for large values of *r_max*. As a rough guideline, *r_max=*"inf" is roughly as fast as *r_max*=15.0 (though this is, of course, system dependant).

**Note:** The ADF construction will be conducted in parallel if the DASK package is installed. DASK can be installed, via anaconda, with the following command: `conda install -n FOX -y -c conda-forge dask`.

## 2.2 Root Mean Squared Displacement & Fluctuation

### 2.2.1 Root Mean Squared Displacement

The root mean squared displacement (RMSD) represents the average displacement of a set or subset of atoms as a function of time or, equivalently, molecular indices in a MD trajectory.

\[
\rho^{RMSD}(t) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (r_i(t) - r^{ref}_i)^2}
\]

Given a trajectory, `mol`, stored as a `MultiMolecule` instance, the RMSD can be calculated with the `MultiMolecule.init_rmsd()` method using the following command:

```python
>>> rmsd = mol.init_rmsd(atom_subset=None)
```

The resulting `rmsd` is a Pandas dataframe, an object which is effectively a hybrid between a dictionary and a NumPy array.

Below is an example RMSD of a CdSe quantum dot pacified with formate ligands. The RMSD is printed for cadmium, selenium and oxygen atoms.

```python
>>> from FOX import MultiMolecule, example_xyz

>>> mol = MultiMolecule.from_xyz(example_xyz)
>>> rmsd = mol.init_rmsd(atom_subset=('Cd', 'Se', 'O'))
>>> rmsd.plot(title='RMSD')
```

### 2.2.2 Root Mean Squared Fluctuation

The root mean squared fluctuation (RMSF) represents the time-averaged displacement, with respect to the time-averaged position, as a function of atomic indices.

\[
\rho^{RMSF}_i = \sqrt{\langle (r_i - \langle r_i \rangle)^2 \rangle}
\]

Given a trajectory, `mol`, stored as a `MultiMolecule` instance, the RMSF can be calculated with the `MultiMolecule.init_rmsf()` method using the following command:

```python
from FOX import MultiMolecule, example_xyz

mol = MultiMolecule.from_xyz(example_xyz)
 rmsd = mol.init_rmsd(atom_subset=('Cd', 'Se', 'O'))
 rmsd.plot(title='RMSD')
```
>>> rmsd = mol.init_rmsf(atom_subset=None)

The resulting `rmsf` is a Pandas dataframe, an object which is effectively a hybrid between a dictionary and a Numpy array.

Below is an example RMSF of a CdSe quantum dot pacified with formate ligands. The RMSF is printed for cadmium, selenium and oxygen atoms.

```python
>>> from FOX import MultiMolecule, example_xyz

>>> mol = MultiMolecule.from_xyz(example_xyz)

>>> rmsd = mol.init_rmsf(atom_subset=('Cd', 'Se', 'O'))

>>> rmsd.plot(title='RMSF')
```

![RMSF plot](image)
### 2.2.3 Discerning shell structures

See the `MultiMolecule.init_shell_search()` method.

```python
>>> from FOX import MultiMolecule, example_xyz
>>> import matplotlib.pyplot as plt

>>> mol = MultiMolecule.from_xyz(example_xyz)
>>> rmsf, rmsf_idx, rdf = mol.init_shell_search(atom_subset=('Cd', 'Se'))

>>> fig, (ax, ax2) = plt.subplots(ncols=2)
>>> rmsf.plot(ax=ax, title='Modified RMSF')
>>> rdf.plot(ax=ax2, title='Modified RDF')
>>> plt.show()
```

The results above can be utilized for discerning shell structures in, e.g., nanocrystals or dissolved solutes, the RDF minima representing transitions between different shells.

- There are clear minima for Se at ~ 2.0, 5.2, 7.0 & 8.5 Ångstrom
- There are clear minima for Cd at ~ 4.0, 6.0 & 8.2 Ångstrom

With the `MultiMolecule.get_at_idx()` method it is process the results of `MultiMolecule.init_shell_search()`, allowing you to create slices of atomic indices based on aforementioned distance ranges.
It is even possible to use this dictionary with atom names & indices for renaming atoms in a MultiMolecule instance:

```python
>>> print(list(mol.atoms))
['Cd', 'Se', 'C', 'H', 'O']
>>> del mol.atoms['Cd']
>>> del mol.atoms['Se']
>>> mol.atoms.update(idx_dict)
>>> print(list(mol.atoms))
['C', 'H', 'O', 'Se_1', 'Se_2', 'Se_3', 'Se_4', 'Se_5', 'Cd_1', 'Cd_2', 'Cd_3']
```

2.2.4 The atom_subset argument

In the above two examples, `atom_subset=None` was used as an optional keyword, one which allows one to customize for which atoms the RMSD & RMSF should be calculated and how the results are distributed over the various columns.

There are a total of four different approaches to the `atom_subset` argument:

1. `atom_subset=None`: Examine all atoms and store the results in a single column.
2. `atom_subset=int`: Examine a single atom, based on its index, and store the results in a single column.
3. `atom_subset=str` or `atom_subset=list(int)`: Examine multiple atoms, based on their atom type or indices, and store the results in a single column.
4. `atom_subset=tuple(str)` or `atom_subset=tuple(list(int))`: Examine multiple atoms, based on their atom types or indices, and store the results in multiple columns. A column is created for each string or nested list in `atoms`.

It should be noted that lists and/or tuples can be interchanged for any other iterable container (e.g. a Numpy array), as long as the iterables elements can be accessed by their index.
2.2.5 API

MultiMolecule.init_rmsd(mol_subset=None, atom_subset=None, reset_origin=True)
Initialize the RMSD calculation, returning a dataframe.

Parameters

- **mol_subset** (slice) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all \( m \) molecules in this instance if None.
- **atom_subset** (Sequence) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all \( n \) atoms per molecule in this instance if None.
- **reset_origin** (bool) – Reset the origin of each molecule in this instance by means of a partial Procrustes superimposition, translating and rotating the molecules.

Returns A dataframe of RMSDs with one column for every string or list of ints in atom_subset. Keys consist of atomic symbols (e.g. "Cd") if atom_subset contains strings, otherwise a more generic ‘series ‘ + str(int) scheme is adopted (e.g. "series 2"). Molecular indices are used as index.

Return type pd.DataFrame

MultiMolecule.init_rmsf(mol_subset=None, atom_subset=None, reset_origin=True)
Initialize the RMSF calculation, returning a dataframe.

Parameters

- **mol_subset** (slice) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all \( m \) molecules in this instance if None.
- **atom_subset** (Sequence) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all \( n \) atoms per molecule in this instance if None.
- **reset_origin** (bool) – Reset the origin of each molecule in this instance by means of a partial Procrustes superimposition, translating and rotating the molecules.

Returns A dataframe of RMSFs with one column for every string or list of ints in atom_subset. Keys consist of atomic symbols (e.g. "Cd") if atom_subset contains strings, otherwise a more generic ‘series ‘ + str(int) scheme is adopted (e.g. "series 2"). Molecular indices are used as indices.

Return type pd.DataFrame

MultiMolecule.init_shell_search(mol_subset=None, atom_subset=None, rdf_cutoff=0.5)
Calculate and return properties which can help determining shell structures.

The following two properties are calculated and returned:

- The mean distance (per atom) with respect to the center of mass (i.e. a modified RMSF).
- A series mapping arbitrary atomic indices in the RMSF to the actual atomic indices.
- The radial distribution function (RDF) with respect to the center of mass.

Parameters

- **mol_subset** (slice) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all \( m \) molecules in this instance if None.
• **atom_subset** *(Sequence)* – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all \( n \) atoms per molecule in this instance if `None`.

• **rdf_cutoff** *(float)* – Remove all values in the RDF below this value (Angstrom). Useful for dealing with divergence as the “inter-atomic” distance approaches 0.0 Å.

Returns

Returns the following items:

• A dataframe holding the mean distance of all atoms with respect to the center of mass.
• A series mapping the indices from 1. to the actual atomic indices.
• A dataframe holding the RDF with respect to the center of mass.

Return type `pd.DataFrame`, `pd.Series` and `pd.DataFrame`

```python
def get_at_idx(rmsf, idx_series, dist_dict):
    # Create subsets of atomic indices.
    # The subset is created (using `rmsf` and `idx_series`) based on distance criteria in `dist_dict`.
    # For example, `dist_dict = {'Cd': [3.0, 6.5]}` will create and return a dictionary with three keys:
    # One for all atoms whose RMSF is smaller than 3.0, one where the RMSF is between 3.0 and 6.5, and finally one
    # where the RMSF is larger than 6.5.

    # Examples
    >>> dist_dict = {'Cd': [3.0, 6.5]}
    >>> idx_series = pd.Series(np.arange(12))
    >>> rmsf = pd.DataFrame({'Cd': np.arange(12, dtype=float)})
    >>> get_at_idx(rmsf, idx_series, dist_dict)
    {'Cd_1': [0, 1, 2],
     'Cd_2': [3, 4, 5],
     'Cd_3': [7, 8, 9, 10, 11]}
```
2.3 The MultiMolecule Class

The API of the MultiMolecule class.

2.3.1 API FOX.MultiMolecule

class FOX.classes.multi_mol.MultiMolecule(coords: numpy.ndarray, atoms: Optional[Dict[str, List[int]]] = None, bonds: Optional[numpy.ndarray] = None, properties: Optional[Dict[str, Any]] = None)

A class designed for handling and manipulating large numbers of molecules.

More specifically, different conformations of a single molecule as derived from, for example, an intrinsic reaction coordinate calculation (IRC) or a molecular dynamics trajectory (MD). The class has access to four attributes (further details are provided under parameters):

   Parameters
   • coords (m * n * 3 np.ndarray[np.float64]) – A 3D array with the cartesian coordinates of m molecules with n atoms.
   • atoms (dict [str, list [int]]) – A dictionary with atomic symbols as keys and matching atomic indices as values. Stored in the MultiMolecule.atoms attribute.
   • bonds (k * 3 np.ndarray[np.int64]) – A 2D array with indices of the atoms defining all k bonds (columns 1 & 2) and their respective bond orders multiplied by 10 (column 3). Stored in the MultiMolecule.bonds attribute.
   • properties (dict) – A Settings instance for storing miscellaneous user-defined (meta-)data. Is devoid of keys by default. Stored in the MultiMolecule.properties attribute.

atoms: dict [str, list [int]]
A dictionary with atomic symbols as keys and matching atomic indices as values.

bonds: :math:`k*3` |np.ndarray|_ [|np.int64|]_
A 2D array with indices of the atoms defining all k bonds (columns 1 & 2) and their respective bond orders multiplied by 10 (column 3).

properties: |plams.Settings|_
A Settings instance for storing miscellaneous user-defined (meta-)data. Is devoid of keys by default.

round (decimals=0, inplace=True)
Round the Cartesian coordinates of this instance to a given number of decimals.

Parameters
   • decimals (int) – The number of decimals per element.
   • inplace (bool) – Instead of returning the new coordinates, perform an inplace update of this instance.

Return type Optional[MultiMolecule]

delete_atoms (atom_subset)
Create a copy of this instance with all atoms in atom_subset removed.

Parameters atom_subset (Sequence) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all n atoms per molecule in this instance if None.
Returns A new `MultiMolecule` instance with all atoms in `atom_subset` removed.

Return type `FOX.MultiMolecule`

Raises `TypeError` – Raised if `atom_subset` is `None`.

### add_atoms (coords, symbols='Xx')
Create a copy of this instance with all atoms in `atom_subset` appended.

```
>>> import numpy as np
>>> from FOX import MultiMolecule, example_xyz

>>> mol = MultiMolecule.from_xyz(example_xyz)

>>> coords: np.ndarray = np.random.rand(73, 3)  # Add 73 new atoms with random coords
>>> symbols = 'Br'

>>> mol_new: MultiMolecule = mol.add_atoms(coords, symbols)

>>> print(repr(mol))
MultiMolecule(..., shape=(4905, 227, 3), dtype='float64')

>>> print(repr(mol_new))
MultiMolecule(..., shape=(4905, 300, 3), dtype='float64')
```

### Parameters

- **coords** ([array-like]) – A `(3,)`, `(n, 3)`, `(m, 3)` or `(m, n, 3)` array-like object with `m == len(self)`. Represents the Cartesian coordinates of the to-be-added atoms.

- **symbols** ([str or Iterable[str]]) – One or more atomic symbols of the to-be-added atoms.

Returns A new `MultiMolecule` instance with all atoms in `atom_subset` appended.

Return type `FOX.MultiMolecule`

### guess_bonds (atom_subset=None)
Guess bonds within the molecules based on atom type and inter-atomic distances.

Bonds are guessed based on the first molecule in this instance Performs an inplace modification of `self.bonds`

Parameters **atom_subset** ([Sequence]) – A tuple of atomic symbols. Bonds are guessed between all atoms whose atomic symbol is in `atom_subset`. If `None`, guess bonds for all atoms in this instance.

Return type `None`

### random_slice (start=0, stop=None, p=0.5, inplace=False)
Construct a new `MultiMolecule` instance by randomly slicing this instance.

The probability of including a particular element is equivalent to `p`.

Parameters

- **start** ([int]) – Start of the interval.

- **stop** ([int]) – End of the interval.
• **p** *(float)* – The probability of including each particular molecule in this instance. Values must be between 0.0 (0%) and 1.0 (100%).
• **inplace** *(bool)* – Instead of returning the new coordinates, perform an inplace update of this instance.

**Returns** If inplace is True, return a new MultiMolecule instance.
**Return type** None or FOX.MultiMolecule

**Raises** ValueError – Raised if p is smaller than 0.0 or larger than 1.0.

### reset_origin

(reset_origin=\(mol\_subset=None, atom\_subset=None, inplace=True\))

Realign all molecules in this instance.

All molecules in this instance are rotating and translating, by performing a partial partial Procrustes superimposition with respect to the first molecule in this instance.

The superimposition is carried out with respect to the first molecule in this instance.

**Parameters**

• **mol_subset** *(slice)* – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all \(m\) molecules in this instance if None.

• **atom_subset** *(Sequence)* – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all \(n\) atoms per molecule in this instance if None.

• **inplace** *(bool)* – Instead of returning the new coordinates, perform an inplace update of this instance.

**Returns** If inplace is True, return a new MultiMolecule instance.
**Return type** None or FOX.MultiMolecule

### sort

(sort=\(sort\_by='symbol', reverse=False, inplace=True\))

Sort the atoms in this instance and self.atoms, performing in inplace update.

**Parameters**

• **sort_by** *(str or Sequence [int])* – The property which is to be used for sorting. Accepted values: "symbol" (i.e. alphabetical), "atnum", "mass", "radius" or "connectors". See the plams.PeriodicTable module for more details. Alternatively, a user-specified sequence of indices can be provided for sorting.

• **reverse** *(bool)* – Sort in reversed order.

• **inplace** *(bool)* – Instead of returning the new coordinates, perform an inplace update of this instance.

**Returns** If inplace is True, return a new MultiMolecule instance.
**Return type** None or FOX.MultiMolecule

### residue_argsort

(residue_argsort=\(concatenate=True\))

Return the indices that would sort this instance by residue number.

Residues are defined based on molecular fragments based on self.bonds.

**Parameters** **concatenate** *(bool)* – If False, returned a nested list with atomic indices. Each sublist contains the indices of a single residue.

**Returns** A 1D array of indices that would sort \(n\) atoms this instance.
Return type  \( n \) \( np.ndarray \) [np.int64]

\textbf{get_center_of_mass} \((\text{mol\_subset=}\text{None}, \text{atom\_subset=}\text{None})\)
Get the center of mass.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{mol\_subset} \((\text{slice})\) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all \( m \) molecules in this instance if \text{None}.
  \item \texttt{atom\_subset} \((\text{Sequence})\) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all \( n \) atoms per molecule in this instance if \text{None}.
\end{itemize}

\textbf{Returns}  A 2D array with the centres of mass of \( m \) molecules with \( n \) atoms.

\textbf{Return type}  \( m \times 3 \) \( np.ndarray \) [np.float64]

\textbf{get_bonds_per_atom} \((\text{atom\_subset=}\text{None})\)
Get the number of bonds per atom in this instance.

\textbf{Parameters} \texttt{atom\_subset} \((\text{Sequence})\) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all \( n \) atoms per molecule in this instance if \text{None}.

\textbf{Returns}  A 1D array with the number of bonds per atom, for all \( n \) atoms in this instance.

\textbf{Return type}  \( n \) \( np.ndarray \) [np.int64]

\textbf{init_average_velocity} \((\text{timestep=}1.0, \text{rms=}\text{False}, \text{mol\_subset=}\text{None}, \text{atom\_subset=}\text{None})\)
Calculate the average atomic velocity.

The average velocity (in fs/A) is calculated for all atoms in \text{atom\_subset} over the course of a trajectory.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{timestep} \((\text{float})\) – The stepsize, in femtoseconds, between subsequent frames.
  \item \texttt{rms} \((\text{bool})\) – Calculate the root-mean squared average velocity instead.
  \item \texttt{mol\_subset} \((\text{slice})\) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all \( m \) molecules in this instance if \text{None}.
  \item \texttt{atom\_subset} \((\text{Sequence})\) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all \( n \) atoms per molecule in this instance if \text{None}.
\end{itemize}

\textbf{Returns}  A dataframe holding \( m - 1 \) velocities averaged over one or more atom subsets.

\textbf{Return type}  \texttt{pd.DataFrame}

\textbf{init_time_averaged_velocity} \((\text{timestep=}1.0, \text{rms=}\text{False}, \text{mol\_subset=}\text{None}, \text{atom\_subset=}\text{None})\)
Calculate the time-averaged velocity.

The time-averaged velocity (in fs/A) is calculated for all atoms in \text{atom\_subset} over the course of a trajectory.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{timestep} \((\text{float})\) – The stepsize, in femtoseconds, between subsequent frames.
  \item \texttt{rms} \((\text{bool})\) – Calculate the root-mean squared time-averaged velocity instead.
\end{itemize}
• **mol_subset** (*slice*) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all $m$ molecules in this instance if `None`.

• **atom_subset** (*Sequence*) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all $n$ atoms per molecule in this instance if `None`.

**Returns** A dataframe holding $m - 1$ time-averaged velocities.

**Return type** `pd.DataFrame`

`init_rmsd(mol_subset=None, atom_subset=None, reset_origin=True)`

Initialize the RMSD calculation, returning a dataframe.

**Parameters**

• **mol_subset** (*slice*) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all $m$ molecules in this instance if `None`.

• **atom_subset** (*Sequence*) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all $n$ atoms per molecule in this instance if `None`.

• **reset_origin** (*bool*) – Reset the origin of each molecule in this instance by means of a partial Procrustes superimposition, translating and rotating the molecules.

**Returns** A dataframe of RMSDs with one column for every string or list of ints in `atom_subset`. Keys consist of atomic symbols (e.g. "Cd") if `atom_subset` contains strings, otherwise a more generic ‘series’ + str(int) scheme is adopted (e.g. "series 2"). Molecular indices are used as indices.

**Return type** `pd.DataFrame`

`init_rmsf(mol_subset=None, atom_subset=None, reset_origin=True)`

Initialize the RMSF calculation, returning a dataframe.

**Parameters**

• **mol_subset** (*slice*) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all $m$ molecules in this instance if `None`.

• **atom_subset** (*Sequence*) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all $n$ atoms per molecule in this instance if `None`.

• **reset_origin** (*bool*) – Reset the origin of each molecule in this instance by means of a partial Procrustes superimposition, translating and rotating the molecules.

**Returns** A dataframe of RMSFs with one column for every string or list of ints in `atom_subset`. Keys consist of atomic symbols (e.g. "Cd") if `atom_subset` contains strings, otherwise a more generic ‘series’ + str(int) scheme is adopted (e.g. "series 2"). Molecular indices are used as indices.

**Return type** `pd.DataFrame`

`get_average_velocity(timestep=1.0, rms=False, mol_subset=None, atom_subset=None)`

Return the mean or root-mean squared velocity.

**Parameters**

• **timestep** (*float*) – The stepsize, in femtoseconds, between subsequent frames.

2.3. The MultiMolecule Class
• **rms** (*bool*) – Calculate the root-mean squared average velocity instead.

• **mol_subset** (*slice*) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all $m$ molecules in this instance if `None`.

• **atom_subset** (*Sequence*) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all $n$ atoms per molecule in this instance if `None`.

**Returns**  A 1D array holding $m - 1$ velocities averaged over one or more atom subsets.

**Return type**  $m - 1$ `np.ndarray` [`np.float64`]

get_time_averaged_velocity (*timestep=1.0*, *rms=False*, *mol_subset=None*, *atom_subset=None*)

Return the mean or root-mean squared velocity (mean = time-averaged).

**Parameters**

• **timestep** (*float*) – The stepsize, in femtoseconds, between subsequent frames.

• **rms** (*bool*) – Calculate the root-mean squared average velocity instead.

• **mol_subset** (*slice*) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all $m$ molecules in this instance if `None`.

• **atom_subset** (*Sequence*) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all $n$ atoms per molecule in this instance if `None`.

**Returns**  A 1D array holding $n$ time-averaged velocities.

**Return type**  $n$ `np.ndarray` [`np.float64`]

get_velocity (*timestep=1.0*, *norm=True*, *mol_subset=None*, *atom_subset=None*)

Return the atomic velocities.

The velocity (in fs/A) is calculated for all atoms in `atom_subset` over the course of a trajectory.

**Parameters**

• **timestep** (*float*) – The stepsize, in femtoseconds, between subsequent frames.

• **norm** (*bool*) – If `True` return the norm of the $x$, $y$ and $z$ velocity components.

• **mol_subset** (*slice*) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all $m$ molecules in this instance if `None`.

• **atom_subset** (*Sequence*) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all $n$ atoms per molecule in this instance if `None`.

**Returns**  A 2D or 3D array of atomic velocities, the number of dimensions depending on the value of `norm` (`True` = 2D; `False` = 3D).

**Return type**  $m * n$ or $m * n * 3$ `np.ndarray` [`np.float64`]

get_rmsd (*mol_subset=None*, *atom_subset=None*)

Calculate the root mean square displacement (RMSD).

The RMSD is calculated with respect to the first molecule in this instance.

**Parameters**
• **mol_subset** (*slice*) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all $m$ molecules in this instance if None.

• **atom_subset** (*Sequence*) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all $n$ atoms per molecule in this instance if None.

**Returns** A dataframe with the RMSD as a function of the XYZ frame numbers.

**Return type** `pd.DataFrame`

**get_rmsf** (*mol_subset=None, atom_subset=None*)

Calculate the root mean square fluctuation (RMSF).

**Parameters**

• **mol_subset** (*slice*) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all $m$ molecules in this instance if None.

• **atom_subset** (*Sequence*) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all $n$ atoms per molecule in this instance if None.

**Returns** A dataframe with the RMSF as a function of atomic indices.

**Return type** `pd.DataFrame`

**init_shell_search** (*mol_subset=None, atom_subset=None, rdf_cutoff=0.5*)

Calculate and return properties which can help determining shell structures.

The following two properties are calculated and returned:

• The mean distance (per atom) with respect to the center of mass (*i.e.* a modified RMSF).

• A series mapping arbitrary atomic indices in the RMSF to the actual atomic indices.

• The radial distribution function (RDF) with respect to the center of mass.

**Parameters**

• **mol_subset** (*slice*) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all $m$ molecules in this instance if None.

• **atom_subset** (*Sequence*) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all $n$ atoms per molecule in this instance if None.

• **rdf_cutoff** (*float*) – Remove all values in the RDF below this value (Angstrom). Usefull for dealing with divergence as the “inter-atomic” distance approaches 0.0 Å.

**Returns**

Returns the following items:

• A dataframe holding the mean distance of all atoms with respect to the to center of mass.

• A series mapping the indices from 1. to the actual atomic indices.

• A dataframe holding the RDF with respect to the center of mass.

**Return type** `pd.DataFrame, pd.Series` and `pd.DataFrame`
**static get_at_idx** *(rmsf, idx_series, dist_dict)*

Create subsets of atomic indices.

The subset is created (using **rmsf** and **idx_series**) based on distance criteria in **dist_dict**.

For example, `dist_dict = {'Cd': [3.0, 6.5]}` will create and return a dictionary with three keys: One for all atoms whose RMSF is smaller than 3.0, one where the RMSF is between 3.0 and 6.5, and finally one where the RMSF is larger than 6.5.

### Examples

```python
>>> dist_dict = {'Cd': [3.0, 6.5]}
>>> idx_series = pd.Series(np.arange(12))
>>> rmsf = pd.DataFrame({'Cd': np.arange(12, dtype=float)})
>>> get_at_idx(rmsf, idx_series, dist_dict)
{'Cd_1': [0, 1, 2],
 'Cd_2': [3, 4, 5],
 'Cd_3': [7, 8, 9, 10, 11]}
```

### Parameters

- **rmsf** *(pd.DataFrame)* – A dataframe holding the results of an RMSF calculation.
- **idx_series** *(pd.Series)* – A series mapping the indices from **rmsf** to actual atomic indices.
- **dist_dict** *(dict [str, list [float]])* – A dictionary with atomic symbols (see **rmsf.columns**) and a list of interatomic distances.

### Returns

A dictionary with atomic symbols as keys, and matching atomic indices as values.

### Return type

**dict [str, list [int]]**

### Raises

**KeyError** – Raised if a key in **dist_dict** is absent from **rmsf**.

**init_rdf** *(mol_subset=None, atom_subset=None, dr=0.05, r_max=12.0, mem_level=2)*

Initialize the calculation of radial distribution functions (RDFs).

RDFs are calculated for all possible atom-pairs in **atom_subset** and returned as a dataframe.

### Parameters

- **mol_subset** *(slice)* – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all *m* molecules in this instance if None.
- **atom_subset** *(Sequence)* – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all *n* atoms per molecule in this instance if None.
- **dr** *(float)* – The integration step-size in Ångström, *i.e.* the distance between concentric spheres.
- **r_max** *(float)* – The maximum to be evaluated interatomic distance in Ångström.
- **mem_level** *(int)* – Set the level of to-be consumed memory and, by extension, the execution speed. Given a molecule subset of size *m*, atom subsets of (up to) size *n* and the
resulting RDF with \( p \) points (\( p = r_{\text{max}} / \text{dr} \)), the \textbf{mem_level} values can be interpreted as following:

- 0: Slow; memory scaling: \( n^2 \)
- 1: Medium; memory scaling: \( n^2 + m \cdot p \)
- 2: Fast; memory scaling: \( n^2 \cdot m \)

\textbf{Returns} A dataframe of radial distribution functions, averaged over all conformations in \texttt{xyz_array}. Keys are of the form: \texttt{at\_symbol1 + ’ ’ + at\_symbol2} (e.g. "Cd Cu"). Radii are used as index.

\textbf{Return type} \texttt{pd.DataFrame}

g\texttt{et\_dist\_mat}(mol\_subset=None, atom\_subset=None, None)
Create and return a distance matrix for all molecules and atoms in this instance.

\textbf{Parameters}

- \textbf{mol\_subset} (\texttt{slice}) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all \( m \) molecules in this instance if \texttt{None}.

- \textbf{atom\_subset} (\texttt{Sequence}) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all \( n \) atoms per molecule in this instance if \texttt{None}.

\textbf{Returns} A 3D distance matrix of \( m \) molecules, created out of two sets of \( n \) and \( k \) atoms.

\textbf{Return type} \texttt{m \cdot n \cdot k \cdot np.ndarray[np.float64]}

g\texttt{et\_pair\_dict}(atom\_subset, r=2)
Take a subset of atoms and return a dictionary.

\textbf{Parameters}

- \textbf{atom\_subset} (\texttt{Sequence}) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all \( n \) atoms per molecule in this instance if \texttt{None}.

- \textbf{r} (\texttt{int}) – The length of the to-be returned subsets.

\textbf{Return type} \texttt{Dict[\texttt{str}, Tuple[np.ndarray, ...]]}

\texttt{init\_power\_spectrum}(mol\_subset=None, atom\_subset=None, freq\_max=4000)
Calculate and return the power spectrum associated with this instance.

\textbf{Parameters}

- \textbf{mol\_subset} (\texttt{slice}) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all \( m \) molecules in this instance if \texttt{None}.

- \textbf{atom\_subset} (\texttt{Sequence}) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all \( n \) atoms per molecule in this instance if \texttt{None}.

- \textbf{freq\_max} (\texttt{int}) – The maximum to be returned wavenumber (\texttt{cm**-1}).

\textbf{Returns} A DataFrame containing the power spectrum for each set of atoms in \texttt{atom\_subset}.

\textbf{Return type} \texttt{pd.DataFrame}

g\texttt{et\_vacf}(mol\_subset=None, atom\_subset=None)
Calculate and return the velocity autocorrelation function (VACF).
Parameters

- **mol_subset** (*slice*) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all \( m \) molecules in this instance if \( \text{None} \).

- **atom_subset** (*Sequence*) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all \( n \) atoms per molecule in this instance if \( \text{None} \).

Returns A DataFrame containing the power spectrum for each set of atoms in **atom_subset**.

Return type **pd.DataFrame**

**init_adf** *(mol_subset=\text{None}, atom_subset=\text{None}, r_max=8.0, weight=\langle \text{function neg_exp} \rangle)*

Initialize the calculation of distance-weighted angular distribution functions (ADFs).

ADFs are calculated for all possible atom-pairs in **atom_subset** and returned as a dataframe.

Parameters

- **mol_subset** (*slice*) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all \( m \) molecules in this instance if \( \text{None} \).

- **atom_subset** (*Sequence*) – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all \( n \) atoms per molecule in this instance if \( \text{None} \).

- **r_max** (*float* or *str*) – The maximum inter-atomic distance (in Angstrom) for which angles are constructed. The distance cutoff can be disabled by settings this value to \( \text{np.inf} \), "\text{np.inf}" or "\text{inf}".

- **weight** (*Callable[[np.ndarray], np.ndarray], optional*) – A callable for creating a weighting factor from inter-atomic distances. The callable should take an array as input and return an array. Given an angle \( \phi_{ijk} \), to the distance \( r_{ijk} \) is defined as \( \max[r_{ij}, r_{jk}] \). Set to \( \text{None} \) to disable distance weighting.

Returns A dataframe of angular distribution functions, averaged over all conformations in this instance.

Return type **pd.DataFrame**

Note: Disabling the distance cutoff is strongly recommended (i.e. it is faster) for large values of **r_max**. As a rough guideline, \( \text{r_max}="\text{inf}" \) is roughly as fast as \( \text{r_max}=15.0 \) (though this is, of course, system dependant).

Note: The ADF construction will be conducted in parallel if the **DASK** package is installed. **DASK** can be installed, via anaconda, with the following command: `conda install -n FOX -y -c conda-forge dask`.

**get_angle_mat** *(mol_subset=0, atom_subset=\text{None}, \text{None}, \text{None}, get_r_max=\text{False})*

Create and return an angle matrix for all molecules and atoms in this instance.

Parameters

- **mol_subset** (*slice*) – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all \( m \) molecules in this instance if \( \text{None} \).
• **atom_subset** (*Sequence*) — Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all $n$ atoms per molecule in this instance if `None`.

• **get_r_max** (*bool*) — Whether or not the maximum distance should be returned or not.

**Returns** A 4D angle matrix of $m$ molecules, created out of three sets of $n$, $k$ and $l$ atoms. If `get_r_max = True`, also return the maximum distance.

**Return type** $m * n * k * l$ *ndarray* [np.float64] and (optionally) *float*

### as_pdb *(filename, mol_subset=0)*

Convert a `MultiMolecule` object into one or more Protein DataBank files (.pdb).

Utilizes the `plams.Molecule.write` method.

**Parameters**

• **filename** (*str*) — The path+filename (including extension) of the to be created file.

• **mol_subset** (*slice*) — Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all $m$ molecules in this instance if `None`.

**Return type** `None`

### as_mol2 *(filename, mol_subset=0)*

Convert a `MultiMolecule` object into one or more .mol2 files.

Utilizes the `plams.Molecule.write` method.

**Parameters**

• **filename** (*str*) — The path+filename (including extension) of the to be created file.

• **mol_subset** (*slice*) — Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all $m$ molecules in this instance if `None`.

**Return type** `None`

### as_mol *(filename, mol_subset=0)*

Convert a `MultiMolecule` object into one or more .mol files.

Utilizes the `plams.Molecule.write` method.

**Parameters**

• **filename** (*str*) — The path+filename (including extension) of the to be created file.

• **mol_subset** (*slice*) — Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all $m$ molecules in this instance if `None`.

**Return type** `None`

### as_xyz *(filename, mol_subset=None)*

Create an .xyz file out of this instance.

Comments will be constructed by iteration through `MultiMolecule.properties["comments"]` if the following two conditions are fulfilled:

• The "comments" key is actually present in `MultiMolecule.properties`.

• `MultiMolecule.properties["comments"]` is an iterable.
Parameters

- **filename (str)** – The path+filename (including extension) of the to be created file.

- **mol_subset (slice)** – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all m molecules in this instance if None.

Return type None

**as_mass_weighted** *(mol_subset=None, atom_subset=None, inplace=False)*

Transform the Cartesian of this instance into mass-weighted Cartesian coordinates.

Parameters

- **mol_subset (slice)** – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all m molecules in this instance if None.

- **atom_subset (Sequence)** – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all n atoms per molecule in this instance if None.

- **inplace (bool)** – Instead of returning the new coordinates, perform an inplace update of this instance.

Returns if inplace = False return a new MultiMolecule instance with the mass-weighted Cartesian coordinates of m molecules with n atoms.

Return type m * n * 3 np.ndarray [np.float64] or None

**from_mass_weighted** *(mol_subset=None, atom_subset=None)*

Transform this instance from mass-weighted Cartesian into Cartesian coordinates.

Performs an inplace update of this instance.

Parameters

- **mol_subset (slice)** – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all m molecules in this instance if None.

- **atom_subset (Sequence)** – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all n atoms per molecule in this instance if None.

Return type None

**as_Molecule** *(mol_subset=None, atom_subset=None)*

Convert this instance into a list of plams.Molecule.

Parameters

- **mol_subset (slice)** – Perform the calculation on a subset of molecules in this instance, as determined by their molecular index. Include all m molecules in this instance if None.

- **atom_subset (Sequence)** – Perform the calculation on a subset of atoms in this instance, as determined by their atomic index or atomic symbol. Include all n atoms per molecule in this instance if None.

Returns A list of m PLAMS molecules constructed from this instance.

Return type m list [plams.Molecule]
classmethod from_Molecule(mol_list, subset='atoms')
Construct a MultiMolecule instance from one or more PLAMS molecules.

Parameters

• mol_list (plams.Molecule or list [plams.Molecule]) – A PLAMS molecule or list of PLAMS molecules.

• subset (Sequence [str]) – Transfer a subset of plams.Molecule attributes to this instance. If None, transfer all attributes. Accepts one or more of the following values as strings: "properties", "atoms" and/or "bonds".

Returns A MultiMolecule instance constructed from mol_list.

Return type FOX.MultiMolecule

classmethod from_xyz(filename, bonds=None, properties=None)
Construct a MultiMolecule instance from a (multi) .xyz file.
Comment lines extracted from the .xyz file are stored, as array, under MultiMolecule.properties["comments"].

Parameters

• filename (str) – The path+filename of an .xyz file.

• bonds (k * 3 np.ndarray [np.int64]) – An optional 2D array with indices of the atoms defining all k bonds (columns 1 & 2) and their respective bond orders multiplied by 10 (column 3). Stored in the MultiMolecule.bonds attribute.

• properties (dict) – A Settings object (subclass of dictionary) intended for storing miscellaneous user-defined (meta-)data. Is devoid of keys by default. Stored in the MultiMolecule.properties attribute.

Returns A MultiMolecule instance constructed from filename.

Return type FOX.MultiMolecule

classmethod from_kf(filename, bonds=None, properties=None)
Construct a MultiMolecule instance from a KF binary file.

Parameters

• filename (str) – The path+filename of an KF binary file.

• bonds (k * 3 np.ndarray [np.int64]) – An optional 2D array with indices of the atoms defining all k bonds (columns 1 & 2) and their respective bond orders multiplied by 10 (column 3). Stored in the MultiMolecule.bonds attribute.

• properties (dict) – A Settings object (subclass of dictionary) intended for storing miscellaneous user-defined (meta-)data. Is devoid of keys by default. Stored in the MultiMolecule.properties attribute.

Returns A MultiMolecule instance constructed from filename.

Return type FOX.MultiMolecule
2.3.2 API FOX._MultiMolecule

class FOX.classes.multi_mol_magic._MultiMolecule(coords: numpy.ndarray, atoms: Optional[Dict[str, List[int]]] = None, bonds: Optional[numpy.ndarray] = None, properties: Optional[Dict[str, Any]] = None)

Private superclass of MultiMolecule. Handles all magic methods and @property decorated methods.

property loc
A getter and setter for atom-type-based slicing.

Examples

```python
>>> mol = MultiMolecule(...)
>>> mol.atoms['Cd'] = [0, 1, 2, 3, 4, 5]
>>> mol.atoms['Se'] = [6, 7, 8, 9, 10, 11]
>>> mol.atoms['O'] = [12, 13, 14]

>>> (mol.loc['Cd'] == mol[mol.atoms['Cd']]).all()
True

t = mol.atoms['Cd'] + mol.atoms['Se'] + mol.atoms['O']

>>> (mol.loc['Cd', 'Se', 'O'] == mol[t]).all()
True

>>> mol.loc['Cd'] = 1

>>> print((mol.loc['Cd'] == 1).all())
True

>>> del mol.loc['Cd']
ValueError: cannot delete array elements
```

Parameters mol (MultiMolecule) – A MultiMolecule instance; see AtGetter.atoms.

Return type LocGetter

property atom12
Get or set the indices of the atoms for all bonds in MultiMolecule.bonds as 2D array.

Return type _MultiMolecule

property atom1
Get or set the indices of the first atoms in all bonds of MultiMolecule.bonds as 1D array.

Return type _MultiMolecule

property atom2
Get or set the indices of the second atoms in all bonds of MultiMolecule.bonds as 1D array.
property order
Get or set the bond orders for all bonds in MultiMolecule.bonds as 1D array.

Return type ndarray

property x
Get or set the x coordinates for all atoms in instance as 2D array.

Return type _MultiMolecule

property y
Get or set the y coordinates for all atoms in this instance as 2D array.

Return type _MultiMolecule

property z
Get or set the z coordinates for all atoms in this instance as 2D array.

Return type _MultiMolecule

property symbol
Get the atomic symbols of all atoms in MultiMolecule.atoms as 1D array.

Return type ndarray

property atnum
Get the atomic numbers of all atoms in MultiMolecule.atoms as 1D array.

Return type ndarray

property mass
Get the atomic masses of all atoms in MultiMolecule.atoms as 1D array.

Return type ndarray

property radius
Get the atomic radii of all atoms in MultiMolecule.atoms as 1D array.

Return type ndarray

property connectors
Get the atomic connectors of all atoms in MultiMolecule.atoms as 1D array.

Return type ndarray

Parameters

- order (str) – Controls the memory layout of the copy. See np.ndarray.copy for details.
- copy_attr (bool) – Whether or not the attributes of this instance should be returned as copies or views.

Returns
A copy of this instance.

Return type FOX.MultiMolecule
2.4 Addaptive Rate Monte Carlo

The general idea of the MonteCarlo class, and its subclasses, is to fit a classical potential energy surface (PES) to an ab-initio PES by optimizing the classical forcefield parameters. This forcefield optimization is conducted using the Addaptive Rate Monte Carlo (ARMC, 1) method described by S. Cosseddu et al in J. Chem. Theory Comput., 2017, 13, 297–308.

The implemented algorithm can be summarized as following:

2.4.1 The algorithm

1. A trial state, $S_l$, is generated by moving a random parameter retrieved from a user-specified parameter set (e.g. atomic charge).
2. It is checked whether or not the trial state has been previously visited.
   - If True, retrieve the previously calculated PES.
   - If False, calculate a new PES with the generated parameters $S_l$.
     
     $$ p(k \leftarrow l) = \begin{cases} 
     1, & \Delta \epsilon_{QM-MM}(S_k) \Delta \epsilon_{QM-MM}(S_l) \\
     0, & \Delta \epsilon_{QM-MM}(S_k) \Delta \epsilon_{QM-MM}(S_l) 
     \end{cases} \quad (2.1) $$

3. The move is accepted if the new set of parameters, $S_l$, lowers the auxiliary error ($\Delta \epsilon_{QM-MM}$) with respect to the previous set of accepted parameters, $S_k$ (see (2.1)). Given a PES descriptor, $r$, consisting of a matrix with $N$ elements, the auxiliary error is defined in (2.2).

   $$ \Delta \epsilon_{QM-MM} = \frac{\sum_i^N |r_i^{QM} - r_i^{MM}|^2}{\sum_i^N r_i^{QM}} \quad (2.2) $$

4. The parameter history is updated. Based on whether or not the new parameter set is accepted the auxiliary error of either $S_l$ or $S_k$ is increased by the variable $\phi$ (see (2.3)). In this manner, the underlying PES is continuously modified, preventing the optimizer from getting stuck in a (local) minima in the parameter space.

   $$ \Delta \epsilon_{QM-MM}(S_k) + \phi \quad \text{if} \quad \Delta \epsilon_{QM-MM}(S_k) \Delta \epsilon_{QM-MM}(S_l) $$

   $$ \Delta \epsilon_{QM-MM}(S_l) + \phi \quad \text{if} \quad \Delta \epsilon_{QM-MM}(S_k) \Delta \epsilon_{QM-MM}(S_l) \quad (2.3) $$

5. The parameter $\phi$ is updated at regular intervals in order to maintain a constant acceptance rate, $\alpha_t$. This is illustrated in (2.4), where $\phi$ is updated the beginning of every super-iteration $\kappa$. In this example the total number of iterations, $\kappa \omega$, is divided into $\kappa$ super- and $\omega$ sub-iterations.

   $$ \phi_{\kappa \omega} = \phi_{(\kappa-1)\omega} \times \gamma^{\omega(\alpha_t - \pi_{(\kappa-1)})} \quad \kappa = 1, 2, 3, \ldots, N \quad (2.4) $$
## 2.4.2 Arguments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Parameter description</th>
</tr>
</thead>
<tbody>
<tr>
<td>param.prm_file</td>
<td>•</td>
<td>The path+filename of a CHARMM parameter file.</td>
</tr>
<tr>
<td>param.charge</td>
<td>•</td>
<td>A dictionary with atoms and matching atomic charges.</td>
</tr>
<tr>
<td>param.epsilon</td>
<td>•</td>
<td>A dictionary with atom-pairs and the matching Lennard-Jones $\epsilon$ parameter.</td>
</tr>
<tr>
<td>param.sigma</td>
<td>•</td>
<td>A dictionary with atom-pairs and the matching Lennard-Jones $\sigma$ parameter.</td>
</tr>
<tr>
<td>psf.str_file</td>
<td>•</td>
<td>The path+filename to one or more stream file; used for assigning atom types and charges to ligands.</td>
</tr>
<tr>
<td>psf.rtf_file</td>
<td>•</td>
<td>The path+filename to one or more MATCH-produced rtf file; used for assigning atom types and charges to ligands.</td>
</tr>
<tr>
<td>psf.psf_file</td>
<td>•</td>
<td>The path+filename to one or more psf files; used for assigning atom types and charges to ligands.</td>
</tr>
<tr>
<td>psf.ligand_atoms</td>
<td>•</td>
<td>All atoms within a ligand, used for defining residues.</td>
</tr>
<tr>
<td>pes</td>
<td>•</td>
<td>A dictionary holding one or more functions for constructing PES descriptors.</td>
</tr>
<tr>
<td>molecule</td>
<td>•</td>
<td>A list of one or more MultiMolecule instances or .xyz filenames of a reference PES.</td>
</tr>
<tr>
<td>job.logfile</td>
<td>armc.log</td>
<td>The path+filename for the to-be created PLAMs logfile.</td>
</tr>
<tr>
<td>job.job_type</td>
<td>scm.plams.Cp2kJob</td>
<td>The job type, see Job.</td>
</tr>
<tr>
<td>job.name</td>
<td>armc</td>
<td>The base name of the various molecular dynamics jobs.</td>
</tr>
<tr>
<td>job.path</td>
<td>.</td>
<td>The base path for storing the various molecular dynamics jobs.</td>
</tr>
<tr>
<td>job.folder</td>
<td>MM_MD_workdir</td>
<td>The name of the to-be created directory for storing all molecular dynamics jobs.</td>
</tr>
<tr>
<td>job.keepfiles</td>
<td>False</td>
<td>Whether the raw MD results should be saved or deleted.</td>
</tr>
<tr>
<td>job.md_settings</td>
<td>•</td>
<td>A dictionary with the MD job settings. Alternatively, the filename of YAML file can be supplied.</td>
</tr>
<tr>
<td>job.preopt_setting</td>
<td>•</td>
<td>A dictionary of geometry preoptimization job settings. Suplemented by job.md_settings.</td>
</tr>
<tr>
<td>hdf5_file</td>
<td>ARMC.hdf5</td>
<td>The filename of the to-be created HDF5 file with all ARMC results.</td>
</tr>
<tr>
<td>armc.iter_len</td>
<td>50000</td>
<td>The total number of ARMC iterations $\omega$.</td>
</tr>
<tr>
<td>armc.sub_iter_len</td>
<td>100</td>
<td>The length of each ARMC subiteration $\omega$.</td>
</tr>
<tr>
<td>armc.gamma</td>
<td>2.0</td>
<td>The constant $\gamma$, see (2.4).</td>
</tr>
<tr>
<td>armc.a_target</td>
<td>0.25</td>
<td>The target acceptance rate $\alpha_t$, see (2.4).</td>
</tr>
</tbody>
</table>
Once a the .yaml file with the ARMC settings has been sufficiently customized the parameter optimization can be started via the command prompt with: `init_armc my_settings.yaml`.

Previous calculations can be continued with `init_armc my_settings.yaml --restart True`.

### 2.4.3 The pes block

Potential energy surface (PES) descriptors can be described in the "pes" block. Provided below is an example where the radial distribution function (RDF) is used as PES descriptor, more specifically the RDF constructed from all possible combinations of cadmium, selenium and oxygen atoms.

```yaml
pes:
  rdf:
    func: FOX.MultiMolecule.init_rdf
    kwarg:
      atom_subset: [Cd, Se, O]
```

Depending on the system of interest it might be of interest to utilize a PES descriptor other than the RDF, or potentially even multiple PES descriptors. In the latter case the total auxiliary error is defined as the sum of the auxiliary errors of all individual PES descriptors, $R$ (see (2.5)).

\[
\Delta \epsilon_{QM-MM} = \sum_r \Delta \epsilon_r^{QM-MM}
\]  

(2.5)

An example is provided below where both radial and angular distribution functions (RDF and ADF, respectively) are used as PES descriptors. In this example the RDF is constructed for all combinations of cadmium, selenium and oxygen atoms (Cd, Se & O), whereas the ADF is constructed for all combinations of cadmium and selenium atoms (Cd & Se).

```yaml
pes:
  rdf:
    func: FOX.MultiMolecule.init_rdf
    args: []
    kwarg:
      atom_subset: [Cd, Se, O]
  adf:
    func: FOX.MultiMolecule.init_adf
    args: []
    kwarg:
      atom_subset: [Cd, Se]
```

In principle any function, class or method can be provided here, as type object, as long as the following requirements are fulfilled:

- The name of the block must consist of a user-specified string ("rdf" and "adf" in the example(s) above).
- The "func" key must contain a string representation of the requested function, method or class. Auto-FOX will internally convert the string into a callable object.
- The supplied callable must be able to operate on NumPy arrays or instances of its `MultiMolecule` subclass.
- Arguments and keyword argument can be provided with the "args" and "kwarg" keys, respectively. The "args" and "kwarg" keys are entirely optional and can be skipped if desired.

An example of a custom, albeit rather nonsensical, PES descriptor involving the `numpy.sum` function is provided below:
pes:
  numpy_sum:
    func: numpy.sum
    kwargs:
      axis: 0

This .yaml input, given a `MultiMolecule` instance mol, is equivalent to:

```python
>>> import numpy
>>> func = numpy.sum
>>> args = []
>>> kwargs = {'axis': 0}
>>> func(mol, *arg, **kwarg)
```

### 2.4.4 The param block

```
param:
  charge:
    keys: [input, force_eval, mm, forcefield, charge]
    constraints:
      - 0 < Cs < 2
      - 1 < Pb < 3
      - Cs == 0.5 * Br
    Cs: 1.000
    Pb: 2.000
  epsilon:
    unit: kjmol
    keys: [input, force_eval, mm, forcefield, nonbonded, lennard-jones]
    Cs Cs: 0.1882
    Cs Pb: 0.7227
    Pb Pb: 2.7740
  sigma:
    unit: nm
    keys: [input, force_eval, mm, forcefield, nonbonded, lennard-jones]
    constraints: 'Cs Cs == Pb Pb'
    Cs Cs: 0.60
    Cs Pb: 0.50
    Pb Pb: 0.60
```

The "param" key in the .yaml input contains all user-specified to-be optimized parameters.

There are three critical (and two optional) components to the "param" block:

- The key of each block (charge, epsilon & sigma).
- The "keys" sub-block, which points to the section path in the CP2K settings (e.g. ['input', 'force_eval', 'mm', 'forcefield', 'charge']).
- The sub-blocks containing either singular atoms or atom pairs.

Together, these three components point to the appropriate path of the forcefield parameter(s) of interest. As of the moment, all bonded and non-bonded potentials implemented in CP2K can be accessed via this section of the input file. For example, the following input is suitable if one wants to optimize a torsion potential (starting from $k = 10 \text{ kcal/mol}$) for all C-C-C-C bonds:
Besides the three above-mentioned mandatory components, one can (optionally) supply the unit of the parameter and/or constrain its value to a certain range. When supplying units, it is the responsibility of the user to ensure the units are supported by CP2K. Furthermore, parameter constraints are, as of the moment, limited to specifying minimum and/or maximum values (e.g. \(0 < C_s < 2\)). Additional (more elaborate) constraints are currently already available for atomic charges in the move.charge_constraints block (see below).

### 2.4.5 Parameter Guessing

```plaintext
param:
  epsilon:
    unit: kjmol
    keys: [input, force_eval, mm, forcefield, nonbonded, lennard-jones]
    Cs Cs: 0.1882
    Cs Pb: 0.7227
    Pb Pb: 2.7740
    guess: rdf
  sigma:
    unit: nm
    keys: [input, force_eval, mm, forcefield, nonbonded, lennard-jones]
  frozen:
    guess: uff

\[ V_{LJ} = 4\varepsilon \left( \frac{\sigma}{r} \right)^{12} - \left( \frac{\sigma}{r} \right)^{6} \]
```

Non-bonded interactions (i.e. the Lennard-Jones \(\varepsilon\) and \(\sigma\) values) can be guessed if they’re not explicitly by the user. There are currently two implemented guessing procedures: "uff" and "rdf". Parameter guessing for parameters other than \(\varepsilon\) and \(\sigma\) is not supported as of the moment.

The "uff" approach simply takes all missing parameters from the Universal Force Field (UFF)[2]. Pair-wise parameters are constructed using the standard combinatorial rules: the arithmetic mean for \(\sigma\) and the geometric mean for \(\varepsilon\).

The "rdf" approach utilizes the radial distribution function for estimating \(\sigma\) and \(\varepsilon\). \(\sigma\) is taken as the base of the first RDF peak, while the first minimum of the Boltzmann-inverted RDF is taken as \(\varepsilon\). "crystal_radius" and "ion_radius" use a similar approach to "uff", the key difference being the origin of the parameters: 10.1107/S0567739476001551: R. D. Shannon, Revised effective ionic radii and systematic studies of interatomic distances in halides and chalcogenides, Acta Cryst. (1976). A32, 751-767. Note that:

- Values are averaged with respect to all charges and coordination numbers per atom type.
- These two guess-types can only be used for estimating \(\sigma\) parameters.

If "guess" is placed within the "frozen" block, than the guessed parameters will be treated as constants rather than to-be optimized variables.

**Note**

The guessing procedure requires the presence of both a .prm and .psf file. See the "prm_file" and "psf" blocks, respectively.
2.4.6 State-averaged ARMC

molecule:
- /path/to/md_acetate.xyz
- /path/to/md_phosphate.xyz
- /path/to/md_sulfate.xyz

psf:
  rtf_file:
  - acetate.rtf
  - phosphate.rtf
  - sulfate.rtf
ligand_atoms: [S, P, O, C, H]

pes:
rdf:
  func: FOX.MultiMolecule.init_rdf
  kwargs:
  - atom_subset: [Cd, Se, O]
  - atom_subset: [Cd, Se, P, O]
  - atom_subset: [Cd, Se, S, O]

2.4.7 FOX.MonteCarlo API

```python
class FOX.classes.monte_carlo.MonteCarlo(molecule, param, md_settings, preopt_settings=None, rmsd_threshold=5.0, job_type=<class 'scm.plams.interfaces.thirdparty.cp2k.Cp2kJob'>, hdf5_file='ARMC.hdf5', apply_move=<ufunc 'multiply'>, move_range=None, keep_files=False, logger=None, pes_post_process=None)
```

The base `MonteCarlo` class.

**property job_name**
Get the (lowered) name of `MonteCarlo.job_type`.

Return type: `str`

**property molecule**
Get `value` or set `value` as a tuple of `MultiMolecule` instances.

Return type: `Tuple[MultiMolecule,...]`

**property md_settings**
Get `value` or set `value` as a `plams.Settings` instance.

Return type: `Tuple[Settings,...]`

**property preopt_settings**
Get `value` or set `value` as a `plams.Settings` instance.

Return type: `Tuple[Settings,...]`
property move_range
Get value or set value as a np.ndarray.

Return type ndarray

property logger
Get or set the logger.

Return type Logger

keys()
Return a view of MonteCarlo.history_dict’s keys.

Return type KeysView

items()
Return a view of MonteCarlo.history_dict’s items.

Return type ItemsView

values()
Return a view of MonteCarlo.history_dict’s values.

Return type ValuesView

add_pes_evaluator (name, func, args=(), kwargs=mappingproxy({}))
Add a callable to this instance for constructing PES-descriptors.

Examples

```python
>>> from FOX import MonteCarlo, MultiMolecule

>>> mc = MonteCarlo(...)  
>>> mol = MultiMolecule.from_xyz(...)  

# Prepare arguments
>>> name = 'rdf'  
>>> func = FOX.MultiMolecule.init_rdf  
>>> atom_subset = ['Cd', 'Se', 'O']  # Keyword argument for func

# Add the PES-descriptor constructor
>>> mc.add_pes_evaluator(name, func, kwargs={'atom_subset': atom_subset})
```

Parameters

- **name (str)** – The name under which the PES-descriptor will be stored (e.g. "RDF").
- **func (Callable)** – The callable for constructing the PES-descriptor. The callable should take an array-like object as input and return a new array-like object as output.
- **args (Sequence)** – A sequence of positional arguments.
- **kwargs (dict or Iterable[dict])** – A dictionary or an iterable of dictionaries with keyword arguments. Providing an iterable allows one to use a unique set of keyword arguments for each molecule in MonteCarlo.molecule.

Return type None

move()
Update a random parameter in self.param by a random value from self.move.range.
Performs in inplace update of the 'param' column in self.param. By default the move is applied in a multiplicative manner. self.job.md_settings and self.job.preopt_settings are updated to reflect the change in parameters.

Examples

```python
>>> print(armc.param['param'])
charge  Br  -0.731687
       Cs   0.731687
epsilon Br Br  1.045000
       Cs Br  0.437800
       Cs Cs  0.300000
sigma Br Br  0.421190
       Cs Br  0.369909
       Cs Cs  0.592590
Name: param, dtype: float64

>>> for _ in range(1000):  # Perform 1000 random moves
    armc.move()

>>> print(armc.param['param'])
charge  Br  -0.597709
       Cs   0.444592
epsilon Br Br  0.653053
       Cs Br  1.088848
       Cs Cs  1.025769
sigma Br Br  0.339293
       Cs Br  0.136361
       Cs Cs  0.101097
Name: param, dtype: float64
```

Returns A tuple with the (new) values in the 'param' column of self.param.

Return type tuple [float]

clip_move(idx, value)
Ensure that value falls within a user-specified range.

Return type float

run_md()
Run a geometry optimization followed by a molecular dynamics (MD) job.

Returns a new MultiMolecule instance constructed from the MD trajectory and the path to the MD results. If no trajectory is available (i.e. the job crashed) return None instead.

- The MD job is constructed according to the provided settings in self.job.

Returns A list of MultiMolecule instance(s) constructed from the MD trajectory & a list of paths to the PLAMS results directories. The MultiMolecule list is replaced with None if the job crashes.

Return type FOX.MultiMolecule and tuple [str]

clear_job_cache()
Clear MonteCarlo.job_cache and, optionally, delete all cp2k output files.

Return type None
**get_pes_descriptors** (key, get_first_key=False)
Check if a key is already present in **history_dict**.
If True, return the matching list of PES descriptors; If False, construct and return a new list of PES descriptors.
• The PES descriptors are constructed by the provided settings in **self.pes**.

**Parameters**
• **key** (tuple [float]) – A key in **history_dict**.
• **get_first_key** (bool) – Keep the both the files and the job_cache if this is the first ARMC iteration. Usefull for manual inspection in case cp2k hard-crashes at this point.

**Returns** A previous value from **history_dict** or a new value from an MD calculation & a MultiMolecule instance constructed from the MD simulation. Values are set to np.inf if the MD job crashed.

**Return type** dict [str, np.ndarray [np.float64]] and FOX.MultiMolecule

---

### 2.4.8 FOX.ARMC API

**class** FOX.classes.armc.ARMC (iter_len=50000, sub_iter_len=100, gamma=200, a_target=0.25, phi=1.0, apply_phi=<ufunc 'add'>, **kwargs)

The Addaptive Rate Monte Carlo class (**ARMC**).
A subclass of MonteCarlo.

**Parameters**
• **iter_len** (int) – The total number of ARMC iterations \( k \omega \).
• **sub_iter_len** (int) – The length of each ARMC subiteration \( \omega \).
• **gamma** (float) – The constant \( \gamma \).
• **a_target** (float) – The target acceptance rate \( \alpha_t \).
• **phi** (float) – The variable \( \phi \).
• **apply_phi** (*Callable*) – The callable used for applying \( \phi \) to the auxiliary error. The callable should be able to take 2 floats as argument and return a new float.
• ****kwargs (|Any|_ – Keyword arguments for the MonteCarlo superclass.

**property** super_iter_len
Get ARMC.iter_len // ARMC.sub_iter_len.

**Return type** int

**classmethod** from_yaml (filename)
Create a **ARMC** instance from a .yaml file.

**Parameters** filename (str) – The path+filename of a .yaml file containing all **ARMC** settings.

**Returns** A new **ARMC** instance and a dictionary with keyword arguments for run_armc().

**Return type** FOX.ARMC and dict

**to_yaml** (filename, logfile=None, path=None, folder=None)
Convert an **ARMC** instance into a .yaml readable by **ARMC.from_yaml**.
Parameters `filename (str, bytes, os.pathlike or io.IOBase) – A filename or a
file-like object.

Return type None

`do_inner (kappa, omega, acceptance, key_old)

Run the inner loop of the ARMC.__call__() method.

Parameters

• kappa (int) – The super-iteration, \( \kappa \), in ARMC.__call__().
• omega (int) – The sub-iteration, \( \omega \), in ARMC.__call__().
• history_dict (dict [tuple [float], np.ndarray [np.float64]]) – A dictionary with pa-
rameters as keys and a list of PES descriptors as values.
• key_new (tuple [float]) – A tuple with the latest set of forcefield parameters.

Returns The latest set of parameters.

Return type tuple [float]

`get_aux_error (pes_dict)

Return the auxiliary error \( \Delta \epsilon_{QM-MM} \).

The auxiliary error is constructed using the PES descriptors in values with respect to self.ref.

The default function is equivalent to:

\[
\Delta \epsilon_{QM-MM} = \sum_i \frac{|p_i^{QM} - r_i^{MM}|^2}{\psi_i^{QM}}
\]

Parameters `pes_dict ([dict [str, np.ndarray [np.float64]]) – An dictionary with \( m \ast n \) PES
descriptors each.

Returns An array with \( m \ast n \) auxiliary errors

Return type \( m \ast n \) np.ndarray [np.float64]

`update_phi (acceptance)

Update the variable \( \phi \).

\( \phi \) is updated based on the target acceptance rate, \( \alpha_t \), and the acceptance rate, acceptance, of the current
super-iteration.

• The values are updated according to the provided settings in self.armc.

The default function is equivalent to:

\[
\phi_{\kappa\omega} = \phi_{(\kappa-1)\omega} + \gamma \text{sgn}(\alpha_t - \tau(\kappa-1))
\]

Parameters `acceptance (np.ndarray [bool]) – A 1D boolean array denoting the accepted
moves within a sub-iteration.

Returns None

`restart ()

Restart a previously started Addaptive Rate Monte Carlo procedure.

Return type None
2.5 Multi-XYZ reader

A reader of multi-xyz files has been implemented in the FOX.io.read_xyz module. The .xyz file format is designed for storing the atomic symbols and cartesian coordinates of one or more molecules. The herein implemented FOX.io.read_xyz.read_multi_xyz() function allows for the fast, and memory-efficient, retrieval of the various molecular geometries stored in an .xyz file.

An .xyz file, example_xyz_file, can also be directly converted into a MultiMolecule instance.

```python
>>> from FOX import MultiMolecule, example_xyz

>>> mol = MultiMolecule.from_xyz(example_xyz)

>>> print(type(mol))
<class 'FOX.classes.multi_mol.MultiMolecule'>
```

2.5.1 API

FOX.io.read_xyz.read_multi_xyz(filename, return_comment=True)

Read a (multi).xyz file.

Parameters

• filename (str) – The path+filename of a (multi).xyz file.

• return_comment (bool) – Whether or not the comment line in each Cartesian coordinate block should be returned. Returned as a 1D array of strings.

Return type

Union[Tuple[ndarray, Dict[str, List[int]]], Tuple[ndarray, Dict[str, List[int]], ndarray]]

Returns

• \( m \times n \times 3 \) ndarray[float64], dict[str, list[int]] and

• (optional) \( m \) ndarray[ndarray[...]] –

  • A 3D array with Cartesian coordinates of \( m \) molecules with \( n \) atoms.

  • A dictionary with atomic symbols as keys and lists of matching atomic indices as values.

• (Optional) a 1D array with \( m \) comments.

Raises XYZError – Raised when issues are encountered related to parsing .xyz files.

classmethod MultiMolecule.from_xyz(filename, bonds=None, properties=None)

Construct a MultiMolecule instance from a (multi).xyz file.

Comment lines extracted from the .xyz file are stored, as array, under MultiMolecule.properties["comments"].

Parameters

• filename (str) – The path+filename of an .xyz file.

• bonds (k * 3 np.ndarray[...]) – An optional 2D array with indices of the atoms defining all \( k \) bonds (columns 1 & 2) and their respective bond orders multiplied by 10 (column 3). Stored in the MultiMolecule.bonds attribute.

• properties (dict) – A Settings object (subclass of dictionary) intended for storing miscellaneous user-defined (meta-)data. Is devoid of keys by default. Stored in the MultiMolecule.properties attribute.
Returns  A `MultiMolecule` instance constructed from `filename`.

Return type  `FOX.MultiMolecule`

```python
FOX.example_xyz:  str = '/home/docs/checkouts/readthedocs.org/user_builds/auto-fox/checkouts/latest/FOX/data/Cd68Se55_26COO_MD_trajec.xyz'
```

The path+filename of the example multi-xyz file.

## 2.6 FOX.ff.lj_param

A module for estimating Lennard-Jones parameters.

### Examples

```python
>>> import pandas as pd
>>> from FOX import MultiMolecule, example_xyz, estimate_lennard_jones

>>> xyz_file:  str = example_xyz
>>> atom_subset = ['Cd', 'Se', 'O']

>>> mol = MultiMolecule.from_xyz(xyz_file)
>>> rdf: pd.DataFrame = mol.init_rdf(atom_subset=atom_subset)
>>> param: pd.DataFrame = estimate_lennard_jones(rdf)

>>> print(param)

<table>
<thead>
<tr>
<th>Atom pairs</th>
<th>sigma (Angstrom)</th>
<th>epsilon (kj/mol)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cd Cd</td>
<td>3.95</td>
<td>2.097554</td>
</tr>
<tr>
<td>Cd Se</td>
<td>2.50</td>
<td>4.759017</td>
</tr>
<tr>
<td>Cd O</td>
<td>2.20</td>
<td>3.360966</td>
</tr>
<tr>
<td>Se Se</td>
<td>4.20</td>
<td>2.976106</td>
</tr>
<tr>
<td>Se O</td>
<td>3.65</td>
<td>0.992538</td>
</tr>
<tr>
<td>O O</td>
<td>2.15</td>
<td>6.676584</td>
</tr>
</tbody>
</table>
```

### 2.6.1 Index

- `estimate_lj(rdf[, temperature, sigma_estimate])`  Estimate the Lennard-Jones $\sigma$ and $\varepsilon$ parameters using an RDF.
- `get_free_energy(distribution[, temperature, ...])`  Convert a distribution function into a free energy function.

### 2.6.2 API

`FOX.ff.lj_param.estimate_lj(rdf, temperature=298.15, sigma_estimate='base')`  Estimate the Lennard-Jones $\sigma$ and $\varepsilon$ parameters using an RDF.

Given a radius $r$, the Lennard-Jones potential $V_{LJ}(r)$ is defined as following:

$$V_{LJ}(r) = 4\varepsilon \left( \frac{\sigma}{r} \right)^{12} - \left( \frac{\sigma}{r} \right)^{6}$$

The $\sigma$ and $\varepsilon$ parameters are estimated as following:
• \(\sigma\): The radii at which the first inflection point or peak base occurs in \texttt{rdf}.

• \(\varepsilon\): The minimum value in of the \texttt{rdf} ree energy multiplied by \(-1\).

• All values are calculated per atom pair specified in \texttt{rdf}.

Parameters

• \texttt{rdf} (\texttt{pandas.DataFrame}) – A radial distribution function. The columns should consist of atom-pairs.

• \texttt{temperature} (\texttt{float}) – The temperature in Kelvin.

• \texttt{sigma\_estimate} (\texttt{str}) – Whether \(\sigma\) should be estimated based on the base of the first peak or its inflection point. Accepted values are "base" and "inflection", respectively.

Returns A Pandas DataFrame with two columns, "sigma" (Angstrom) and "epsilon" (kcal/mol), holding the Lennard-Jones parameters. Atom-pairs from \texttt{rdf} are used as index.

Return type \texttt{pandas.DataFrame}

See also:

\texttt{MultiMolecule.init\_rdf()} Initialize the calculation of radial distribution functions (RDFs).

\texttt{get\_free\_energy()} Convert a distribution function into a free energy function.

FOX.ff.lj_param.get\_free\_energy(distribution, \texttt{temperature}=298.15, \texttt{unit}='kcal/mol', \texttt{inf\_replace}=\texttt{nan})

Convert a distribution function into a free energy function.

Given a distribution function \(g(r)\), the free energy \(F(g(r))\) can be retrieved using a Boltzmann inversion:

\[
F(g(r)) = -RT \ln(g(r))
\]

Two examples of valid distribution functions would be the radial- and angular distribution functions.

Parameters

• \texttt{distribution} (\texttt{array-like}) – A distribution function (\texttt{e.g.} an RDF) as an array-like object.

• \texttt{temperature} (\texttt{float}) – The temperature in Kelvin.

• \texttt{inf\_replace} (\texttt{float}, optional) – A value used for replacing all instances of infinity (\texttt{np.inf}).

• \texttt{unit} (\texttt{str}) – The to-be returned unit. See \texttt{scm.plams.Units} for a comprehensive overview of all allowed values.

Returns An array-like object with a free-energy function (kJ/mol) of \texttt{distribution}.

Return type \texttt{pandas.DataFrame}

See also:

\texttt{MultiMolecule.init\_rdf()} Initialize the calculation of radial distribution functions (RDFs).

\texttt{MultiMolecule.init\_adf()} Initialize the calculation of distance-weighted angular distribution functions (ADFs).
2.7 PSFContainer

2.7.1 FOX.io.read_psf

A class for reading protein structure (.psf) files.

Index

PSFContainer([filename, title, atoms, ...])  A container for managing protein structure files.

API

class FOX.io.read_psf.PSFContainer (filename=None, title=None, atoms=None, bonds=None, angles=None, dihedrals=None, impropers=None, donors=None, acceptors=None, no_nonbonded=None)

A container for managing protein structure files.

The PSFContainer class has access to three general sets of methods.

Methods for reading & constructing .psf files:

• PSFContainer.read()
• PSFContainer.write()

Methods for updating atom types:

• PSFContainer.update_atom_charge()
• PSFContainer.update_atom_type()

Methods for extracting bond, angle and dihedral-pairs from plams.Molecule instances:

• PSFContainer.generate_bonds()
• PSFContainer.generate_angles()
• PSFContainer.generate_dihedrals()
• PSFContainer.generate_impropers()
• PSFContainer.generate_atoms()

Parameters

• filename (1 numpy.ndarray [str]) – Optional: A 1D array-like object containing a single filename. See also PSFContainer.filename.
• title (n numpy.ndarray [str]) – Optional: A 1D array of strings holding the title block. See also PSFContainer.title.
• atoms (n * 8 pandas.DataFrame) – Optional: A Pandas DataFrame holding the atoms block. See also PSFContainer.atoms.
• bonds (n * 2 numpy.ndarray [int]) – Optional: A 2D array-like object holding the indices of all atom-pairs defining bonds. See also PSFContainer.bonds.
• angles (n * 3 numpy.ndarray [int]) – Optional: A 2D array-like object holding the indices of all atom-triplets defining angles. See also PSFContainer.angles.
• **dihedrals** (*n* x 4 `numpy.ndarray` [int]) – Optional: A 2D array-like object holding the indices of all atom-quartets defining proper dihedral angles. See also `PSFContainer.dihedrals`.

• **impropers** (*n* x 4 `numpy.ndarray` [int]) – Optional: A 2D array-like object holding the indices of all atom-quartets defining improper dihedral angles. See also `PSFContainer.impropers`.

• **donors** (*n* x 1 `numpy.ndarray` [int]) – Optional: A 2D array-like object holding the atomic indices of all hydrogen-bond donors. See also `PSFContainer.donors`.

• **acceptors** (*n* x 1 `numpy.ndarray` [int]) – Optional: A 2D array-like object holding the atomic indices of all hydrogen-bond acceptors. See also `PSFContainer.acceptors`.

• **no_nonbonded** (*n* x 2 `numpy.ndarray` [int]) – Optional: A 2D array-like object holding the indices of all atom-pairs whose nonbonded interactions should be ignored. See also `PSFContainer.no_nonbonded`.

**filename**: :math:`1` :class:`numpy.ndarray`[:class:`str`]

A 1D array with a single string as filename.

**title**: :math:`n` :class:`numpy.ndarray`[:class:`str`]

A 1D array of strings holding the title block.

**atoms**: :math:`n` x 8 :class:`pandas.DataFrame`

A Pandas DataFrame holding the atoms block. The DataFrame should possess the following column keys:

• "segment name"
• "residue ID"
• "residue name"
• "atom name"
• "atom type"
• "charge"
• "mass"
• "0"

**bonds**: :math:`n` x 2 :class:`numpy.ndarray`[:class:`int`]

A 2D array holding the indices of all atom-pairs defining bonds. Indices are expected to be 1-based.

**angles**: :math:`n` x 3 :class:`numpy.ndarray`[:class:`int`]

A 2D array holding the indices of all atom-triplets defining angles. Indices are expected to be 1-based.

**dihedrals**: :math:`n` x 4 :class:`numpy.ndarray`[:class:`int`]

A 2D array holding the indices of all atom-quartets defining proper dihedral angles. Indices are expected to be 1-based.

**impropers**: :math:`n` x 4 :class:`numpy.ndarray`[:class:`int`]

A 2D array holding the indices of all atom-quartets defining improper dihedral angles. Indices are expected to be 1-based.

**donors**: :math:`n` x 1 :class:`numpy.ndarray`[int]

A 2D array holding the atomic indices of all hydrogen-bond donors. Indices are expected to be 1-based.

**acceptors**: :math:`n` x 1 :class:`numpy.ndarray`[int]

A 2D array holding the atomic indices of all hydrogen-bond acceptors. Indices are expected to be 1-based.
no_nonbonded: :math:`n * 2` :class:`numpy.ndarray` [:class:`int`]
A 2D array holding the indices of all atom-pairs whose nonbonded interactions should be ignored. Indices are expected to be 1-based.

A mapping with Numpy print options. See np.set_printoptions.

A mapping with Pandas print options. See Options and settings.

PRIVATE_ATTR: Set[str] = frozenset({'_np_printoptions', '_pd_printoptions'})
A set with the names of private instance attributes. These attributes will be excluded whenever calling PSF.as_dict().

_SHAPE_DICT = mappingproxy({'filename': {'shape': 1}, 'title': {'shape': 1}, 'atoms': {'shape': 8}, 'bonds': {'shape': 2, ... 8, 'header': '{:>10d} !NACC: acceptors'}, 'no_nonbonded': {'shape': 2, 'row_len': 4, 'header': '{:>10d} !NNB'}})
A dictionary containing array shapes among other things

_HEADER_DICT: Mapping[str, str] = mappingproxy({ '!NTITLE': 'title', '!NATOM': 'atoms', '!NBOND': 'bonds', '!NTHETA': 'angles', '!NPHI': 'dihedrals', '!NIMPHI': 'impropers', '!NDON': 'donors', '!NACC': 'acceptors', '!NNB': 'no_nonbonded'})
A dictionary mapping .psf headers to PSFContainer attribute names

__init__ (filename=None, title=None, atoms=None, bonds=None, angles=None, dihedrals=None, impropers=None, donors=None, acceptors=None, no_nonbonded=None)
Initialize a PSFContainer instance.

Return type None

static __is_dict (value)
Check if value is a dict instance; raise a TypeError if not.

Return type dict

__repr__() 
Return a (machine readable) string representation of this instance.

The string representation consists of this instances’ class name in addition to all (non-private) instance variables.

Returns A string representation of this instance.

Return type str

See also:

PSFContainer._PRIVATE_ATTR A set with the names of private instance variables.

PSFContainer._repr_fallback Fallback function for PSFContainer.__repr__() in case of recursive calls.

PSFContainer._str_iterator() Return an iterable for the iterating over this instances’ attributes.

PSFContainer._str() Returns a string representation of a single key/value pair.

__str_iterator__ ()
Return an iterable for the PSFContainer.__repr__() method.

Return type Iterable[Tuple[str, Any]]

__eq__(value)
Check if this instance is equivalent to value.

The comparison checks if the class type of this instance and value are identical and if all (non-private) instance variables are equivalent.

Returns Whether or not this instance and value are equivalent.
Return type bool

See also:

PSFContainer._PRIVATE_ATTR A set with the names of private instance variables.

PSFContainer._eq Return if v1 and v2 are equivalent.

PSFContainer._eqFallback Fallback function for PSFContainer._eq() incase of recursive calls.

as_dict (return_private=False)
Construct a dictionary from this instance with all non-private instance variables.
The returned dictionary values are shallow copies.

Parameters return_private (bool) – If True, return both public and private instance variables. Private instance variables are defined in PSFContainer._PRIVATE_ATTR.

Returns A dictionary with keyword arguments for initializing a new instance of this class.

Return type dict [str, Any]

See also:

PSFContainer.from_dict() Construct a instance of this objects’ class from a dictionary with keyword arguments.

PSFContainer._PRIVATE_ATTR A set with the names of private instance variables.

copy (deep=True)
Return a shallow or deep copy of this instance.

Parameters deep (bool) – Whether or not to return a deep or shallow copy.

Returns A new instance constructed from this instance.

Return type PSFContainer

__copy__()
Return a shallow copy of this instance; see PSFContainer.copy().

Return type ~AT

property filename
Get PSFContainer.filename as string or assign an array-like object as a 1D array.

Return type str

property title
Get PSFContainer.title or assign an array-like object as a 1D array.

Return type ndarray

property atoms
Get PSFContainer.atoms or assign an a DataFrame.

Return type DataFrame

property bonds
Get PSFContainer.bonds or assign an array-like object as a 2D array.

Return type ndarray
property angles
    Get \texttt{PSFContainer.\texttt{angles}} or assign an array-like object as a 2D array.
    Return type \texttt{ndarray}

property dihedrals
    Get \texttt{PSFContainer.\texttt{dihedrals}} or assign an array-like object as a 2D array.
    Return type \texttt{ndarray}

property impropers
    Get \texttt{PSFPSFContainer.impropers} or assign an array-like object as a 2D array.
    Return type \texttt{ndarray}

property donors
    Get \texttt{PSFContainer.donors} or assign an array-like object as a 2D array.
    Return type \texttt{ndarray}

property acceptors
    Get \texttt{PSFContainer.acceptors} or assign an array-like object as a 2D array.
    Return type \texttt{ndarray}

property no_nonbonded
    Get \texttt{PSFContainer.no\_nonbonded} or assign an array-like object as a 2D array.
    Return type \texttt{ndarray}

_set\_nd\_array (name, value, ndmin, dtype)
    Assign an array-like object (value) to the name attribute as ndarray.
    Performs an inplace update of this instance.

    Parameters
    
    • name (str) – The name of the to-be set attribute.
    • value (array-like) – The array-like object to-be assigned to name. The supplied object is converted into into an array beforehand.
    • ndmin (int) – The minimum number of dimensions of the to-be assigned array.
    • dtype (type or numpy.dtype) – The desired datatype of the to-be assigned array.
    • Exceptions –
    • ---------- –
    • ValueError – Raised if value array construction was unsuccessful.
    Return type None

property segment\_name
    Get or set the "segment name" column in \texttt{PSFContainer.atoms}.
    Return type Series

property residue\_id
    Get or set the "residue ID" column in \texttt{PSFContainer.atoms}.
    Return type Series

__hash__ (self)
    Return the hash of this instance.
The returned hash is constructed from two components: * The hash of this instances’ class type. * The hashes of all key/value pairs in this instances’ (non-private) attributes.

If an unhashable instance variable is encountered, e.g. a list, then its id() is used for hashing.

This method will raise a TypeError if the class attribute AbstractDataClass._HASHABLE is False.

See also:

AbstractDataClass._PRIVATE_ATTR A set with the names of private instance variables.

AbstractDataClass._HAS_HASHABLE Whether or not this class is hashable.

AbstractDataClass._hash_fallback Fallback function for AbstractDataClass. __hash__() incase of recursive calls.

AbstractDataClass._hash An instance variable for caching the hash() of this instance.

Return type int

__weakref__
list of weak references to the object (if defined)

property residue_name
Get or set the "residue name" column in PSFContainer.atoms.

Return type Series

property atom_name
Get or set the "atom name" column in PSFContainer.atoms.

Return type Series

property atom_type
Get or set the "atom type" column in PSFContainer.atoms.

Return type Series

property charge
Get or set the "charge" column in PSFContainer.atoms.

Return type Series

property mass
Get or set the "mass" column in PSFContainer.atoms.

Return type Series

classmethod read(filename, encoding=None, **kwargs)
Construct a new instance from this object’s class by reading the content of filename.

Parameters

* filename (str, bytes, os.PathLike or a file object) – The path+filename or a file object of the to-be read .psf file. In practice, any iterable can substitute the role of file object as long iteration returns either strings or bytes (see encoding).

* encoding (str, optional) – Encoding used to decode the input (e.g. "utf-8"). Only relevant when a file object is supplied to filename and the datastream is not in text mode.

* **kwargs (Any) – Optional keyword arguments that will be passed to both PSFContainer._read_iterate() and PSFContainer._read_postprocess().
See also:

PSFContainer._read_iterate() An abstract method for parsing the opened file in PSFContainer.read().

PSFContainer._read_postprocess() Post processing the class instance created by PSFContainer.read().

Return type PSFContainer
classmethod _read_iterate(iterator)
An abstract method for parsing the opened file in read.

Parameters
iterator (Iterator [str]) – An iterator that returns str instances upon iteration.

Return type Dict[str,Any]
Returns

- dict [str, Any] – A dictionary with keyword arguments for a new instance of this objects’ class.
- **kwargs (Any) – Optional keyword arguments.

See also:

read() The main method for reading files.

__read_postprocess (filename, encoding=None, **kwargs)
Post processing the class instance created by read().

Parameters

- filename (str, bytes, os.PathLike or a file object) – The path+filename or a file object of the to-be read .psf file. In practice, any iterable can substitute the role of file object as long iteration returns either strings or bytes (see encoding).
- encoding (str, optional) – Encoding used to decode the input (e.g. "utf-8"). Only relevant when a file object is supplied to filename and the datastream is not in text mode.
- **kwargs (Any) – Optional keyword arguments that will be passed to both PSFContainer._read_iterate() and PSFContainer._read_postprocess().

See also:

PSFContainer.read() The main method for reading files.

Return type None
classmethod _post_process_psf(psf_dict)
Post-process the output of PSF.read(), casting the values into appropriate objects.

- The title block is converted into a 1D array of strings.
- The atoms block is converted into a Pandas DataFrame.
- All other blocks are converted into 2D arrays of integers.
Parameters **psf_dict** (**dict [str, numpy.ndarray]**) – A dictionary holding the content of a .psf file (see PSFContainer.read_psf()).

**Returns** The .psf output, *psf_dict*, with properly formatted values.

**Return type** **dict [str, numpy.ndarray]**

write (**filename**, **encoding=None**, **kwargs**)  
Write the content of this instance to **filename**.

**Parameters**

- **filename** (**str, bytes, os.PathLike or a file object**) – The path+filename or a file object of the to-be read .psf file. Contrary to _read_postprocess(), file objects can not be substituted for generic iterables.

- **encoding** (**str, optional**) – Encoding used to decode the input (e.g. "utf-8"). Only relevant when a file object is supplied to **filename** and the datastream is not in text mode.

- **kwargs** (**Any**) – Optional keyword arguments that will be passed to _write_iterate().

**See also:**

PSFContainer._write_iterate() Write the content of this instance to an opened datastream.

PSFContainer._get_writer() Take a write() method and ensure its first argument is properly encoded.

**Return type** None

_write_iterate (**write**, **kwargs**)  
Write the content of this instance to an opened datastream.

The to-be-written content of this instance should be passed as **str**. Any (potential) encoding is handled by the **write** parameter.

**Example**

Basic example of a potential _write_iterate() implementation.

```python  
>>> iterator = self.as_dict().items()  
>>> for key, value in iterator:  
...     value: str = f'{key} = {value}'  
...     write(value)  
>>> return None
```

**Parameters**

- **writer** (**Callable**) – A callable for writing the content of this instance to a **file object**. An example would be the `io.TextIOWrapper.write()` method.

- **kwargs** (**optional**) – Optional keyword arguments.

**See also:**

PSFContainer.write() The main method for writing files.

**Return type** None
_write_top (write)
Write the top-most section of the to-be create .psf file.

The following blocks are serialized:
• PSF.title
• PSF.atoms

Parameters write (Callable[[AnyStr], None]) – A callable for writing the content of this instance to a file object. An example would be the io.TextIOWrapper.write() method.

Returns A string constructed from the above-mentioned psf blocks.

Return type str

See also:
PSFContainer.write() The main method for writing .psf files.

_write_bottom (write)
Write the bottom-most section of the to-be create .psf file.

The following blocks are serialized:
• PSF.bonds
• PSF.angles
• PSF.dihedrals
• PSF.impropers
• PSF.donors
• PSF.acceptors
• PSF.no_nonbonded

Parameters write (Callable[[AnyStr], None]) – A callable for writing the content of this instance to a file object. An example would be the io.TextIOWrapper.write() method.

See also:
PSFContainer.write() The main method for writing .psf files.

Return type None

static _serialize_array (array, items_per_row=4)
Serialize an array into a single string; used for creating .psf files.

Newlines are placed for every items_per_row rows in array.

Parameters
• array (numpy.ndarray) – A 2D array.
• items_per_row (int) – The number of values per row before switching to a new line.

Returns A serialized array.
Return type  str

See also:

PSFContainer.write()  The main method for writing .psf files.

update_atom_charge(atom_type, charge)
Change the charge of atom_type to charge.

Parameters

• atom_type (str) – An atom type in PSFContainer.atoms ["atom type"].

• charge (float) – The new atomic charge to-be assigned to atom_type. See PSFContainer.atoms ["charge"].

Raises ValueError – Raised if charge cannot be converted into a float.

Return type  None

update_atom_type(atom_type_old, atom_type_new)
Change the atom type of a atom_type_old to atom_type_new.

Parameters

• atom_type_old (str) – An atom type in PSFContainer.atoms ["atom type"].

• atom_type_new (str) – The new atom type to-be assigned to atom_type. See PSFContainer.atoms ["atom type"].

Return type  None

generate_bonds(mol)
Update PSFContainer.bonds with the indices of all bond-forming atoms from mol.

Parameters  mol (plams.Molecule) – A PLAMS Molecule.

Return type  None

generate_angles(mol)
Update PSFContainer.angles with the indices of all angle-defining atoms from mol.

Parameters  mol (plams.Molecule) – A PLAMS Molecule.

Return type  None

generate_dihedrals(mol)
Update PSFContainer.dihedrals with the indices of all proper dihedral angle-defining atoms from mol.

Parameters  mol (plams.Molecule) – A PLAMS Molecule.

Return type  None

generate_impropers(mol)
Update PSFContainer.impropers with the indices of all improper dihedral angle-defining atoms from mol.

Parameters  mol (plams.Molecule) – A PLAMS Molecule.

Return type  None

generate_atoms(mol, id_map=None)
Update PSFContainer.atoms with the all properties from mol.
Dataframe keys in `PSFContainer.atoms` are set based on the following values in `mol`:

<table>
<thead>
<tr>
<th>DataFrame column</th>
<th>Value</th>
<th>Backup value(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;segment name&quot;</td>
<td>&quot;MOL{:d}&quot;; See &quot;atom type&quot; and &quot;residue name&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;residue ID&quot;</td>
<td><code>Atom.properties[&quot;pdb_info&quot;][&quot;ResidueNumber&quot;]</code></td>
<td>1</td>
</tr>
<tr>
<td>&quot;residue name&quot;</td>
<td><code>Atom.properties[&quot;pdb_info&quot;][&quot;ResidueName&quot;]</code></td>
<td>&quot;COR&quot;</td>
</tr>
<tr>
<td>&quot;atom name&quot;</td>
<td><code>Atom.symbol</code></td>
<td></td>
</tr>
<tr>
<td>&quot;atom type&quot;</td>
<td><code>Atom.properties[&quot;symbol&quot;]</code></td>
<td><code>Atom.symbol</code></td>
</tr>
<tr>
<td>&quot;charge&quot;</td>
<td><code>Atom.properties[&quot;charge_float&quot;]</code></td>
<td><code>Atom.properties[&quot;charge&quot;] &amp; 0.0</code></td>
</tr>
<tr>
<td>&quot;mass&quot;</td>
<td><code>Atom.mass</code></td>
<td></td>
</tr>
<tr>
<td>&quot;0&quot;</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

If a value is not available in a particular `Atom.properties` instance then a backup value will be set.

**Parameters**

- `mol (plams.Molecule)` – A PLAMS Molecule.
- `id_map (Mapping[int, Hashable], optional)` – A mapping of ligand residue ID’s to a custom (Hashable) descriptor. Can be used for generating residue names for quantum dots with multiple different ligands.

**Return type** None

`_construct_segment_name (id_map=None)`

Generate a list for the PSF.atoms["segment name"] column.

**Return type** List[str]

`to_atom_dict()`

Create a dictionary of atom types and lists with their respective indices.

**Returns** A dictionary with atom types as keys and lists of matching atomic indices as values. The indices are 0-based.

**Return type** dict[str, list[int]]

`write_pdb (mol, pdb_file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>, copy_mol=True)`

Construct a .pdb file from this instance and `mol`.

**Parameters**

- `mol (plams.Molecule)` – A PLAMS Molecule.
- `copy_mol (bool)` – If True, create a copy of `mol` instead of modifying it inplace.
- `pdb_file (str of TextIOBase)` – A filename or a file-like object.

**Return type** None
2.8 PRMContainer

2.8.1 FOX.io.read_prm

A class for reading and generating .prm parameter files.

Index

| PRMContainer([filename, atoms, bonds, ...]) | A container for managing prm files. |
| PRMContainer.read(filename[, encoding]) | Construct a new instance from this object's class by reading the content of filename. |
| PRMContainer.write([filename, encoding]) | Write the content of this instance to filename. |
| PRMContainer.overlay_mapping(prm_name, param_df) | Update a set of parameters, prm_name, with those provided in param_df. |
| PRMContainer.overlay_cp2k_settings(cp2k_settings) | Forcefield information from PLAMS-style CP2K settings. |

API

class FOX.io.read_prm.PRMContainer (filename=None, atoms=None, bonds=None, angles=None, dihedrals=None, improper=None, improperers=None, nonbonded=None, nonbonded_header=None, nbfix=None, hbond=None)

A container for managing prm files.

pd_printoptions : class:`dict` [:class:`str`, :class:`object`], private
A dictionary with Pandas print options. See Options and settings.

CP2K_TO_PRM : class:`Mapping<collections.abc.Mapping>` [:class:`str`, :class:`PRMMapping<FOX.io.cp2k_to_prm.PRMMapping>`]
A mapping providing tools for converting CP2K settings to .prm-compatible values. See CP2K_TO_PRM.

classmethod PRMContainer.read (filename, encoding=None, **kwargs)
Construct a new instance from this object’s class by reading the content of filename.

Parameters

- **filename** (str, bytes, os.PathLike or a file object) – The path+filename or a file object of the to-be-read .psf file. In practice, any iterable can substitute the role of file object as long iteration returns either strings or bytes (see encoding).
- **encoding** (str, optional) – Encoding used to decode the input (e.g. "utf-8"). Only relevant when a file object is supplied to filename and the datastream is not in text mode.
- ****kwars (Any) – Optional keyword arguments that will be passed to both PRMContainer._read_iterate() and PRMContainer._read_postprocess().

See also:

PRMContainer._read_iterate() An abstract method for parsing the opened file in PRMContainer.read().

PRMContainer._read_postprocess() Post processing the class instance created by PRMContainer.read().
Return type `PRMContainer`

`PRMContainer.write(filename=None, encoding=None, **kwargs)`

Write the content of this instance to `filename`.

Parameters

- `filename` (`str, bytes, os.PathLike` or a file object) – The path+filename or a file object of the to-be read .psf file. Contrary to `_read_postprocess()`, file objects cannot be substituted for generic iterables.

- `encoding` (`str`, optional) – Encoding used to decode the input (e.g. "utf-8"). Only relevant when a file object is supplied to `filename` and the datastream is not in text mode.

- `**kwargs` (`Any`) – Optional keyword arguments that will be passed to `_write_iterate()`.

See also:

- `PRMContainer._write_iterate()` Write the content of this instance to an opened datastream.
- `PRMContainer._get_writer()` Take a `write()` method and ensure its first argument is properly encoded.

Return type `None`

`PRMContainer.overlay_mapping(prm_name, param_df, units=None)`

Update a set of parameters, `prm_name`, with those provided in `param_df`.

Examples

```python
>>> from FOX import PRMContainer

>>> prm = PRMContainer(...)  

>>> param_dict = {}

>>> param_dict['epsilon'] = {'Cd Cd': ..., 'Cd Se': ..., 'Se Se': ...}  # epsilon

>>> param_dict['sigma'] = {'Cd Cd': ..., 'Cd Se': ..., 'Se Se': ...}  # sigma

>>> units = ('kcal/mol', 'angstrom')  # input units for epsilon and sigma

>>> prm.overlay_mapping('nonbonded', param_dict, units=units)
```

Parameters

- `prm_name` (`str`) – The name of the parameter of interest. See the keys of `PRMContainer.CP2K_TO_PRM` for accepted values.

- `param_df` (`pandas.DataFrame` or `nested Mapping`) – A DataFrame or nested mapping with the to-be added parameters. The keys should be a subset of `PRMContainer.CP2K_TO_PRM[prm_name]["columns"]`. If the index/nested sub-keys consist of strings then they’ll be split and turned into a `pandas.MultiIndex`. Note that the resulting values are not sorted.

- `units` (`Iterable [str]`, optional) – An iterable with the input units of each column in `param_df`. If None, default to the defaults specified in `PRMContainer.CP2K_TO_PRM[prm_name]["unit"]`. 
**Return type** None

PRMContainer.ovelay_cp2k_settings(cp2k_settings)

Extract forcefield information from PLAMS-style CP2K settings.

Performs an inplace update of this instance.

**Examples**

Example input value for cp2k_settings. In the provided example the cp2k_settings are directly extracted from a CP2K .inp file.

```python
>>> import cp2kparser  # https://github.com/nlesc-nano/CP2K-Parser

>>> filename = str(...)

>>> cp2k_settings: dict = cp2kparser.read_input(filename)

>>> print(cp2k_settings)

{'force_eval': {'mm': {'forcefield': {'nonbonded': {'lennard-jones': [...]}}}}}
```

**Parameters**

- **cp2k_settings** (Mapping) – A Mapping with PLAMS-style CP2K settings.

**See also:**

PRMMapping() PRMMapping A mapping providing tools for converting CP2K settings to .prm-compatible values.

**Return type** None

### 2.9 Recipes

#### 2.9.1 FOX.recipes.param

A set of functions for analyzing and plotting ARMC results.

**Examples**

A general overview of the functions within this module.

```python
>>> import pandas as pd

>>> from FOX.recipes import get_best, overlay_descriptor, plot_descriptor

>>> hdf5_file: str = ...

>>> param: pd.Series = get_best(hdf5_file, name='param')  # Extract the best parameters

>>> rdf: pd.DataFrame = get_best(hdf5_file, name='rdf')  # Extract the matching RDF

# Compare the RDF to its reference RDF and plot

>>> rdf_dict = overlay_descriptor(hdf5_file, name='rdf')

>>> plot_descriptor(rdf_dict)
```
Examples

A small workflow for calculating free energies using distribution functions such as the radial distribution function (RDF).

```python
>>> import pandas as pd
>>> from FOX import get_free_energy
>>> from FOX.recipes import get_best, overlay_descriptor, plot_descriptor

>>> hdf5_file: str = ...

>>> rdf: pd.DataFrame = get_best(hdf5_file, name='rdf')
>>> G: pd.DataFrame = get_free_energy(rdf, unit='kcal/mol')

>>> rdf_dict = overlay_descriptor(hdf5_file, name='rdf')
>>> G_dict = {key: get_free_energy(value) for key, value in rdf_dict.items()}
>>> plot_descriptor(G_dict)
```

Examples

A workflow for plotting parameters as a function of ARMC iterations.

```python
>>> import numpy as np
>>> import pandas as pd
>>> from FOX import from_hdf5
>>> from FOX.recipes import plot_descriptor

>>> hdf5_file: str = ...

>>> param: pd.DataFrame = from_hdf5(hdf5_file, 'param')
>>> param.index.name = 'ARMC iteration'
>>> param_dict = {key: param[key] for key in param.columns.levels[0]}

>>> plot_descriptor(param_dict)
```
This approach can also be used for the plotting of other properties such as the auxiliary error.

```python
>>> ... 
>>> err: pd.DataFrame = from_hdf5(hdf5_file, 'aux_error')
>>> err.index.name = 'ARMC iteration'
>>> err_dict = {'Auxiliary Error': err}
>>> plot_descriptor(err_dict)
```

On occasion it might be desirable to only print the error of, for example, accepted iterations. Given a sequence of booleans (bool_seq), one can slice a DataFrame or Series (df) using df.loc[bool_seq].

```python
>>> ... 
>>> acceptance: np.ndarray = from_hdf5(hdf5_file, 'acceptance')  # Boolean array
>>> err_slice_dict = {key: df.loc[acceptance], value for key, df in err_dict.items()}
>>> plot_descriptor(err_slice_dict)
```
get_best(hdf5_file[, name, i])

Return the PES descriptor or ARMC property which yields the lowest error.

Parameters
• hdf5_file (str) – The path+filename of the ARMC .hdf5 file.
• name (str) – The name of the PES descriptor, e.g. "rdf". Alternatively one can supply an ARMC property such as "acceptance", "param" or "aux_error".
• i (int) – The index of the desired PES. Only relevant for PES-descriptors of state-averaged ARMCs.

Returns A DataFrame of the optimal PES descriptor or other (user-specified) ARMC property.

Return type pandas.DataFrame or pd.Series

overlay_descriptor(hdf5_file[, name, i])

Return the PES descriptor which yields the lowest error and overlay it with the reference PES descriptor.

Parameters
• hdf5_file (str) – The path+filename of the ARMC .hdf5 file.
• name (str) – The name of the PES descriptor, e.g. "rdf".
• i (int) – The index of desired PES. Only relevant for state-averaged ARMCs.

Returns A dictionary of DataFrames. Values consist of DataFrames with two keys: "MM-MD" and "QM-MD". Atom pairs, such as "Cd Cd", are used as keys.

Return type dict [str, pandas.DataFrame]

plot_descriptor(descriptor[, show_fig,...])

Plot a DataFrame or iterable consisting of one or more DataFrames.

Requires the matplotlib package.

Parameters
• descriptor (pandas.DataFrame or Iterable [pandas.DataFrame]) – A DataFrame or an iterable consisting of DataFrames.
• show_fig (bool) – Whether to show the figure or not.
• kind (str) – The plot kind to-be passed to pandas.DataFrame.plot().
• sharex/sharey (bool) – Whether or not the to-be created plots should share their x/y-axes.
**kwargs (Any) – Further keyword arguments for the pandas.DataFrame.plot() method.

Returns A matplotlib Figure.

Return type Figure

See also:

get_best() Return the PES descriptor or ARMC property which yields the lowest error.

overlay_descriptor() Return the PES descriptor which yields the lowest error and overlay it with the reference PES descriptor.

### 2.9.2 FOX.recipes.psf

A set of functions for creating .psf files.

#### Examples

Example code for generating a .psf file. Ligand atoms within the ligand .xyz file and the qd .xyz file should be in the exact same order. For example, implicit hydrogen atoms added by the from_smiles functions are not guaranteed to be ordered, even when using canonical SMILES strings.

```python
>>> from scm.plams import Molecule, from_smiles
>>> from FOX import PSFContainer
>>> from FOX.recipes import generate_psf

# Accepts .xyz, .pdb, .mol or .mol2 files
>>> qd = Molecule(...)  # Accepts an .xyz, .pdb, .mol or .mol2 file
>>> rtf_file : str = ...

>>> ligand_len = 18 + 33 + 2
>>> ligand_atoms = {'C', 'H', 'O'}

>>> ligand: Molecule = extract_ligand(qd, ligand_len, ligand_atoms)

>>> psf: PSFContainer = generate_psf(qd_xyz, ligand_xyz, rtf_file=rtf_file)
>>> psf.write(psf_file)
```

#### Examples

If no ligand .xyz is on hand, or its atoms are in the wrong order, it is possible the extract the ligand directly from the quantum dot. This is demonstrated below with oleate ($C_{18}H_{33}O_2$).

```python
>>> from scm.plams import Molecule
>>> from FOX import PSFContainer
>>> from FOX.recipes import generate_psf, extract_ligand

>>> qd = Molecule(...)  # Accepts an .xyz, .pdb, .mol or .mol2 file
>>> rtf_file : str = ...

>>> ligand_len = 18 + 33 + 2
>>> ligand_atoms = {'C', 'H', 'O'}

>>> ligand: Molecule = extract_ligand(qd, ligand_len, ligand_atoms)

>>> psf: PSFContainer = generate_psf(qd, ligand, rtf_file=rtf_file)
>>> psf.write(...)
```
Examples

Example for multiple ligands.

```python
>>> from typing import List
>>> from scm.plams import Molecule
>>> from FOX import PSFContainer
>>> from FOX.recipes import generate_psf2

qd = Molecule(...)  # Accepts an .xyz, .pdb, .mol or .mol2 file
ligands = ('C[O-]', 'CC[O-]', 'CCC[O-]')
rtf_files = (...,

psf: PSFContainer = generate_psf2(qd, *ligands, rtf_file=rtf_files)
psf.write(...)
```

If the psf construction with `generate_psf2()` fails to identify a particular ligand, it is possible to return all (failed) potential ligands with the `ret_failed_lig` parameter.

```python
>>> ...
ligands = ('CCCCCCCCC[O-]', 'CCCCBr')
failed_mol_list: List[Molecule] = generate_psf2(qd, *ligands, ret_failed_lig=True)
```

Index

- `generate_psf(qd, ligand[, rtf_file, str_file])` Generate a PSFContainer instance for qd.
- `generate_psf2(qd, *ligands[, rtf_file, ...])` Generate a PSFContainer instance for qd with multiple different ligands.
- `extract_ligand(qd, ligand_len, ligand_atoms)` Extract a single ligand from qd.

API

FOX.recipes.psf.**generate_psf** *(qd, ligand[, rtf_file, str_file]=None)*

Generate a PSFContainer instance for qd.

**Parameters**

- `qd` *(str or Molecule)* – The ligand-pacifated quantum dot. Should be supplied as either a Molecule or .xyz file.
- `ligand` *(str or Molecule)* – A single ligand. Should be supplied as either a Molecule or .xyz file.
- `rtf_file` *(str, optional)* – The path+filename of the ligand’s .rtf file. Used for assigning atom types. Alternatively, one can supply a .str file with the `str_file` argument.
- `str_file` *(str, optional)* – The path+filename of the ligand’s .str file. Used for assigning atom types. Alternatively, one can supply a .rtf file with the `rtf_file` argument.

**Returns** A PSFContainer instance with the new .psf file.

**Return type** PSFContainer
FOX.recipes.psf.generate_psf2 (qd, *ligands, rtf_file=None, str_file=None, ret_failed_lig=False)

Generate a PSFContainer instance for qd with multiple different ligands.

Parameters

- **qd** (str or Molecule) – The ligand-pacifated quantum dot. Should be supplied as either a Molecule or .xyz file.
- ***ligands** (str, Molecule or Chem.Mol) – One or more PLAMS/RDkit Molecules and/or SMILES strings representing ligands.
- **rtf_file** (str or Iterable [str], optional) – The path+filename of the ligand’s .rtf files. Filenames should be supplied in the same order as ligands. Used for assigning atom types. Alternatively, one can supply a .str file with the **str_file** argument.
- **str_file** (str or Iterable [str], optional) – The path+filename of the ligand’s .str files. Filenames should be supplied in the same order as ligands. Used for assigning atom types. Alternatively, one can supply a .rtf file with the **rtf_file** argument.
- **ret_failed_lig** (bool) – If True, return a list of all failed (potential) ligands if the function cannot identify any ligands within a certain range. Useful for debugging. If False, raise a MoleculeError.

Returns A single ligand Molecule.

Return type Molecule

Raises MoleculeError – Raised if the function fails to identify any ligands within a certain range. If ret_failed_lig = True, return a list of failed (potential) ligands instead and issue a warning.

FOX.recipes.psf.extract_ligand (qd, ligand_len, ligand_atoms)

Extract a single ligand from qd.

Parameters

- **qd** (str or Molecule) – The ligand-pacifated quantum dot. Should be supplied as either a Molecule or .xyz file.
- **ligand_len** (int) – The number of atoms within a single ligand.
- **ligand_atoms** (str or Iterable [str]) – One or multiple strings with the atomic symbols of all atoms within a single ligand.

Returns A single ligand Molecule.

Return type Molecule

2.9.3 FOX.recipes.ligands

A set of functions for analyzing ligands.

Examples

An example for generating a ligand center of mass RDF.

```python
>>> import numpy as np
>>> import pandas as pd
>>> from FOX import MultiMolecule, example_xyz
>>> from FOX.recipes import get_lig_center
```

>>> mol = MultiMolecule.from_xyz(example_xyz)
>>> start = 123  # Start of the ligands
>>> step = 4     # Size of the ligands

# Add dummy atoms to the ligand-center of mass and calculate the RDF
>>> lig_centra: np.ndarray = get_lig_center(mol, start, step)
>>> mol_new: MultiMolecule = mol.add_atoms(lig_centra, symbols='Xx')
>>> rdf: pd.DataFrame = mol_new.init_rdf(atom_subset=['Xx'])

Or the ADF.

>>> ...

>>> adf: pd.DataFrame = mol_new.init_rdf(atom_subset=['Xx'], r_max=np.inf)

Or the potential of mean force \textit{(i.e. Boltzmann-inverted RDF)}.

>>> ...

>>> from scipy import constants
>>> from scm.plams import Units

>>> RT: float = 298.15 * constants.Boltzmann
>>> kj_to_kcal: float = Units.conversion_ratio('kj/mol', 'kcal/mol')

>>> with np.errstate(divide='ignore'):
...    rdf_invert: pd.DataFrame = -RT * np.log(rdf) * kj_to_kcal
...    rdf_invert[rdf_invert == np.inf] = np.nan  # Set all infinities to not-a-number
Focus on a specific ligand subset is possible by slicing the new ligand Cartesian coordinate array.

```python
grouped_ligands = keep_lig  # Keep these ligands; discard the rest
lig_centra_subset = lig_centra[:, grouped_ligands]
```

Add dummy atoms to the ligand-center of mass and calculate the RDF.

```python
mol_new2 = mol.add_atoms(lig_centra_subset, symbols='Xx')
rdf = mol_new2.init_rdf(atom_subset=set('Xx'))
```

**Examples**

An example for generating a ligand center of mass RDF from a quantum dot with multiple unique ligands. A `.psf` file will herein be used as starting point.

```python
import numpy as np
from FOX import PSFContainer, MultiMolecule, group_by_values
from FOX.recipes import get_multi_lig_center

mol = MultiMolecule.from_xyz(...)
psf = PSFContainer.read(...)

# Gather the indices of each ligand
idx_dict = group_by_values(enumerate(psf.residue_id, start=1))

# Use the `.psf` segment names as symbols
symbols = [psf.segment_name[i].iloc[0] for i in idx_dict.values()]

# Add dummy atoms to the ligand-center of mass and calculate the RDF
lig_centra = get_multi_lig_center(mol, idx_dict.values())
mol_new = mol.add_atoms(lig_centra, symbols=symbols)
rdf = mol_new.init_rdf(atom_subset=set(symbols))
```
Index

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_lig_center</code></td>
<td>Return an array with the (mass-weighted) mean position of each ligands in mol.</td>
</tr>
<tr>
<td><code>get_multi_lig_center</code></td>
<td>Return an array with the (mass-weighted) mean position of each ligands in mol.</td>
</tr>
</tbody>
</table>

API

```python
FOX.recipes.ligands.get_lig_center(mol, start, step[, stop, ...])
```

Return an array with the (mass-weighted) mean position of each ligands in mol.

**Parameters**

- `mol (MultiMolecule)` – A MultiMolecule instance.
- `start (int)` – The atomic index of the first ligand atoms.
- `step (int)` – The number of atoms per ligand.
- `stop (int, optional)` – Can be used for neglecting any ligands beyond a user-specified atomic index.
- `mass_weighted (bool)` – If True, return the mass-weighted mean ligand position rather than its unweighted counterpart.

**Returns**
A new array with the ligand’s centra of mass. If mol.shape == (m, n, 3) then, given k new ligands, the to-be returned array’s shape is (m, k, 3).

**Return type**
`numpy.ndarray`

```python
FOX.recipes.ligands.get_multi_lig_center(mol, idx_iter[, ...])
```

Contrary to `get_lig_center()`, this function can handle molecules with multiple non-unique ligands.

**Parameters**

- `mol (MultiMolecule)` – A MultiMolecule instance.
- `idx_iter (Iterable [Sequence [int]])` – An iterable consisting of integer sequences. Each integer sequence represents a single ligand (by its atomic indices).
- `mass_weighted (bool)` – If True, return the mass-weighted mean ligand position rather than its unweighted counterpart.

**Returns**
A new array with the ligand’s centra of mass. If mol.shape == (m, n, 3) then, given k new ligands (aka the length of idx_iter), the to-be returned array’s shape is (m, k, 3).

**Return type**
`numpy.ndarray`
2.10 file_container

2.10.1 FOX.io.file_container

An abstract container for reading and writing files.

Index

AbstractFileContainer() An abstract container for reading and writing files.

API

class FOX.io.file_container.AbstractFileContainer
    An abstract container for reading and writing files.
    Two public methods are defined within this class:
    
    - AbstractFileContainer.read(): Construct a new instance from this object’s class by
      reading the content to a file or file object. How the content of the to-be read file is parsed has to be
      defined in the AbstractFileContainer._read_iterate() abstract method.
    
    - AbstractFileContainer.write(): Write the content of this instance to an opened file or file
      object. How the content of the to-be exported class instance is parsed has to be defined in the
      AbstractFileContainer._write_iterate()

    The opening, closing and en-/decoding of files is handled by two above-mentioned methods;
    the parsing AbstractFileContainer._read_iterate() * AbstractFileContainer._write_iterate()

    __contains__(value)
    Check if this instance contains value.

    Return type bool

classmethod read(filename, encoding=None, **kwargs)
    Construct a new instance from this object’s class by reading the content of filename.

    Parameters
    
    - filename (str, bytes, os.PathLike or a file object) – The path+filename or a file
      object of the to-be read .psf file. In practice, any iterable can substitute the role of file
      object as long iteration returns either strings or bytes (see encoding).
    
    - encoding (str, optional) – Encoding used to decode the input (e.g. "utf-8"). Only
      relevant when a file object is supplied to filename and the datastream is not in text mode.
    
    - **kwargs (Any) – Optional keyword arguments that will be passed
      to both AbstractFileContainer._read_iterate() and
      AbstractFileContainer._read_postprocess().

    See also:

    AbstractFileContainer._read_iterate() An abstract method for parsing the opened file in
    AbstractFileContainer.read().

    AbstractFileContainer._read_postprocess() Post processing the class instance created
    by AbstractFileContainer.read().
Return type \(\text{AbstractFileContainer}\)

**abstract classmethod** \_read\_iterate (iterator, \*\*kwargs)
An abstract method for parsing the opened file in \textit{read}.

**Parameters**

- \textit{iterator} (\texttt{Iterator [str]}) – An iterator that returns \texttt{str} instances upon iteration.

**Return type** \texttt{Dict[str, Any]}

**Returns**

- \texttt{dict [str, Any]} – A dictionary with keyword arguments for a new instance of this objects' class.
- \texttt{\*\*kwargs (Any)} – Optional keyword arguments.

See also:

\textit{read()}  The main method for reading files.

\_read\_postprocess (filename, encoding=None, \*\*kwargs)
Post processing the class instance created by \textit{read}().

**Parameters**

- \texttt{filename (str, bytes, os.PathLike or a file object)} – The path+filename or a file object of the to-be read .psf file. In practice, any iterable can substitute the role of file object as long iteration returns either strings or bytes (see \texttt{encoding}).
- \texttt{encoding (str, optional)} – Encoding used to decode the input (e.g. "utf-8"). Only relevant when a file object is supplied to \texttt{filename} and the datastream is \texttt{not} in text mode.
- \texttt{\*\*kwargs (Any)} – Optional keyword arguments that will be passed to both \texttt{AbstractFileContainer._read\_iterate()} and \texttt{AbstractFileContainer._read\_postprocess()}.  

See also:

\texttt{AbstractFileContainer.read()}  The main method for reading files.

**Return type** \texttt{None}

\texttt{write (filename, encoding=None, \*\*kwargs)}
Write the content of this instance to \texttt{filename}.

**Parameters**

- \texttt{filename (str, bytes, os.PathLike or a file object)} – The path+filename or a file object of the to-be read .psf file. Contrary to \texttt{_read\_postprocess()}, file objects can not be substituted for generic iterables.
- \texttt{encoding (str, optional)} – Encoding used to decode the input (e.g. "utf-8"). Only relevant when a file object is supplied to \texttt{filename} and the datastream is \texttt{not} in text mode.
- \texttt{\*\*kwargs (Any)} – Optional keyword arguments that will be passed to \texttt{_write\_iterate()}.  

See also:

\texttt{AbstractFileContainer._write\_iterate()}  Write the content of this instance to an opened datastream.
AbstractFileContainer._get_writer()  Take a write() method and ensure its first argument is properly encoded.

Return type  None

static _get_writer(writer, encoding=None)
Take a write() method and ensure its first argument is properly encoded.

Parameters
• writer(Callable) – A write method such as io.TextIOWrapper.write().
• encoding (str, optional) – Encoding used to encode the input of writer (e.g. "utf-8"). This value will be used in str.encode() for encoding the first positional argument provided to instance_method. If None, return instance_method unaltered without any encoding.

Returns  A decorated writer parameter. The first positional argument provided to the decorated callable will be encoded using encoding. writer is returned unaltered if encoding=None.

Return type  Callable

See also:

AbstractFileContainer.write()  The main method for writing files.

abstract _write_iterate(write, **kwargs)
Write the content of this instance to an opened datastream.

The to-be written content of this instance should be passed as str. Any (potential) encoding is handled by the write parameter.

Example
Basic example of a potential _write_iterate() implementation.

```python
>>> iterator = self.as_dict().items()
>>> for key, value in iterator:
...     value: str = f'{key} = {value}'
...     write(value)
>>> return None
```

Parameters
• writer(Callable) – A callable for writing the content of this instance to a file object. An example would be the io.TextIOWrapper.write() method.
• **kwargs (optional) – Optional keyword arguments.

See also:

AbstractFileContainer.write()  The main method for writing files.

Return type  None
classmethod `inherit_annotations()`

A decorator for inheriting annotations and docstrings.

Can be applied to methods of `AbstractFileContainer` subclasses to automatically inherit the docstring and annotations of identical-named functions of its superclass.

**Examples**

```python
>>> class sub_class(AbstractFileContainer):
...     ...
...     @AbstractFileContainer.inherit_annotations()
...     def write(filename, encoding=None, **kwargs):
...         pass

>>> sub_class.write.__doc__ == AbstractFileContainer.write.__doc__
True

>>> sub_class.write.__annotations__ == AbstractFileContainer.write.__annotations__
True
```

**Return type** `type`

`__weakref__`

list of weak references to the object (if defined)

## 2.11 FOX.io.read_prm

An abstract container for reading and writing files.

### 2.11.1 Index

`PRMContainer([filename, atoms, bonds, ...])`

A container for managing prm files.

### 2.11.2 API

```python
class FOX.io.read_prm.PRMPContainer(filename=None, atoms=None, bonds=None, angles=None, dihedrals=None, improper=None, impropers=None, nonbonded=None, nonbonded_header=None, nbfix=None, hbond=None)
```

A container for managing prm files.

```python
pd_printoptions: :class:`dict` [::class:`str`, :class:`object`], private
```

A dictionary with Pandas print options. See Options and settings.

```python
```

A mapping providing tools for converting CP2K settings to .prm-compatible values. See `CP2K_TO_PRM`.

```python
_PRIVATE_ATTR: ClassVar[FrozenSet[str]] = frozenset({'_pd_printoptions'})
```

A `frozenset` with the names of private instance attributes. These attributes will be excluded whenever calling `PRMContainer.as_dict()`.
HEADERS: Tuple[str, ...] = ('ATOMS', 'BONDS', 'ANGLES', 'DIHEDRALS', 'NBFIX', 'HBOND')
A tuple of supported .psf headers.

INDEX: Mapping[str, List[int]] = mappingproxy({'atoms': [2], 'bonds': [0, 1], 'angles': [0, 1], 'dihedrals': [0, 1], 'improper': [0, 1, 2, 3], 'impropers': [0, 1, 2, 3], 'nonbonded': [0, 1], 'nonbonded_header': [0, 1]})
Define the columns for each DataFrame which hold its index

COLUMNS: Mapping[str, Tuple[Union[None, int, float], ...]] = mappingproxy({'atoms': (None, -1, None, nan), 'bonds': ... nan, nan, nan), 'improper': (None, None, None, None, nan, 0, nan), 'impropers': (None, None, None, None, nan, 0, nan)})
Placeholder values for DataFrame columns

__init__ (filename=None, atoms=None, bonds=None, angles=None, dihedrals=None, improper=None, impropers=None, nonbonded=None, nonbonded_header=None, nbfix=None, hbond=None)
Initialize a PRMContainer instance.

Return type None

property improper
Alias for PRMContainer.impropers.

Return type Optional[DataFrame]

static _is_mapping (value)
Check if value is a dict instance; raise a TypeError if not.

Return type dict

__repr__ ()
Return a (machine readable) string representation of this instance.

The string representation consists of this instances’ class name in addition to all (non-private) instance variables.

Returns A string representation of this instance.

Return type str

See also:

PRMContainer._PRIVATE_ATTR A set with the names of private instance variables.

PRMContainer._repr_fallback Fallback function for PRMContainer.__repr__() incase of recursive calls.

PRMContainer._str_iterator() Return an iterable for the iterating over this instances’ attributes.

PRMContainer._str() Returns a string representation of a single key/value pair.

__eq__ (value)
Check if this instance is equivalent to value.

The comparison checks if the class type of this instance and value are identical and if all (non-private) instance variables are equivalent.

Returns Whether or not this instance and value are equivalent.

Return type bool

See also:

PRMContainer._PRIVATE_ATTR A set with the names of private instance variables.

PRMContainer._eq Return if v1 and v2 are equivalent.
**PRMContainer._eq_fallback**  Fallback function for `PRMContainer.__eq__() incase of recursive calls.

`copy (deep=True)`
Return a shallow or deep copy of this instance.

**Parameters**
- **deep (bool)** – Whether or not to return a deep or shallow copy.

**Returns**
A new instance constructed from this instance.

**Return type** `PRMContainer`

`__copy__()`
Return a shallow copy of this instance; see `PRMContainer.copy()`.

**Return type** `~AT`

**classmethod read(filename, encoding=None, **kwargs)**
Construct a new instance from this object’s class by reading the content of `filename`.

**Parameters**
- **filename (str, bytes, os.PathLike or a file object)** – The path+filename or a file object of the to-be read .psf file. In practice, any iterable can substitute the role of file object as long iteration returns either strings or bytes (see `encoding`).
- **encoding (str, optional)** – Encoding used to decode the input (e.g. "utf-8"). Only relevant when a file object is supplied to `filename` and the datastream is not in text mode.
- ****kwargs (Any) – Optional keyword arguments that will be passed to both `PRMContainer._read_iterate()` and `PRMContainer._read_postprocess()`.

See also:

`PRMContainer._read_iterate()` An abstract method for parsing the opened file in `PRMContainer.read()`.

`PRMContainer._read_postprocess()` Post processing the class instance created by `PRMContainer.read()`.

**Return type** `PRMContainer`

**classmethod _read_iterate (iterator)**
An abstract method for parsing the opened file in `read`.

**Parameters**
- **iterator (Iterator [str])** – An iterator that returns str instances upon iteration.

**Return type** `Dict[str, Any]`

**Returns**
- **dict [str, Any]** – A dictionary with keyword arguments for a new instance of this objects’ class.
- ****kwargs (Any) – Optional keyword arguments.

See also:

`read()` The main method for reading files.
classmethod `_read_post_iterate` *(kwargs)*

Post process the dictionary produced by `PRMContainer._read_iterate()`.

**Return type** None

`_read_postprocess` *(filename, encoding=None, **kwargs)*

Post processing the class instance created by `read()`.

**Parameters**

- `filename` *(str, bytes, os.PathLike or a file object)* – The path+filename or a file object of the to-be read .psf file. In practice, any iterable can substitute the role of file object as long iteration returns either strings or bytes (see `encoding`).

- `encoding` *(str, optional)* – Encoding used to decode the input (e.g. "utf-8"). Only relevant when a file object is supplied to `filename` and the datastream is not in text mode.

- `**kwargs` *(Any)* – Optional keyword arguments that will be passed to both `PRMContainer._read_iterate()` and `PRMContainer._read_postprocess()`.

See also:

`PRMContainer.read()` The main method for reading files.

**Return type** None

`write` *(filename=None, encoding=None, **kwargs)*

Write the content of this instance to `filename`.

**Parameters**

- `filename` *(str, bytes, os.PathLike or a file object)* – The path+filename or a file object of the to-be read .psf file. Contrary to `_read_postprocess()`, file objects cannot be substituted for generic iterables.

- `encoding` *(str, optional)* – Encoding used to decode the input (e.g. "utf-8"). Only relevant when a file object is supplied to `filename` and the datastream is not in text mode.

- `**kwargs` *(Any)* – Optional keyword arguments that will be passed to `_write_iterate()`.

See also:

`PRMContainer._write_iterate()` Write the content of this instance to an opened datastream.

`PRMContainer._get_writer()` Take a `write()` method and ensure its first argument is properly encoded.

**Return type** None

`__hash__` *(self)*

Return the hash of this instance.

The returned hash is constructed from two components: * The hash of this instances’ class type. * The hashes of all key/value pairs in this instances’ (non-private) attributes.

If an unhashable instance variable is encountered, e.g. a `list`, then its `id()` is used for hashing.

This method will raise a `TypeError` if the class attribute `AbstractDataClass._HASHABLE` is `False`. 
See also:

**AbstractDataClass**._PRIVATE_ATTR A set with the names of private instance variables.

**AbstractDataClass**._HASHABLE Whether or not this class is hashable.

**AbstractDataClass**._hash_fallback Fallback function for **AbstractDataClass**.__hash__() incase of recursive calls.

**AbstractDataClass**._hash An instance variable for caching the **hash()** of this instance.

Return type: int

__weakref__

dict of weak references to the object (if defined)

___write_iterate___\(write, **kwargs\)

Write the content of this instance to an opened datastream.

The to-be written content of this instance should be passed as str. Any (potential) encoding is handled by the write parameter.

Example

Basic example of a potential __write_iterate() implementation.

```python
>>> iterator = self.as_dict().items()
>>> for key, value in iterator:
...     value: str = f'{key} = {value}'
...     write(value)
>>> return None
```

Parameters

- **writer** (Callable) — A callable for writing the content of this instance to a file object. An example would be the io.TextIOWrapper.write() method.
- ****kwargs (optional) — Optional keyword arguments.

See also:

**PRMContainer.write()** The main method for writing files.

Return type: None

overlay_mapping\(prm_name, param_df, units=None)\)

Update a set of parameters, prm_name, with those provided in param_df.

Examples

```python
>>> from FOX import PRMContainer
>>> prm = PRMContainer(...)  
>>> param_dict = {}  
>>> param_dict['epsilon'] = {'Cd Cd': ..., 'Cd Se': ..., 'Se Se': ...}  
```

(continues on next page)
>>> param_dict['sigma'] = {'Cd Cd': ..., 'Cd Se': ..., 'Se Se': ...}  # sigma
>>> units = ('kcal/mol', 'angstrom')  # input units for epsilon and sigma
>>> prm.overlay_mapping('nonbonded', param_dict, units=units)

Parameters

- `prm_name` (str) – The name of the parameter of interest. See the keys of `PRMContainer.CP2K_TO_PRM` for accepted values.
- `param_df` (pandas.DataFrame or nested Mapping) – A DataFrame or nested mapping with the to-be added parameters. The keys should be a subset of `PRMContainer.CP2K_TO_PRM[prm_name]['columns']`. If the index/nested sub-keys consist of strings then they’ll be split and turned into a pandas.MultiIndex. Note that the resulting values are not sorted.
- `units` (Iterable [str], optional) – An iterable with the input units of each column in `param_df`. If None, default to the defaults specified in `PRMContainer.CP2K_TO_PRM[prm_name]['unit']`.

Return type None

overlay_cp2k_settings (cp2k_settings)
Extract forcefield information from PLAMS-style CP2K settings.
Performs an inplace update of this instance.

Examples

Example input value for `cp2k_settings`. In the provided example the `cp2k_settings` are directly extracted from a CP2K .inp file.

```python
>>> import cp2kparser  # https://github.com/nlesc-nano/CP2K-Parser

>>> filename = str(...)  
>>> cp2k_settings: dict = cp2kparser.read_input(filename)  
>>> print(cp2k_settings)  
{
    'force_eval': {
        'mm': {
            'forcefield': {
                'nonbonded': {
                    'lennard-jones': [...]}}}}
    }
```

Parameters `cp2k_settings` (Mapping) – A Mapping with PLAMS-style CP2K settings.

See also:

PRMMapping() `PRMMapping` A mapping providing tools for converting CP2K settings to .prm-compatible values.

Return type None

_overlay_cp2k_settings (cp2k_settings, name, columns, key_path, key, unit, default_unit, post_process)
Helper function for `PRMContainer.overlay_cp2k_settings()`.
Return type  None

2.12 cp2k_to_prm

2.12.1 FOX.io.cp2k_to_prm

A TypedMapping subclass converting CP2K settings to .prm-compatible values.

Index

<table>
<thead>
<tr>
<th>PRMMapping(name, key, columns, key_path, ...)</th>
<th>A TypedMapping providing tools for converting CP2K settings to .prm-compatible values.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP2K_TO_PRM</td>
<td></td>
</tr>
</tbody>
</table>

API

class FOX.io.cp2k_to_prm.PRMMapping(name, key, columns, key_path, unit, default_unit, post_process)

A TypedMapping providing tools for converting CP2K settings to .prm-compatible values.

Parameters

- name (str) – The name of the PRMContainer attribute. See PRMMapping.name.
- columns (int or Iterable[int]) – The names relevant PRMContainer DataFrame columns. See PRMMapping.columns.
- key_path (str or Iterable[str]) – The path of CP2K Settings keys leading to the property of interest. See PRMMapping.key_path.
- key (str or Iterable[str]) – The key(s) within PRMMapping.key_path containing the actual properties of interest, e.g. "epsilon" and "sigma". See PRMMapping.key.
- unit (str or Iterable[str]) – The desired output unit. See PRMMapping.unit.
- default_unit (str or Iterable[str, optional]) – The default unit as utilized by CP2K. See PRMMapping.default_unit.
- post_process (Callable or Iterable[Callable]) – Callables for post-processing the value of interest. Set a particular callable to None to disable post-processing. See PRMMapping.post_process.

name: :class:`str`
The name of the PRMContainer attribute.

columns: :class:`tuple` [:class:`int`]
The names relevant PRMContainer DataFrame columns.

key_path: :class:`tuple` [:class:`str`]
The path of CP2K Settings keys leading to the property of interest.

key: :class:`tuple` [:class:`str`]
The key(s) within PRMMapping.key_path containing the actual properties of interest, e.g. "epsilon" and "sigma".
unit: :class:`tuple` [:class:`str`]
The desired output unit.

default_unit: :class:`tuple` [:class:`str`, optional]
The default unit as utilized by CP2K.

post_process: :class:`tuple` [:class:`Callable<collections.abc.Callable>`, optional]
Callables for post-processing the value of interest. Set a particular callable to None to disable post-processing.

FOX.io.cp2k_to_prm.CP2K_TO_PRM : MappingProxyType[str, PRMMapping]
A Mapping containing PRMMapping instances.

MappingProxyType({
    'nonbonded':
        PRMMapping(name='nbfix', columns=[2, 3],
        key_path=('input', 'force_eval', 'mm', 'forcefield', 'nonbonded'),
        key=('epsilon', 'sigma'),
        unit=('kcal/mol', 'angstrom'),
        default_unit=('kcal/mol', 'kelvin'),
        post_process=(None, sigma_to_r2)),

    'nonbonded14':
        PRMMapping(name='nbfix', columns=[4, 5],
        key_path=('input', 'force_eval', 'mm', 'forcefield', 'nonbonded'),
        key=('epsilon', 'sigma'),
        unit=('kcal/mol', 'angstrom'),
        default_unit=('kcal/mol', 'kelvin'),
        post_process=(None, sigma_to_r2)),

    'bonds':
        PRMMapping(name='bonds', columns=[2, 3],
        key_path=('input', 'force_eval', 'mm', 'forcefield', 'bond'),
        key=('k', 'r0'),
        unit=('kcal/mol/A**2', 'angstrom'),
        default_unit=('internal_cp2k', 'bohr'),
        post_process=(None, None)),

    'angles':
        PRMMapping(name='angles', columns=[3, 4],
        key_path=('input', 'force_eval', 'mm', 'forcefield', 'bend'),
        key=('k', 'theta0'),
        unit=('kcal/mol', 'degree'),
        default_unit=('hartree', 'radian'),
        post_process=(None, None)),

    'urrey-bradley':
        PRMMapping(name='angles', columns=[5, 6],
        key_path=('input', 'force_eval', 'mm', 'forcefield', 'bend'),
        key=('k', 'r0'),
        unit=('kcal/mol/A**2', 'angstrom'),
        default_unit=('internal_cp2k', 'bohr'),
        post_process=(None, None)),

(continues on next page)
2.13 typed_mapping

2.13.1 FOX.typed_mapping

A module which adds the TypedMapping class.

Index

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TypedMapping()</td>
<td>A Mapping type which only allows a specific set of keys.</td>
</tr>
<tr>
<td>TypedMapping.<strong>setattr</strong>(name, value)</td>
<td>Implement setattr(self, name, value).</td>
</tr>
<tr>
<td>TypedMapping.<strong>delattr</strong>(name)</td>
<td>Implement delattr(self, name).</td>
</tr>
<tr>
<td>TypedMapping.<strong>setitem</strong>(name, value)</td>
<td>Implement self[name] = value.</td>
</tr>
<tr>
<td>TypedMapping.<strong>bool</strong></td>
<td>Get the <strong>bool</strong>() method of TypedMapping.view.</td>
</tr>
<tr>
<td>TypedMapping.<strong>getitem</strong></td>
<td>Get the <strong>getitem</strong>() method of TypedMapping.view.</td>
</tr>
<tr>
<td>TypedMapping.<strong>iter</strong></td>
<td>Get the <strong>iter</strong>() method of TypedMapping.view.</td>
</tr>
<tr>
<td>TypedMapping.<strong>len</strong></td>
<td>Get the <strong>len</strong>() method of TypedMapping.view.</td>
</tr>
<tr>
<td>TypedMapping.<strong>contains</strong></td>
<td>Get the <strong>contains</strong>() method of TypedMapping.view.</td>
</tr>
<tr>
<td>TypedMapping.get</td>
<td>Get the get() method of TypedMapping.view.</td>
</tr>
<tr>
<td>TypedMapping.keys</td>
<td>Get the keys() method of TypedMapping.view.</td>
</tr>
<tr>
<td>TypedMapping.items</td>
<td>Get the items() method of TypedMapping.view.</td>
</tr>
<tr>
<td>TypedMapping.values</td>
<td>Get the values() method of TypedMapping.view.</td>
</tr>
</tbody>
</table>
API

class FOX.typed_mapping.TypedMapping
   A Mapping type which only allows a specific set of keys.
   Values cannot be altered after their assignment.
   
   _PRIVATE_ATTR: :class:`frozenset` [:class:`str`], classvar
      A frozenset defining all private instance variables.
   
   _ATTR: :class:`frozenset` [:class:`str`], classvar
      A frozenset containing all allowed keys. Should be defined at the class level.
   
   view: :class:`MappingProxyType<types.MappingProxyType>` [:class:`str`, :data:`Any<typing.Any>`]
      Return a read-only view of all items specified in TypedMapping._ATTR.

TypedMapping.__setattr__(name, value)
Implement setattr(self, name, value).
   Attributes specified in TypedMapping._PRIVATE_ATTR can freely modified. Attributes specified in
   TypedMapping._ATTR can only be modified when the previous value is None. All other attribute cannot be
   modified any further.
   
   Return type None

TypedMapping.__delattr__(name)
Implement delattr(self, name).
   Raises an AttributeError, instance variables cannot be deleted.
   
   Return type NoReturn

TypedMapping.__setitem__(name, value)
Implement self[name] = value.
   Serves as an alias for TypedMapping.__setattr__() when name is in TypedMapping._ATTR().
   
   Return type None

TypedMapping.__bool__()
Get the __bool__() method of TypedMapping.view.
   
   Return type Callable[[], bool]

TypedMapping.__getitem__()
Get the __getitem__() method of TypedMapping.view.
   
   Return type Callable[[~KT], ~KV]

TypedMapping.__iter__()
Get the __iter__() method of TypedMapping.view.
   
   Return type Callable[[], Iterator[~KT]]

TypedMapping.__len__()
Get the __len__() method of TypedMapping.view.
   
   Return type Callable[[], int]

TypedMapping.__contains__()
Get the __contains__() method of TypedMapping.view.
   
   Return type Callable[[~KT], bool]

TypedMapping.get()
Get the get() method of TypedMapping.view.
Return type `Callable[[~KT, Optional[Any]], ~KV]

TypedMapping.keys()
Get the keys() method of `TypedMapping.view`.

Return type `Callable[[], KeysView[~KT]]`

TypedMapping.items()
Get the items() method of `TypedMapping.view`.

Return type `Callable[[], ItemsView[~KT, ~KV]]`

TypedMapping.values()
Get the values() method of `TypedMapping.view`.

Return type `Callable[[], ValuesView[~KV]]`
Python Module Index

f
FOX.ff.lj_param, 44
FOX.io.cp2k_to_prm, 78
FOX.io.file_container, 69
FOX.io.read_prm, 57
FOX.io.read_psf, 46
FOX.recipes.ligands, 65
FOX.recipes.param, 59
FOX.recipes.psf, 63
FOX.typed_mapping, 80
_read_postprocess ()
(FOX.io.read_prm.PRMContainer method), 75

_read_postprocess ()
(FOX.io.read_psf.PSFContainer method), 52

_serialize_array ()
(FOX.io.read_psf.PSFContainer static method), 54

_set_nd_array ()
(FOX.io.read_psf.PSFContainer method), 50

_str_iterator ()
(FOX.io.read_psf.PSFContainer method), 48

_write_bottom ()
(FOX.io.read_psf.PSFContainer method), 54

_write_iterate ()
(FOX.io.file_container.AbstractFileContainer method), 71

_write_iterate ()
(FOX.io.read_prm.PRMContainer method), 76

_write_iterate ()
(FOX.io.read_psf.PSFContainer method), 53

_write_top ()
(FOX.io.read_psf.PSFContainer method), 54

A
AbstractFileContainer (class in FOX.io.file_container), 69
acceptors (FOX.io.read_psf.PSFContainer attribute), 47
acceptors () (FOX.io.read_psf.PSFContainer property), 50
add_atoms () (FOX.classes.multi_mol.MultiMolecule method), 18
add_pes_evaluator ()
(FOX.classes.monte_carlo.MonteCarlo method), 39
angles (FOX.io.read_psf.PSFContainer attribute), 47
angles () (FOX.io.read_psf.PSFContainer property), 49
ARMC (class in FOX.classes.armc), 41
as_dict () (FOX.io.read_psf.PSFContainer method), 49
as_mass_weighted ()
(FOX.classes.multi_mol.MultiMolecule method), 28
as_mol () (FOX.classes.multi_mol.MultiMolecule method), 27
as_mol2 () (FOX.classes.multi_mol.MultiMolecule method), 27
as_Molecule () (FOX.classes.multi_mol.MultiMolecule method), 28
as_pdb () (FOX.classes.multi_mol.MultiMolecule method), 27
as_xyz () (FOX.classes.multi_mol.MultiMolecule method), 27

atnum () (FOX.classes.multi_mol_magic._MultiMolecule property), 31
atom1 () (FOX.classes.multi_mol_magic._MultiMolecule property), 30
atom12 () (FOX.classes.multi_mol_magic._MultiMolecule property), 30
atom2 () (FOX.classes.multi_mol_magic._MultiMolecule property), 30
atom_name () (FOX.io.read_psf.PSFContainer property), 51
atom_type () (FOX.io.read_psf.PSFContainer property), 51
atoms (FOX.classes.multi_mol.MultiMolecule attribute), 17
atoms () (FOX.io.read_psf.PSFContainer property), 49
B
bonds (FOX.classes.multi_mol.MultiMolecule attribute), 17
bonds (FOX.io.read_psf.PSFContainer attribute), 47
bonds () (FOX.io.read_psf.PSFContainer property), 49

C
charge () (FOX.io.read_psf.PSFContainer property), 51
clear_job_cache ()
(FOX.classes.monte_carlo.MonteCarlo method), 40
clip_move () (FOX.classes.monte_carlo.MonteCarlo method), 40
columns (FOX.io.cp2k_to_prm.PRMMapping attribute), 78
COLUMNS (FOX.io.read_prm.PRMContainer attribute), 73
connectors () (FOX.classes.multi_mol_magic._MultiMolecule property), 31
copy () (FOX.classes.multi_mol_magic._MultiMolecule method), 31
copy () (FOX.io.read_prm.PRMContainer method), 74
copy () (FOX.io.read_psf.PSFContainer method), 49
CP2K_TO_PRM (FOX.io.read_prm.PRMContainer attribute), 57, 72
CP2K_TO_PRM (in module FOX.io.cp2k_to_prm), 79
D
default_unit (FOX.io.cp2k_to_prm.PRMMapping attribute), 79
delete_atoms () (FOX.classes.multi_mol.MultiMolecule method), 17
dihedrals (FOX.io.read_psf.PSFContainer attribute), 47
dihedrals () (FOX.io.read_psf.PSFContainer property), 50

86 Index
do_inner() (FOX.classes.armc.ARMC method), 42
donors (FOX.io.read_psf.PSFContainer attribute), 47
donors() (FOX.io.read_psf.PSFContainer property), 50

e
estimate_lj() (in module FOX.ff.lj_param), 44
example_xyz (in module FOX), 44
extract_ligand() (in module FOX.recipes.psf), 65

f

filename (FOX.io.read_psf.PSFContainer attribute), 47
filename() (FOX.io.read_psf.PSFContainer property), 49
FOX.ff.lj_param module, 44
FOX.io.cp2k_to_prm module, 78
FOX.io.file_container module, 69
FOX.io.read_prm module, 57
FOX.io.read_psf module, 46
FOX.recipes.ligands module, 65
FOX.recipes.param module, 59
FOX.recipes.psf module, 63
FOX.typed_mapping module, 80
from_kf() (FOX.classes.multi_mol.MultiMolecule class method), 29
from_mass_weighted() (FOX.classes.multi_mol.MultiMolecule method), 28
from_Molecule() (FOX.classes.multi_mol.MultiMolecule class method), 28
from_xyz() (FOX.classes.multi_mol.MultiMolecule class method), 29
from_yaml() (FOX.classes.armc.ARMC class method), 41

g
generate_angles() (FOX.io.read_psf.PSFContainer method), 55
generate_atoms() (FOX.io.read_psf.PSFContainer method), 55
generate_bonds() (FOX.io.read_psf.PSFContainer method), 55
generate_dihedrals() (FOX.io.read_psf.PSFContainer method), 55
generate_impropers() (FOX.io.read_psf.PSFContainer method), 55
generate_psff() (in module FOX.recipes.psf), 64
generate_psf() (in module FOX.recipes.psf), 64
generate_psf2() (in module FOX.recipes.psf), 64
generate_angles() (FOX.classes.multi_mol.MultiMolecule method), 26
generate_atoms() (FOX.classes.multi_mol.MultiMolecule static method), 23
generate_bonds() (FOX.classes.multi_mol.MultiMolecule method), 20
generate_center_of_mass() (FOX.classes.multi_mol.MultiMolecule method), 20
generate_dist_mat() (FOX.classes.multi_mol.MultiMolecule method), 25
generate_free_energy() (in module FOX.ff.lj_param), 45
generate_lig_center() (in module FOX.recipes.ligands), 68
generate_multi_lig_center() (in module FOX.recipes.ligands), 68
generate_pair_dict() (FOX.classes.multi_mol.MultiMolecule method), 25
generate_pes_descriptors() (FOX.classes.monte_carlo.MonteCarlo method), 41
generate_rmsd() (FOX.classes.multi_mol.MultiMolecule method), 22
generate_rmsf() (FOX.classes.multi_mol.MultiMolecule method), 23
generate_time_averaged_velocity() (FOX.classes.multi_mol.MultiMolecule method), 22
generate_vacf() (FOX.classes.multi_mol.MultiMolecule method), 25
generate_velocity() (FOX.classes.multi_mol.MultiMolecule method), 22
guess_bonds() (FOX.classes.multi_mol.MultiMolecule method), 18
**HEADERS**

(FOX.io.read_prm.PRMContainer attribute), 72

improper() (FOX.io.read_prm.PRMContainer property), 73

impropers (FOX.io.read_psf.PSFContainer attribute), 47

impropers() (FOX.io.read_psf.PSFContainer property), 50

INDEX (FOX.io.read_prm.PRMContainer attribute), 73

inherit_annotations()

(FOX.io.file_container.AbstractFileContainer class method), 71

init_adf() (FOX.classes.multi_mol.MultiMolecule method), 26

init_average_velocity()

(FOX.classes.multi_mol.MultiMolecule method), 20

init_power_spectrum()

(FOX.classes.multi_mol.MultiMolecule method), 25

init_rdf() (FOX.classes.multi_mol.MultiMolecule method), 24

init_rmsd() (FOX.classes.multi_mol.MultiMolecule method), 21

init_rmsf() (FOX.classes.multi_mol.MultiMolecule method), 21

init_shell_search()

(FOX.classes.multi_mol.MultiMolecule method), 23

init_time_averaged_velocity()

(FOX.classes.multi_mol.MultiMolecule method), 20

items() (FOX.classes.monte_carlo.MonteCarlo method), 39

items() (FOX.typed_mapping.TypedMapping method), 82

**J**

job_name()

(FOX.classes.monte_carlo.MonteCarlo property), 38

**K**

key (FOX.io.cp2k_to_prm.PRMMapping attribute), 78

key_path (FOX.io.cp2k_to_prm.PRMMapping attribute), 78

keys() (FOX.classes.monte_carlo.MonteCarlo method), 39

keys() (FOX.typed_mapping.TypedMapping method), 82

**L**

loc() (FOX.classes.multi_mol_magic._MultiMolecule property), 30

logger() (FOX.classes.monte_carlo.MonteCarlo property), 39

**M**

mass() (FOX.classes.multi_mol_magic._MultiMolecule property), 31

mass() (FOX.io.read_psf.PSFContainer property), 51

md_settings() (FOX.classes.monte_carlo.MonteCarlo property), 38

module

FOX.ff.lj_param, 44
FOX.io.cp2k_to_prm, 78
FOX.io.file_container, 69
FOX.io.read_prm, 57
FOX.io.read_psf, 46
FOX.recipes.ligands, 65
FOX.recipes.param, 59
FOX.recipes.psf, 63
FOX.typed_mapping, 80

mol (FOX.classes.multi_mol_magic._MultiMolecule attribute), 30

molecule() (FOX.classes.monte_carlo.MonteCarlo property), 38

MonteCarlo (class in FOX.classes.monte_carlo), 38

move()

(FOX.classes.monte_carlo.MonteCarlo method), 39

move_range() (FOX.classes.monte_carlo.MonteCarlo property), 38

MultiMolecule (class in FOX.classes.multi_mol), 17

**N**

name (FOX.io.cp2k_to_prm.PRMMapping attribute), 78

no_nonbonded (FOX.io.read_psf.PSFContainer attribute), 47

no_nonbonded() (FOX.io.read_psf.PSFContainer property), 50

np_printoptions (FOX.io.read_psf.PSFContainer attribute), 48

**O**

order() (FOX.classes.multi_mol_magic._MultiMolecule property), 31

overlay_cp2k_settings()

(FOX.io.read_prm.PRMContainer method), 59, 77

overlay_descriptor() (in module FOX.recipes.param), 62

overlay_mapping()

(FOX.io.read_prm.PRMContainer method), 58, 76

88 Index
P

pd_printoptions (FOX.io.read_prm.PRMContainer attribute), 57, 72
pd_printoptions (FOX.io.read_psf.PSFContainer attribute), 48
plot_descriptor() (in module FOX.recipes.param), 62
post_process (FOX.io.cp2k_to_prm.PRMMapping attribute), 79
preopt_settings()

FOX.classes.monte_carlo.MonteCarlo property), 38
PRMContainer (class in FOX.io.read_prm), 57, 72
PRMMapping (class in FOX.io.cp2k_to_prm), 78
properties (FOX.classes.multi_mol.MultiMolecule attribute), 17
PSFContainer (class in FOX.io.read_psf), 46

R

radius() (FOX.classes.multi_mol_magic._MultiMolecule property), 31
random_slice() (FOX.classes.multi_mol.MultiMolecule property), 18
read() (FOX.io.file_container.AbstractFileContainer class method), 69
read() (FOX.io.read_prm.PRMContainer class method), 57, 74
read() (FOX.io.read_psf.PSFContainer class method), 51
read_multi_xyz() (in module FOX.io.read_xyz), 43
reset_origin() (FOX.classes.multi_mol.MultiMolecule method), 19
residue_argsort()

FOX.classes.multi_mol.MultiMolecule method), 19
residue_id() (FOX.io.read_psf.PSFContainer property), 50
residue_name() (FOX.io.read_psf.PSFContainer property), 51
restart() (FOX.classes.armc.ARMC method), 42
round() (FOX.classes.multi_mol.MultiMolecule method), 17
run_md() (FOX.classes.monte_carlo.MonteCarlo method), 40

S

segment_name() (FOX.io.read_psf.PSFContainer property), 50
sort() (FOX.classes.multi_mol.MultiMolecule method), 19
super_iter_len() (FOX.classes.armc.ARMC property), 41
symbol() (FOX.classes.multi_mol_magic._MultiMolecule property), 31

T

title (FOX.io.read_psf.PSFContainer attribute), 47
title() (FOX.io.read_psf.PSFContainer property), 49
to_atom_dict() (FOX.io.read_psf.PSFContainer method), 56
to_yaml() (FOX.classes.armc.ARMC method), 41
TypedMapping (class in FOX.typed_mapping), 81

U

unit (FOX.io.cp2k_to_prm.PRMMapping attribute), 78
update_atom_charge()

FOX.io.read_psf.PSFContainer method), 55
update_atom_type() (FOX.io.read_psf.PSFContainer method), 55
update_phi() (FOX.classes.armc.ARMC method), 42
values() (FOX.classes.monte_carlo.MonteCarlo method), 39
values() (FOX.typed_mapping.TypedMapping method), 82
view (FOX.typed_mapping.TypedMapping attribute), 81

W

write() (FOX.io.file_container.AbstractFileContainer method), 70
write() (FOX.io.read_prm.PRMContainer method), 58, 75
write() (FOX.io.read_psf.PSFContainer method), 53
write_pdb() (FOX.io.read_psf.PSFContainer method), 56

X

x() (FOX.classes.multi_mol_magic._MultiMolecule property), 31

Y

y() (FOX.classes.multi_mol_magic._MultiMolecule property), 31

Z

z() (FOX.classes.multi_mol_magic._MultiMolecule property), 31

Index 89