## CONTENTS

1 Automated Forcefield Optimization Extension 0.8.11  
   1.1 Currently implemented ......................................................... 3  
   1.2 Using Auto-FOX ................................................................. 3  
   1.3 Installation ................................................................. 4  

2 Auto-FOX Documentation  
   2.1 Radial & Angular Distribution Function ................................. 5  
   2.2 Root Mean Squared Displacement & Fluctuation ....................... 11  
   2.3 The MultiMolecule Class ................................................... 16  
   2.4 Addaptive Rate Monte Carlo ............................................. 58  
   2.5 Monte Carlo Parameters .................................................. 63  
   2.6 Multi-XYZ reader ........................................................... 77  
   2.7 FOX.ff.lj_param ............................................................ 78  
   2.8 PSFContainer ............................................................... 80  
   2.9 PRMContainer ............................................................... 85  
   2.10 Properties ................................................................. 87  
   2.11 Recipes ................................................................. 92  
   2.12 cp2k_to_prm ................................................................. 103  
   2.13 Index ................................................................. 104  
   2.14 API ................................................................. 104  
   2.15 Index ................................................................. 107  
   2.16 API ................................................................. 107  
   2.17 Index ................................................................. 108  
   2.18 API ................................................................. 108  
   2.19 Index ................................................................. 112  
   2.20 API ................................................................. 113  

Python Module Index ......................................................... 115  

Index ................................................................. 117
Contents:
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 -n FOX -c conda-forge python rdkit "h5py>=2.10"`
  - 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@0.8 --upgrade`
- Congratulations, Auto-FOX is now installed and ready for use!

1.3.3 Optional dependencies

- 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 FOX. 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 FOX.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=0, 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
Distance-weighted ADF

Atom pairs
Cd Cd Cd
Cd Cd Se
Cd Se Cd
Cd Se Se
Se Cd Se
Se Se Se

phi / Degrees
2.1. API

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

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, optional)` – 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[str], optional)` – 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.

**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, optional)` – 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[str], optional)` – 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)` – The maximum inter-atomic distance (in Angstrom) for which angles are constructed. The distance cuttoff 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 `\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 cuttoff 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
```

`FOX.recipes.time_resolved_rdf(mol, start=0, stop=None, step=500, **kwargs)`

Calculate the time-resolved radial distribution function (RDF).

### Examples

```python
>>> from FOX import MultiMolecule, example_xyz
>>> from FOX.recipes import time_resolved_rdf

# Calculate each RDF over the course of 500 frames
>>> time_step = 500
>>> mol = MultiMolecule.from_xyz(example_xyz)

>>> rdf_list = time_resolved_rdf(
...     mol, step=time_step, atom_subset=['Cd', 'Se']
... )
```

### Parameters

- **mol** (*MultiMolecule*) – The trajectory in question.
- **start** (*int*) – The initial frame.
- **stop** (*int*, optional) – The final frame. Set to *None* to iterate over all frames.
- **step** (*int*) – The number of frames per individual RDF. Note that lower *step* values will result in increased numerical noise.
- ****kwargs (Any) – Further keyword arguments for `init_rdf()`.

### Returns

A list of dataframes, each containing an RDF calculated over the course of *step* frames.

### See also:

`init_rdf()` Calculate the radial distribution function.
Parameters

- **mol** (*MultiMolecule*) – The trajectory in question.
- **start** (*int*) – The initial frame.
- **stop** (*int*, optional) – The final frame. Set to *None* to iterate over all frames.
- **step** (*int*) – The number of frames per individual RDF. Note that lower **step** values will result in increased numerical noise.
- ****kwargs (**Any**) – Further keyword arguments for *init_rdf()*.

**Returns** A list of dataframes, each containing an RDF calculated over the course of **step** frames.

**Return type** List[pandas.DataFrame]

See also:

*init_rdf()* Calculate the radial distribution function.

### 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.

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

Given a trajectory, mol, stored as a *FOX.MultiMolecule* instance, the RMSD can be calculated with the *FOX.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 (RMSD) represents the time-averaged displacement, with respect to the time-averaged position, as a function of atomic indices.

$$\rho_i^{\text{RMSF}} = \sqrt{\langle (r_i - \langle r_i \rangle)^2 \rangle}$$

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

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

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

```python
>>> 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

```from FOX import MultiMolecule, example_xyz
```

```python
>>> mol = MultiMolecule.from_xyz(example_xyz)

```mol = MultiMolecule.from_xyz(example_xyz)
```

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

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

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

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

RMSF

<table>
<thead>
<tr>
<th>Atoms</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
</table>

Arbitrary atomic index
2.2.3 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.

2.2.4 API

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

Initialize the RMSD calculation, returning a dataframe.

**Parameters**

- `mol_subset` *(slice, optional)* – 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[str], optional)* – 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, optional)* – 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[str], optional)* – 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, optional) – 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[str], optional) – 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 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 MultiMolecule.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

• \texttt{rmsf (pd.DataFrame)} – A dataframe holding the results of an RMSF calculation.
• \texttt{idx\_series (pd.DataFrame)} – A series mapping the indices from \texttt{rmsf} to actual atomic indices.
• \texttt{dist\_dict (dict[str, list[float]])} – A dictionary with atomic symbols (see \texttt{rmsf.columns}) and a list of interatomic distances.

Returns

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

Return type \texttt{dict[str, list[int]]}

Raises \texttt{KeyError} – Raised if a key in \texttt{dist\_dict} is absent from \texttt{rmsf}.

### 2.3 The MultiMolecule Class

The API of the \texttt{FOX.MultiMolecule} class.

#### 2.3.1 API FOX.MultiMolecule

class \texttt{FOX.MultiMolecule} (\texttt{coords=None, atoms=None, bonds=None, properties=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

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

\texttt{atoms}

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

Type \texttt{dict[str, list[str]]}

\texttt{bonds}

A 2D array with indices of the atoms defining all \texttt{k} bonds (columns 1 & 2) and their respective bond orders multiplied by 10 (column 3).

Type \texttt{np.ndarray[np.int64], shape (k, 3)}

\texttt{properties}

A Settings instance for storing miscellaneous user-defined (meta-)data. Is devoid of keys by default.
Type `plams.Settings`

`round(decimals=0, *, inplace=False)`
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.

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

Parameters `atom_subset` (Sequence[str]) – 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 molecule 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.

Examples

```python
>>> 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 molecule 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.

2.3. The MultiMolecule Class
Bonds are guessed based on the first molecule in this instance. Performs an inplace modification of
self.bonds

Parameters

atom_subset (Sequence[ ]str, optional) – 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.

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, optional) – End of the interval.
• p (float) – The probability of including each particular molecule in this instance. Values must be between 0 (0%) and 1 (100%).
• inplace (bool) – Instead of returning the new coordinates, perform an inplace update of this instance.

Returns If inplace is True, return a new molecule.

Return type FOX.MultiMolecule or None

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

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, optional) – Perform the calculation on a subset of molecules in this instance, as determined by their molecule index. Include all m molecules in this instance if None.
• atom_subset (Sequence[ ]str, optional) – 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 FOX.MultiMolecule or None

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  FOXMultiMolecule or None

**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  np.ndarray[np.int64], shape (n,)

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

    Get the center of mass.

    Parameters

        • mol_subset (slice, optional) – 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[str], optional) – 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 array with the centres of mass of m molecules with n atoms.

    Return type  np.ndarray[np.float64], shape (m, 3)

**get_bonds_per_atom** (atom_subset=None)

    Get the number of bonds per atom in this instance.

    Parameters  atom_subset (Sequence[str], optional) – 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 with the number of bonds per atom, for all n atoms in this instance.

    Return type  n np.ndarray [np.int64]

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

    Calculate the average atomic velocty.

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

    The velocity is averaged over all atoms in a particular atom subset.

    Parameters

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

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

        • mol_subset (slice, optional) – 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[str], optional)* – 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 \( m - 1 \) velocities averaged over one or more atom subsets.

**Return type** `pd.DataFrame`

### init_time_averaged_velocity

```python
timestep=1.0,  
rms=False,  
mol_subset=None,  
atom_subset=None)
```

Calculate the time-averaged velocity.

The time-averaged 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.

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

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

• **atom_subset** *(Sequence[str], optional)* – 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

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

Initialize the RMSD calculation, returning a dataframe.

**Parameters**

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

• **atom_subset** *(Sequence[str], optional)* – 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`

### init_rmsf

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

Initialize the RMSF calculation, returning a dataframe.

**Parameters**

• **mol_subset** *(slice, optional)* – Perform the calculation on a subset of molecules in this instance, as determined by their molecule index. Include all \( m \) molecules in this instance if None.
• **atom_subset** \((\text{Sequence}[\text{str}], \text{optional})\) – 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** \((\text{bool})\) – 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** \((timestep=1.0, \text{rms=False, mol_subset=None, atom_subset=None})\)

Return the mean or root-mean squared velocity.

Parameters

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

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

• **mol_subset** \((\text{slice}, \text{optional})\) – 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** \((\text{Sequence}[\text{str}], \text{optional})\) – 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 **np.ndarray[\text{np.float64}], shape \((m−1,)\)**

**get_time_averaged_velocity** \((timestep=1.0, \text{rms=False, mol_subset=None, atom_subset=None})\)

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

Parameters

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

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

• **mol_subset** \((\text{slice}, \text{optional})\) – 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** \((\text{Sequence}[\text{str}], \text{optional})\) – 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 **np.ndarray[\text{np.float64}], shape \((n,)\)**

**get_velocity** \((timestep=1.0, \text{norm=True, mol_subset=None, atom_subset=None})\)

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** \((\text{float})\) – 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, optional) – 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[str], optional) – 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** np.ndarray[np.float64], shape $(m,n)$ or $(m,n,3)$

```
def 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, optional) – 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[str], optional) – 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
```

def get_rmsf(mol_subset=None, atom_subset=None):
    Calculate the root mean square fluctuation (RMSF).

    Parameters

    • **mol_subset**(slice, optional) – 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[str], optional) – 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

```
def 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, optional) – 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[str], optional) – 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

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.DataFrame) – 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)*

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, optional)* – 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[str], optional)* – 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.

**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`

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

Create and return a distance matrix for all molecules and atoms in this instance.

**Parameters**

- **mol_subset** *(slice, optional)* – 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[str], optional)* – 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 3D distance matrix of \( m \) molecules, created out of two sets of \( n \) and \( k \) atoms.

**Return type** `np.ndarray[np.float64]`, shape \((m, n, k)\)

**get_pair_dict** *(atom_subset, r=2)*

Take a subset of atoms and return a dictionary.

**Parameters**

- **atom_subset** *(Sequence[str], optional)* – 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** *(int)* – The length of the to-be returned subsets.

**init_power_spectrum** *(mol_subset=None, atom_subset=None, freq_max=4000)*

Calculate and return the power spectrum associated with this instance.

**Parameters**

- **mol_subset** *(slice, optional)* – 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 [str], optional)* – 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**.

• **freq_max** *(int)* – The maximum to be returned wavenumber (cm\(^{-1}\)).

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

**Return type** `pd.DataFrame`

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

Calculate and return the velocity autocorrelation function (VACF).

**Parameters**

• **mol_subset** *(slice, optional)* – 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 [str], optional)* – 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 containing the power spectrum for each set of atoms in **atom_subset**.

**Return type** `pd.DataFrame`

**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, optional)* – 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 [str], optional)* – 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)* – 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 \( \phi_{ijk} \), the distance \( r_{ijk} \) is defined as \( \max |r_{ij}, r_{jk}| \). 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).
Auto-FOX Documentation, Release 0.8.11

**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-forg dask`.

The transposed array.

Same as `self.transpose()`.

### Examples

```python
given:
>>> x = np.array([[1., 2.], [3., 4.]])
>>> x
array([[ 1.,  2.],
       [ 3.,  4.]])
>>> x.T
array([[ 1.,  3.],
       [ 2.,  4.]])
>>> x = np.array([1., 2., 3., 4.])
>>> x
array([ 1.,  2.,  3.,  4.])
>>> x.T
array([ 1.,  2.,  3.,  4.])
```

See also:

- `transpose`

- `all` *(axis=None, out=None, keepdims=False)*
  
  Returns True if all elements evaluate to True.

  Refer to `numpy.all` for full documentation.

  See also:

  - `numpy.all` equivalent function

- `any` *(axis=None, out=None, keepdims=False)*
  
  Returns True if any of the elements of `a` evaluate to True.

  Refer to `numpy.any` for full documentation.

  See also:

  - `numpy.any` equivalent function

- `argmax` *(axis=None, out=None)*
  
  Return indices of the maximum values along the given axis.

  Refer to `numpy.argmax` for full documentation.

  See also:

  - `numpy.argmax` equivalent function
**argmin** *(axis=None, out=None)*

Return indices of the minimum values along the given axis of `a`.

Refer to `numpy.argmin` for detailed documentation.

See also:

`numpy.argmin` equivalent function

**argpartition** *(kth, axis=-1, kind='introselect', order=None)*

Returns the indices that would partition this array.

Refer to `numpy.argpartition` for full documentation.

New in version 1.8.0.

See also:

`numpy.argpartition` equivalent function

**argsort** *(axis=-1, kind=None, order=None)*

Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

See also:

`numpy.argsort` equivalent function

**astype** *(dtype, order='K', casting='unsafe', subok=True, copy=True)*

Copy of the array, cast to a specified type.

Parameters

- **dtype** *(str or dtype)* – Typecode or data-type to which the array is cast.

- **order** *({'C', 'F', 'A', 'K'}, optional)* – Controls the memory layout order of the result. ‘C’ means C order, ‘F’ means Fortran order, ‘A’ means ‘F’ order if all the arrays are Fortran contiguous, ‘C’ order otherwise, and ‘K’ means as close to the order the array elements appear in memory as possible. Default is ‘K’.

- **casting** *({'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional)* – Controls what kind of data casting may occur. Defaults to ‘unsafe’ for backwards compatibility.

  - ‘no’ means the data types should not be cast at all.
  - ‘equiv’ means only byte-order changes are allowed.
  - ‘safe’ means only casts which can preserve values are allowed.
  - ‘same_kind’ means only safe casts or casts within a kind, like float64 to float32, are allowed.
  - ‘unsafe’ means any data conversions may be done.

- **subok** *(bool, optional)* – If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

- **copy** *(bool, optional)* – By default, astype always returns a newly allocated array. If this is set to false, and the `dtype`, `order`, and `subok` requirements are satisfied, the input array is returned instead of a copy.
**Returns** arr_t – Unless copy is False and the other conditions for returning the input array are satisfied (see description for copy input parameter), arr_t is a new array of the same shape as the input array, with dtype, order given by dtype, order.

**Return type** ndarray

**Notes**

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for “unsafe” casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in ‘safe’ casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

**Raises** ComplexWarning – When casting from complex to float or int. To avoid this, one should use a.real.astype(t).

**Examples**

```python
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. , 2. , 2.5])

>>> x.astype(int)
array([1, 2, 2])
```

**property atnum**

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

**property atom1**

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

**property atom12**

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

**property atom2**

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

**base**

Base object if memory is from some other object.

**Examples**

The base of an array that owns its memory is None:

```python
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```python
>>> y = x[2:]
>>> y.base is x
True
```
byteswap (inplace=False)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place. Arrays of byte-strings are not swapped. The real and imaginary parts of a complex number are swapped individually.

Parameters inplace (bool, optional) – If True, swap bytes in-place, default is False.

Returns out – The byteswapped array. If inplace is True, this is a view to self.

Return type ndarray

Examples

```python
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> list(map(hex, A))
['0x1', '0x100', '0x2233']
>>> A.byteswap(inplace=True)
array([ 256,   1, 13090], dtype=int16)
>>> list(map(hex, A))
['0x100', '0x1', '0x3322']
```

Arrays of byte-strings are not swapped

```python
>>> A = np.array([b'ceg', b'fac'])
>>> A.byteswap()
array([b'ceg', b'fac'], dtype='|S3')
```

A.newbyteorder().byteswap() produces an array with the same values but different representation in memory

```python
>>> A = np.array([1, 2, 3])
>>> A.view(np.uint8)
array([1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=uint8)
>>> A.newbyteorder().byteswap(inplace=True)
array([1, 2, 3])
>>> A.view(np.uint8)
array([0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=uint8)
```

choose (choices, out=None, mode='raise')

Use an index array to construct a new array from a set of choices.

Refer to numpy.choose for full documentation.

See also:

numpy.choose equivalent function

clip (min=None, max=None, out=None, **kwargs)

Return an array whose values are limited to [min, max]. One of max or min must be given.

Refer to numpy.clip for full documentation.

See also:

2.3. The MultiMolecule Class
numpy.clip equivalent function

compress(condition, axis=None, out=None)
Return selected slices of this array along given axis.
Refer to numpy.compress for full documentation.
See also:

numpy.compress equivalent function

conj()
Complex-conjugate all elements.
Refer to numpy.conjugate for full documentation.
See also:

numpy.conjugate equivalent function

conjugate()
Return the complex conjugate, element-wise.
Refer to numpy.conjugate for full documentation.
See also:

numpy.conjugate equivalent function

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

copy(order='C', *, deep=True)
Create a copy of this instance.

Parameters

- order (str) – Controls the memory layout of the copy. See 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

ctypes
An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

Parameters None –

Returns c – Possessing attributes data, shape, strides, etc.

Return type Python object

See also:

numpy.ctypeslib
Notes

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

_ctypes.data
A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as self._array_interface_['data'][0].

Note that unlike data_as, a reference will not be kept to the array: code like ctypes.c_void_p((a + b).ctypes.data) will result in a pointer to a deallocated array, and should be spelt (a + b).ctypes.data_as(ctypes.c_void_p)

_ctypes.shape
A ctypes array of length self.ndim where the basetype is the C-integer corresponding to dtype('p') on this platform. This base-type could be ctypes.c_int, ctypes.c_long, or ctypes.c_longlong depending on the platform. The c_intp type is defined accordingly in numpy.ctypeslib. The ctypes array contains the shape of the underlying array.

Type (c_intp*self.ndim)

_ctypes.strides
A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

Type (c_intp*self.ndim)

_ctypes.data_as(obj)
Return the data pointer cast to a particular c-types object. For example, calling self._as_parameter_ is equivalent to self.data_as(ctypes.c_void_p). Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: self.data_as(ctypes.POINTER(ctypes.c_double)).

The returned pointer will keep a reference to the array.

_ctypes.shape_as(obj)
Return the shape tuple as an array of some other c-types type. For example: self.shape_as(ctypes.c_short).

_ctypes.strides_as(obj)
Return the strides tuple as an array of some other c-types type. For example: self.strides_as(ctypes.c_longlong).

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the as_parameter attribute which will return an integer equal to the data attribute.

Examples

```python
>>> import ctypes
>>> x = np.array([[0, 1], [2, 3]], dtype=np.int32)
>>> x
array([[0, 1],
       [2, 3]], dtype=int32)
>>> x.ctypes.data
31962608 # may vary
```
**cumprod** (*axis=None, dtype=None, out=None*)

Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

See also:

- `numpy.cumprod` equivalent function

**cumsum** (*axis=None, dtype=None, out=None*)

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

See also:

- `numpy.cumsum` equivalent function

**data**

Python buffer object pointing to the start of the array’s data.

**diagonal** (*offset=0, axis1=0, axis2=1*)

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to `numpy.diagonal()` for full documentation.

See also:

- `numpy.diagonal` equivalent function

**dot** (*b, out=None*)

Dot product of two arrays.

Refer to `numpy.dot` for full documentation.

See also:

- `numpy.dot` equivalent function

---

**Examples**
```python
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[2., 2.],
       [2., 2.]])

This array method can be conveniently chained:

```python
>>> a.dot(b).dot(b)
array([[8., 8.],
       [8., 8.]])
```

dtype
Data-type of the array’s elements.

Parameters
None –

Returns d

Return type numpy dtype object

See also:
numpy.dtype

Examples

```python
>>> x
array([[0, 1],
       [2, 3]])

```python
>>> x.dtype
dtype('int32')
```python
>>> type(x.dtype)
type('numpy.dtype')
```

dump(file)
Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

Parameters
file (str or Path) – A string naming the dump file.

Changed in version 1.17.0: pathlib.Path objects are now accepted.

dumps() Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

Parameters
None –

fill(value)
Fill the array with a scalar value.

Parameters
value (scalar) – All elements of a will be assigned this value.

Examples

```python
>>> a = np.array([[1, 2]])
>>> a.fill(0)
```
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1., 1.])

flags
Information about the memory layout of the array.

*C_CONTIGUOUS* (*C*)
The data is in a single, C-style contiguous segment.

*F_CONTIGUOUS* (*F*)
The data is in a single, Fortran-style contiguous segment.

*OWNDATA* (*O*)
The array owns the memory it uses or borrows it from another object.

*WRITEABLE* (*W*)
The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.

*ALIGNED* (*A*)
The data and all elements are aligned appropriately for the hardware.

*WRITEBACKIFCOPY* (*X*)
This array is a copy of some other array. The C-API function PyArray_ResolveWritebackIfCopy must be called before deallocating to the base array will be updated with the contents of this array.

*UPDATEIFCOPY* (*U*)
(Deprecated, use WRITEBACKIFCOPY) This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.

*FNC*
*F_CONTIGUOUS* and not *C_CONTIGUOUS*.

*FORC*
*F_CONTIGUOUS* or *C_CONTIGUOUS* (one-segment test).

*BEHAVED* (*B*)
*ALIGNED* and *WRITEABLE*.

*CARRAY* (*CA*)
*BEHAVED* and *C_CONTIGUOUS*.

*FARRAY* (*FA*)
*BEHAVED* and *F_CONTIGUOUS* and not *C_CONTIGUOUS*.
Notes

The flags object can be accessed dictionary-like (as in a.flags['WRITEABLE']), or by using lowercased attribute names (as in a.flags.writeable). Short flag names are only supported in dictionary access.

Only the WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling ndarray.setflags.

The array flags cannot be set arbitrarily:
- UPDATEIFCOPY can only be set False.
- WRITEBACKIFCOPY can only be set False.
- ALIGNED can only be set True if the data is truly aligned.
- WRITEABLE can only be set True if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension arr.strides[dim] may be arbitrary if arr.shape[dim] == 1 or the array has no elements. It does not generally hold that self.strides[-1] == self.itemsize for C-style contiguous arrays or self.strides[0] == self.itemsize for Fortran-style contiguous arrays is true.

flat

A 1-D iterator over the array.

This is a numpy.flatiter instance, which acts similarly to, but is not a subclass of, Python’s built-in iterator object.

See also:

flatten Return a copy of the array collapsed into one dimension.

Examples

```python
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<class 'numpy.flatiter'>
```

An assignment example:
flatten (order='C')

Return a copy of the array collapsed into one dimension.

Parameters

order (str or 'C', 'F', 'A', 'K', optional) – 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran-style) order. 'A' means to flatten in column-major order if a is Fortran contiguous in memory, row-major order otherwise. 'K' means to flatten a in the order the elements occur in memory. The default is 'C'.

Returns

y – A copy of the input array, flattened to one dimension.

Return type

ndarray

See also:

ravel Return a flattened array.

flat A 1-D flat iterator over the array.

Examples

```python
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```
>>> x.getfield(np.float64)
array([[1., 0.],
       [0., 2.]])

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

>>> x.getfield(np.float64, offset=8)
array([[1., 0.],
       [0., 4.]])

imag

The imaginary part of the array.

Examples

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0. , 0.70710678])
>>> x.imag.dtype
dtype('float64')
```

item (*args)

Copy an element of an array to a standard Python scalar and return it.

Parameters *args (Arguments (variable number and type)) –

- none: in this case, the method only works for arrays with one element (a.size == 1), which
element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which ele-
ment to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument
is interpreted as an nd-index into the array.

Returns z – A copy of the specified element of the array as a suitable Python scalar

Return type Standard Python scalar object

Notes

When the data type of a is longdouble or clongdouble, item() returns a scalar array object because there is
no available Python scalar that would not lose information. Void arrays return a buffer object for item(),
unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This
can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array
using Python’s optimized math.

Examples
```python
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

**itemset(** *args***)

Insert scalar into an array (scalar is cast to array’s dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array `a`.

**Parameters**

*args (Arguments) – If one argument: a scalar, only used in case `a` is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

**Notes**

Compared to indexing syntax, `itemset` provides some speed increase for placing a scalar into a particular location in an `ndarray`, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using `itemset` (and `item`) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

**Examples**

```python
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[2, 2, 6],
       [1, 0, 6],
       [1, 0, 9]])
```

**itemsize**

Length of one array element in bytes.

**Examples**
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16

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

Get, set and del operations are performed using the list(s) of atomic indices associated with the provided atomic symbol(s). Accepts either one or more atomic symbols.

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

>>> idx = mol.atoms['Cd'] + mol.atoms['Se'] + mol.atoms['O']
>>> (mol.loc['Cd', 'Se', 'O'] == mol[idx]).all()
True

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

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

Parameters mol (**FOX.MultiMolecule**) – A MultiMolecule instance; see _MolLoc.mol.

mol
A MultiMolecule instance.

Type **FOX.MultiMolecule**

atoms_view
A read-only view of _MolLoc.mol.atoms.

Type **Mapping**

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

max (axis=None, out=None, keepdims=False, initial=<no value>, where=True)
Return the maximum along a given axis.

Refer to **numpy.amax** for full documentation.

See also:

**numpy.amax** equivalent function

---

2.3. The MultiMolecule Class 39
**mean** (*axis=None, dtype=None, out=None, keepdims=False*)  
Returns the average of the array elements along given axis.  
Refer to *numpy.mean* for full documentation.  
See also:  
  * numpy.mean equivalent function

**min** (*axis=None, out=None, keepdims=False, initial=<no value>, where=True*)  
Return the minimum along a given axis.  
Refer to *numpy.amin* for full documentation.  
See also:  
  * numpy.amin equivalent function

**nbytes**  
Total bytes consumed by the elements of the array.

**Notes**

Does not include memory consumed by non-element attributes of the array object.

**Examples**

```python
>>> x = np.zeros((3,5,2), dtype=np.complex128)  
>>> x.nbytes  
480  
>>> np.prod(x.shape) * x.itemsize  
480
```

**ndim**  
Number of array dimensions.

**Examples**

```python
>>> x = np.array([1, 2, 3])  
>>> x.ndim  
1  
>>> y = np.zeros((2, 3, 4))  
>>> y.ndim  
3
```

**newbyteorder** (*new_order='S'*')  
Return the array with the same data viewed with a different byte order.  
Equivalent to:  
  * arr.view(arr.dtype.newbyteorder(new_order))

Changes are also made in all fields and sub-arrays of the array data type.
**Parameters** new_order (string, optional) – Byte order to force; a value from the byte order specifications below. new_order codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of new_order for the alternatives above. For example, any of ‘B’ or ‘b’ or ‘biggish’ are valid to specify big-endian.

**Returns** new_arr – New array object with the dtype reflecting given change to the byte order.

**Return type** array

nonzero()

Return the indices of the elements that are non-zero.

Refer to numpy.nonzero for full documentation.

See also:

numpy.nonzero equivalent function

**property order**

Get or set the bond orders for all bonds in MultiMolecule.bonds as 1D array.

**partition (kth, axis=-1, kind='introselect', order=None)**

Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array. All elements smaller than the kth element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

**Parameters**

- kth (int or sequence of ints) – Element index to partition by. The kth element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of kth it will partition all elements indexed by kth of them into their sorted position at once.
- axis (int, optional) – Axis along which to sort. Default is -1, which means sort along the last axis.
- kind ({'introselect'}, optional) – Selection algorithm. Default is ‘introselect’.
- order (str or list of str, optional) – When a is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need to be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

See also:

numpy.partition Return a parititioned copy of an array.
argpartition  Indirect partition.
sort  Full sort.

Notes

See np.partition for notes on the different algorithms.

Examples

```python
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
>>> a
array([2, 1, 3, 4])
```

```python
>>> a.partition((1, 3))
>>> a
array([1, 2, 3, 4])
```

prod  
(axis=None, dtype=None, out=None, keepdims=False, initial=1, where=True)
Return the product of the array elements over the given axis

Refer to numpy.prod for full documentation.

See also:

numpy.prod  equivalent function

ptp  
(axis=None, out=None, keepdims=False)
Peak to peak (maximum - minimum) value along a given axis.

Refer to numpy.ptp for full documentation.

See also:

numpy.ptp  equivalent function

put  
(indices, values, mode='raise')
Set a.flat[n] = values[n] for all n in indices.

Refer to numpy.put for full documentation.

See also:

numpy.put  equivalent function

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

ravel  
(order)
Return a flattened array.

Refer to numpy.ravel for full documentation.

See also:

numpy.ravel  equivalent function
**ndarray.flat** a flat iterator on the array.

**real**
The real part of the array.

---

Examples

```python
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1. , 0.70710678])
>>> x.real.dtype
dtype('float64')
```

See also:

- [numpy.real](#) equivalent function

**repeat** *(repeats, axis=None)*
Repeat elements of an array.

Refer to [numpy.repeat](#) for full documentation.

See also:

- [numpy.repeat](#) equivalent function

**reshape** *(shape, order='C')*
Returns an array containing the same data with a new shape.

Refer to [numpy.reshape](#) for full documentation.

See also:

- [numpy.reshape](#) equivalent function

**Notes**

Unlike the free function [numpy.reshape](#), this method on `ndarray` allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

**resize** *(new_shape, refcheck=True)*
Change shape and size of array in-place.

**Parameters**

- **new_shape** (tuple of ints, or n ints) – Shape of resized array.
- **refcheck** *(bool, optional)* – If False, reference count will not be checked. Default is True.

**Returns**

**Return type** None

**Raises**

---

2.3. The MultiMolecule Class
• **ValueError** – If a does not own its own data or references or views to it exist, and the data memory must be changed. PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

• **SystemError** – If the order keyword argument is specified. This behaviour is a bug in NumPy.

**See also:**

*resize*  
Return a new array with the specified shape.

**Notes**

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set `refcheck` to False.

**Examples**

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```python
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```python
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```python
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3)  # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```python
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that references or is referenced ...
```

Unless `refcheck` is False:
```python
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

**searchsorted** *(v, side='left', sorter=None)*

Find indices where elements of *v* should be inserted in *a* to maintain order.

For full documentation, see *numpy.searchsorted*

See also:

*numpy.searchsorted* equivalent function

**setfield** *(val, dtype, offset=0)*

Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*’s field defined by *dtype* and beginning *offset* bytes into the field.

Parameters

- **val** *(object)* – Value to be placed in field.
- **dtype** *(dtype object)* – Data-type of the field in which to place *val*.
- **offset** *(int, optional)* – The number of bytes into the field at which to place *val*.

Returns

Return type None

See also:

*getfield*

Examples

```python
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]], dtype=int32)
>>> x
array([[1.0e+000, 1.5e-323, 1.5e-323],
       [1.5e-323, 1.0e+000, 1.5e-323],
       [1.5e-323, 1.5e-323, 1.0e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```
setflags\((\text{write=None}, \text{align=None}, \text{uic=None})\)

Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by \(a\) (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The WRITEBACKIFCOPY and (deprecated) UPDATEIFCOPY flags can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

**Parameters**

- **write**\((bool, \text{optional})\) – Describes whether or not \(a\) can be written to.
- **align**\((bool, \text{optional})\) – Describes whether or not \(a\) is aligned properly for its type.
- **uic**\((bool, \text{optional})\) – Describes whether or not \(a\) is a copy of another “base” array.

**Notes**

Array flags provide information about how the memory area used for the array is to be interpreted. There are 7 Boolean flags in use, only four of which can be changed by the user: WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) (deprecated), replaced by WRITEBACKIFCOPY;

WRITEBACKIFCOPY (X) this array is a copy of some other array (referenced by .base). When the C-API function PyArray_ResolveWritebackIfCopy is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

**Examples**

```python
>>> y = np.array([[3, 1, 7],
...               [2, 0, 0],
...               [8, 5, 9]])
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
WRITEBACKIFCOPY : False
UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
（continues on next page）
```
OWNDATA : True
WRITEABLE : False
ALIGNED : False
WRITEBACKIFCOPY : False
UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True

shape

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape
the array in-place by assigning a tuple of array dimensions to it. As with numpy.reshape, one of the new
shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining
dimensions. Reshaping an array in-place will fail if a copy is required.

Examples

>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[0., 0., 0., 0., 0., 0., 0., 0.],
    [0., 0., 0., 0., 0., 0., 0., 0.],
    [0., 0., 0., 0., 0., 0., 0., 0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> np.zeros((4,2))[::2].shape = (-1,)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Incompatible shape for in-place modification. Use
'.reshape()' to make a copy with the desired shape.

See also:

numpy.reshape similar function
ndarray.reshape similar method

size

Number of elements in the array.

Equal to np.prod(a.shape), i.e., the product of the array’s dimensions.
Notes

*a.size* returns a standard arbitrary precision Python integer. This may not be the case with other methods of obtaining the same value (like the suggested `np.prod(a.shape)`, which returns an instance of `np.int_`), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

Examples

```python
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

**squeeze** *(axis=None)*

Remove single-dimensional entries from the shape of `a`.

Refer to `numpy.squeeze` for full documentation.

See also:

`numpy.squeeze` equivalent function

**std** *(axis=None, dtype=None, out=None, ddof=0, keepdims=False)*

Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

See also:

`numpy.std` equivalent function

**strides**

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element (i[0], i[1], ..., i[n]) in an array `a` is:

```python
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```python
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array `x` will be `(20, 4)`.
See also:

`numpy.lib.stride_tricks.as_strided`

Examples

```python
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset = sum(y.strides * np.array((1,1,1)))
>>> offset / y.itemsize
17
```

```python
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

`sum(axis=None, dtype=None, out=None, keepdims=False, initial=0, where=True)`

Return the sum of the array elements over the given axis.

Refer to `numpy.sum` for full documentation.

See also:

`numpy.sum` equivalent function

`sparse`更换为`swapaxes`

`swapaxes(axis1, axis2)`

Return a view of the array with `axis1` and `axis2` interchanged.

Refer to `numpy.swapaxes` for full documentation.

See also:

`numpy.swapaxes` equivalent function

`property symbol`

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

`take(indices, axis=None, out=None, mode='raise')`

Return an array formed from the elements of `a` at the given indices.

Refer to `numpy.take` for full documentation.
See also:

numpy.take equivalent function

tobytes (order='C')

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

New in version 1.9.0.

Parameters

order ({'C', 'F', None}, optional) – Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

Returns

s – Python bytes exhibiting a copy of a’s raw data.

Return type

bytes

Examples

>>> x = np.array([[0, 1], [2, 3]], dtype='<u2')
>>> x.tobytes()
'\x00\x00\x01\x00\x02\x00\x03\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
'\x00\x00\x02\x00\x01\x00\x03\x00'

tofile (fid, sep='', format='%s')

Write array to a file as text or binary (default).

Data is always written in ‘C’ order, independent of the order of a. The data produced by this method can be recovered using the function fromfile().

Parameters

- fid (file or str or Path) – An open file object, or a string containing a filename.

  Changed in version 1.17.0: pathlib.Path objects are now accepted.

- sep (str) – Separator between array items for text output. If “” (empty), a binary file is written, equivalent to file.write(a.tobytes()).

- format (str) – Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using “format” % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

When fid is a file object, array contents are directly written to the file, bypassing the file object’s write method. As a result, tofile cannot be used with files objects supporting compression (e.g., GzipFile) or file-like objects that do not support fileno() (e.g., BytesIO).
tolist()

Return the array as an `a.ndim`-levels deep nested list of Python scalars.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible builtin Python type, via the `~numpy.ndarray.item` function.

If `a.ndim` is 0, then since the depth of the nested list is 0, it will not be a list at all, but a simple Python scalar.

**Parameters** none –

**Returns** y – The possibly nested list of array elements.

**Return type** object, or list of object, or list of list of object, or ..

**Notes**

The array may be recreated via `a = np.array(a.tolist())`, although this may sometimes lose precision.

**Examples**

For a 1D array, `a.tolist()` is almost the same as `list(a)`, except that `tolist` changes numpy scalars to Python scalars:

```python
>>> a = np.uint32([1, 2])
>>> a_list = list(a)
>>> a_list
[1, 2]
>>> type(a_list[0])
<class 'numpy.uint32'>
>>> a_tolist = a.tolist()
>>> a_tolist
[1, 2]
>>> type(a_tolist[0])
<class 'int'>
```

Additionally, for a 2D array, `tolist` applies recursively:

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

The base case for this recursion is a 0D array:

```python
>>> a = np.array(1)
>>> list(a)
Traceback (most recent call last):
...
TypeError: iteration over a 0-d array
>>> a.tolist()
1
```

tostring(order='C')

A compatibility alias for `tobytes`, with exactly the same behavior.
Despite its name, it returns bytes not strs.

Deprecated since version 1.19.0.

**trace***(offset=0, axis1=0, axis2=1, dtype=None, out=None)***

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See also:

`numpy.trace` equivalent function

**transpose***(*)***

Returns a view of the array with axes transposed.

For a 1-D array this has no effect, as a transposed vector is simply the same vector. To convert a 1-D array into a 2D column vector, an additional dimension must be added. `np.atleast2d(a).T` achieves this, as does `a[:, np.newaxis].T` achieves this, as does `np.newaxis`. For a 2-D array, this is a standard matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

**Parameters**

`axes` (None, tuple of ints, or `n` ints) –

- None or no argument: reverses the order of the axes.
- tuple of ints: `i` in the `j`-th place in the tuple means `a`’s `i`-th axis becomes `a.transpose()`’s `j`-th axis.
- `n` ints: same as an `n`-tuple of the same ints (this form is intended simply as a “convenience” alternative to the tuple form)

**Returns**

`out` – View of `a`, with axes suitably permuted.

**Return type**

`ndarray`

See also:

`ndarray.T` Array property returning the array transposed.

`ndarray.reshape` Give a new shape to an array without changing its data.

**Examples**

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 0],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```
\texttt{var}(axis=None, dtype=None, out=None, ddof=0, keepdims=False)

Returns the variance of the array elements, along given axis.

Refer to \texttt{numpy.var} for full documentation.

See also:

\texttt{numpy.var} equivalent function

\texttt{view}(\texttt{[\[dtype\]}, \texttt{type}])

New view of array with the same data.

\underline{Note:} Passing \texttt{None} for \texttt{dtype} is different from omitting the parameter, since the former invokes \texttt{dtype(None)} which is an alias for \texttt{dtype('float_')}.

\underline{Parameters}

- \texttt{dtype} (\texttt{\textit{data-type} or ndarray sub-class, optional}) – Data-type descriptor of the returned view, e.g., float32 or int16. Omitting it results in the view having the same data-type as \texttt{a}. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the \texttt{type} parameter).

- \texttt{type} (\texttt{\textit{Python type}, optional}) – Type of the returned view, e.g., \texttt{ndarray} or matrix. Again, omission of the parameter results in type preservation.

\underline{Notes}

\texttt{a.view()} is used two different ways:

\texttt{a.view(some\_dtype) or a.view(dtype=some\_dtype)} constructs a view of the array’s memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

\texttt{a.view(ndarray\_subclass) or a.view(type=ndarray\_subclass)} just returns an instance of \texttt{ndarray\_subclass} that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For \texttt{a.view(some\_dtype)}, if \texttt{some\_dtype} has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of \texttt{a} (shown by \texttt{print(a)}). It also depends on exactly how \texttt{a} is stored in memory. Therefore if \texttt{a} is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

\underline{Examples}

```python
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])

Viewing array data using a different type and dtype:
```n
```python
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrix'>
```

Creating a view on a structured array so it can be used in calculations
Making changes to the view changes the underlying array

```python
>>> xv[0,1] = 20
>>> x
array([[1, 20], [3, 4]], dtype=[('a', 'i1'), ('b', 'i1')])
```

Using a view to convert an array to a recarray:

```python
>>> z = x.view(np.recarray)
>>> z.a
array([1, 3], dtype=int8)
```

Views share data:

```python
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```python
>>> x = np.array([[[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[[1, 2], [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
  ...  
ValueError: To change to a dtype of a different size, the array must be C-
˓→contiguous
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[[1, 2],
       [4, 5]], dtype=[('width', '<i2'), ('length', '<i2')])
```

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

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

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

**get_angle_mat** *(mol_subset=0, atom_subset=(None, None, None), get_r_max=False)*
Create and return an angle matrix for all molecules and atoms in this instance.

**Parameters**
• **mol_subset** *(slice, optional)* – 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[str], optional)* – 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** `np.ndarray[np.float64], shape (m, n, k, l)`

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

• **mol_subset** *(slice, optional)* – 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`.

### as_mol2 *(filename, mol_subset=0)*
Convert a `FOX.MultiMolecule` object into one or more .mol2 files.

Utilizes the `plams.Molecule.write` method.

**Parameters**

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

• **mol_subset** *(slice, optional)* – 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`.

### as_mol *(filename, mol_subset=0)*
Convert a `MultiMolecule` object into one or more .mol files.

Utilizes the `plams.Molecule.write` method.

**Parameters**

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

• **mol_subset** *(slice, optional)* – 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`.

### 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`.  

---

### 2.3. The MultiMolecule Class

55
- MultiMolecule.properties["comments"] is an iterable.

Parameters

- **filename** (path-like object) – The path+filename (including extension) of the to be created file.
- **mol_subset** (slice, optional) – 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.

**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, optional) – 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[Union[str, int]], optional) – 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

np.ndarray[np.float64], shape \((m, n, 3)\), optional

**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, optional) – 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[Union[str, int]], optional) – 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.

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

Convert this instance into a list of plams.Molecule.

Parameters

- **mol_subset** (slice, optional) – 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[Union[str, int]], optional) – 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

list[plams.Molecule]
classmethod from_Molecule(mol_list, subset=frozenset(['atoms']))

Construct a MultiMolecule instance from one or more PLAMS molecules.

Parameters

- **mol_list** (plams.Molecule or Sequence[plams.Molecule]) – A PLAMS molecule or list of PLAMS molecules.
- **subset** (Container[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 molecule constructed from mol_list.

Return type FOX.MultiMolecule
classmethod from_xyz(filename, bonds=None, properties=None, read_comment=False)

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** (path-like object) – The path+filename of an .xyz file.
- **bonds** (np.ndarray[np.int64], shape (k, 3)) – 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, optional) – 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.
- **read_comments** (bool) – If True, extract all comment lines from the passed .xyz file and store them under properties.comments.

Returns A molecule 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** (path-like object) – The path+filename of an KF binary file.
- **bonds** (np.ndarray[np.int64], shape (k, 3)) – 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 molecule constructed from filename.

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 \textit{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 \textit{et al} in \textit{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 $S_l$.
   $$p(k \leftarrow l) = \begin{cases} 1, & \Delta\varepsilon_{QM-MM}(S_k) \Delta\varepsilon_{QM-MM}(S_l) \\ 0, & \Delta\varepsilon_{QM-MM}(S_k) \Delta\varepsilon_{QM-MM}(S_l) \end{cases}$$  \hspace{1cm} (2.1)
3. The move is accepted if the new set of parameters, $S_l$, lowers the auxiliary error ($\Delta\varepsilon_{QM-MM}$) 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).
   $$\Delta\varepsilon_{QM-MM} = \frac{\sum_i^N \left| r_{QM}^i - r_{MM}^i \right|^2}{\sum_i^N r_{QM}^i} \hspace{1cm} (2.2)$$
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.
   $$\Delta\varepsilon_{QM-MM}(S_k) + \phi \hspace{0.5cm} \text{if} \hspace{0.5cm} \Delta\varepsilon_{QM-MM}(S_k) \Delta\varepsilon_{QM-MM}(S_l) \hspace{1cm} (2.3)$$
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.
   $$\phi_{k\omega} = \phi_{(k-1)\omega} + \gamma^{(\alpha_t - \pi_{(k-1)})} \hspace{1cm} \kappa = 1, 2, 3, ..., N \hspace{1cm} (2.4)$$

2.4.2 Parameters

```plaintext
param:
   charge:
      param: charge
      constraints:
         - '0.5 < Cd < 1.5'
         - '-0.5 > Se > -1.5'
         - '0 > O_1 > -1'
      Cd: 0.9768
      Se: -0.9768
      O_1: -0.47041
```

(continues on next page)
frozen:
    C_1: 0.4524
lennard_jones:
    - unit: kJ/mol
      param: epsilon
      Cd Cd: 0.3101
      Se Se: 0.4266
      Cd Se: 1.5225
      Cd O_1: 1.8340
      Se O_1: 1.6135
    - unit: nm
      param: sigma
      Cd Cd: 0.1234
      Se Se: 0.4852
      Cd Se: 0.2940
      Cd O_1: 0.2471
      Se O_1: 0.3526

psf:
    str_file: ligand.str
    ligand_atoms: [C, O, H]

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

job:
    molecule: .../mol.xyz

geometry_opt:
    template: qmflows.templates.geometry.specific.cp2k_mm
    settings:
      cell_parameters: [50, 50, 50]
      prm: .../ligand.prm

md:
    template: qmflows.templates.md.specific.cp2k_mm
    settings:
      cell_parameters: [50, 50, 50]
      prm: .../ligand.prm

A comprehensive overview of all available input parameters is provided in Monte Carlo Parameters.

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 caculations 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 the total auxiliary error is defined as the sum of the auxiliary errors of all individual PES descriptors, \( R \) (see (2.5)).

\[
\Delta \varepsilon_{QM-MM} = \sum_r R \Delta \varepsilon_r^{QM-MM} \tag{2.5}
\]

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
    kwarg:
      atom_subset: [Cd, Se, O]
  adf:
    func: FOX.MultiMolecule.init_adf
    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 `FOX.MultiMolecule` subclass.
- Keyword argument can be provided with the `kwarg` key. The `kwarg` key is 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:

```yaml
pes:
  numpy_sum:
    func: numpy.sum
    kwarg:
      axis: 0
```

This .yaml input, given a `MultiMolecule` instance `mol`, is equivalent to:
>>> import numpy
>>> from FOX import MultiMolecule

>>> func = numpy.sum
>>> kwargs = {'axis': 0}

>>> mol = MultiMolecule(...)

2.4.4 The param block

param:
  charge:
    param: charge
  constraints:
    - Cs == -0.5 * Br
    - 0 < Cs < 2
    - 1 < Pb < 3
    Cs: 1.000
    Pb: 2.000
  lennard_jones:
    - param: epsilon
      unit: kjmol
      Cs Cs: 0.1882
      Cs Pb: 0.7227
      Pb Pb: 2.7740
    - unit: nm
      param: sigma
      constraints: Cs Cs == Pb Pb
      Cs Cs: 0.60
      Cs Pb: 0.50
      Pb Pb: 0.60

The block 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 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 $k = 10\ kcal/mol$) for all C-C-C-C bonds:

param:
  torsion:
    param: k
    unit: kcalmol
    C C C C: 10

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.
Lastly, a number of constraints can be applied to the various parameters in the form of minima/maxima and fixed ratios. The special \$LIGAND string can herein be used as an alias representing all atoms within a single ligand. For example, when the formate anion is used as ligand (O2CH), \$LIGAND is equivalent to 2 * O + C + H.

**Note:** The charge parameter is unique in that the total molecular charge is always constrained; it will remain constant with respect to the initial charge of the system. It is the users responsibily to ensure that the initial charge is actually integer.

### 2.4.5 Parameter Guessing

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

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.
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]

2.5 Monte Carlo Parameters

2.5.1 Index

<table>
<thead>
<tr>
<th>param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>param.type</td>
<td>The type of parameter mapping.</td>
</tr>
<tr>
<td>param.move_range</td>
<td>The parameter move range.</td>
</tr>
<tr>
<td>param.func</td>
<td>The callable for performing the Monte Carlo moves.</td>
</tr>
<tr>
<td>param.kwargs</td>
<td>A dictionary with keyword arguments for param.func</td>
</tr>
<tr>
<td>param.validation.</td>
<td>Whether to allow parameters, that are explicitly specified, for absent atoms.</td>
</tr>
<tr>
<td>allow_non_existent</td>
<td></td>
</tr>
<tr>
<td>param.validation.</td>
<td>Check whether the net charge of the system is integer within a given tolerance.</td>
</tr>
<tr>
<td>charge_tolerance</td>
<td></td>
</tr>
<tr>
<td>param.validation.</td>
<td>Whether to enforce the constraints for the initial user-specified parameters.</td>
</tr>
<tr>
<td>enforce_constraints</td>
<td></td>
</tr>
<tr>
<td>param.block.param</td>
<td>The name of the forcefield parameter.</td>
</tr>
<tr>
<td>param.block.unit</td>
<td>The unit in which the forcefield parameters are expressed.</td>
</tr>
<tr>
<td>param.block.</td>
<td>A string or list of strings with parameter constraints.</td>
</tr>
<tr>
<td>constraints</td>
<td></td>
</tr>
<tr>
<td>param.block.guess</td>
<td>Estimate all non-specified forcefield parameters.</td>
</tr>
<tr>
<td>param.block.frozen</td>
<td>A sub-block with to-be frozen parameters.</td>
</tr>
</tbody>
</table>
### psf

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>psf.str_file</td>
<td>The path+filename to one or more stream file.</td>
</tr>
<tr>
<td>psf.rtf_file</td>
<td>The path+filename to one or more MATCH-produced rtf file.</td>
</tr>
<tr>
<td>psf.psi_file</td>
<td>The path+filename to one or more psf files.</td>
</tr>
<tr>
<td>psf.ligand_atoms</td>
<td>All atoms within a ligand.</td>
</tr>
</tbody>
</table>

### pes

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pes.block.func</td>
<td>The callable for performing the Monte Carlo moves.</td>
</tr>
<tr>
<td>pes.block.ref</td>
<td>A list of reference values for when <code>func</code> operates on <code>qmflows.Result</code> objects.</td>
</tr>
<tr>
<td>pes.block.kwargs</td>
<td>A dictionary with keyword arguments for <code>pes.block.func</code>.</td>
</tr>
</tbody>
</table>

### pes_validation

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pes_validation.block.func</td>
<td>The callable for performing the Monte Carlo validation.</td>
</tr>
<tr>
<td>pes_validation.block.ref</td>
<td>A list of reference values for when <code>func</code> operates on <code>qmflows.Result</code> objects.</td>
</tr>
<tr>
<td>pes_validation.block.kwargs</td>
<td>A dictionary with keyword arguments for <code>pes_validation.block.func</code>.</td>
</tr>
</tbody>
</table>

### job

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>job.type</td>
<td>The type of package manager.</td>
</tr>
<tr>
<td>job.molecule</td>
<td>One or more .xyz files with reference (QM) potential energy surfaces.</td>
</tr>
<tr>
<td>job.block.type</td>
<td>An instance of a QMFlows Package.</td>
</tr>
<tr>
<td>job.block.settings</td>
<td>The job settings as used by <code>job.block.type</code>.</td>
</tr>
<tr>
<td>job.block.template</td>
<td>A settings template for updating <code>job.block.settings</code>.</td>
</tr>
</tbody>
</table>

### monte_carlo

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>monte_carlo.type</td>
<td>The type of Monte Carlo procedure.</td>
</tr>
<tr>
<td>monte_carlo.iter_len</td>
<td>The total number of ARMC iterations $n \omega$.</td>
</tr>
<tr>
<td>monte_carlo.sub_iter_len</td>
<td>The length of each ARMC subiteration $\omega$.</td>
</tr>
<tr>
<td>monte_carlologfile</td>
<td>The name of the ARMC logfile.</td>
</tr>
<tr>
<td>monte_carlo.hdf5_file</td>
<td>The name of the ARMC .hdf5 file.</td>
</tr>
<tr>
<td>monte_carlo.path</td>
<td>The path to the ARMC working directory.</td>
</tr>
<tr>
<td>monte_carlo.folder</td>
<td>The name of the ARMC working directory.</td>
</tr>
<tr>
<td>monte_carlo.keep_files</td>
<td>Whether to keep all raw output files or not.</td>
</tr>
</tbody>
</table>

### phi

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>phi.type</td>
<td>The type of phi updater.</td>
</tr>
<tr>
<td>phi.gamma</td>
<td>The constant $\gamma$.</td>
</tr>
<tr>
<td>phi.a_target</td>
<td>The target acceptance rate $\alpha_t$.</td>
</tr>
<tr>
<td>phi.phi</td>
<td>The initial value of the variable $\phi$.</td>
</tr>
<tr>
<td>phi.func</td>
<td>The callable for updating $\phi$.</td>
</tr>
<tr>
<td>phi.kwARGS</td>
<td>A dictionary with keyword arguments for <code>phi.func</code>.</td>
</tr>
</tbody>
</table>
### 2.5.2 `param`

All forcefield-parameter related options.

This settings block accepts an arbitrary number of sub-blocks.

#### Examples

```python
param:
    type: FOX.armc.ParamMapping
    move_range:
        start: 0.005
        stop: 0.1
        step: 0.005
        ratio: null
    func: numpy.multiply
    kwargs: {}
    validation:
        allow_non_existent: False
        charge_tolerance: 0.01
        enforce_constraints: False

charge:
    param: charge
    constraints:
        - '0.5 < Cd < 1.5'
        - '-0.5 > Se > -1.5'
    Cd: 0.9768
    Se: -0.9768
    O_1: -0.47041
    frozen:
        C_1: 0.4524

lennard_jones:
    - unit: kjmol
      param: epsilon
      Cd Cd: 0.3101
      Se Se: 0.4266
      Cd Se: 1.5225
      frozen:
          guess: uff
    - unit: nm
      param: sigma
      Cd Cd: 0.1234
      Se Se: 0.4852
      Cd Se: 0.2940
      frozen:
          guess: uff
```

--

**param**

Parameter

- **Type** - `str` or `FOX.armc.ParamMappingABC` subclass
• **Default Value** - "FOX.armc.ParamMapping"

The type of parameter mapping.
Used for storing and moving user-specified forcefield values.

**See Also**


---

**param.move_range**

Parameter

- **Type** - array-like or `dict`
- **Default Value** - 
  ```
  {"start": 0.005, "stop": 0.1, "step": 0.005, "ratio": None}
  ```

The parameter move range.
This value accepts one of the following two types of inputs:

1. A list of allowed moves (e.g. `[0.9, 0.95, 1.05, 1.0]`).

2. **A dictionary with the **start**, **stop** and **step** keys.** For example, the list in 1. can be reproduced with ```
  {"start": 0.05, "stop": 0.1, "step": 0.05, "ratio": None}
  ```

When running the ARMC parallel procedure `monte_carlo.type = FOX.armc.ARMCPT` option 1. should be supplied as a nested list (e.g. `[[0.9, 0.95, 1.05, 1.0], [0.8, 0.9, 1.1, 1.2]]`) and option 2. requires the additional "ratio" keyword (e.g. `[1, 2]`).

---

**param.func**

Parameter

- **Type** - `str` or `Callable[[np.ndarray, np.ndarray], np.ndarray]`
- **Default Value** - "numpy.multiply"

The callable for performing the Monte Carlo moves.
The passed callable should be able to take two NumPy arrays as arguments and return a new one.

**See Also**

`numpy.multiply()` Multiply arguments element-wise.

---

**param.kwarg**

Parameter

- **Type** - `dict[str, object]`
- **Default Value** - {}

A dictionary with keyword arguments for `param.func`.

---

**param.validation.allow_non_existent**

Parameter
• **Type**: bool

  **Default Value**: False

  Whether to allow parameters, that are explicitly specified, for absent atoms.

  This check is performed once, before the start of the ARMC procedure.

`param.validation.charge_tolerance`

  Parameter

  • **Type**: float

  • **Default Value**: 0.01

  Check whether the net charge of the system is integer within a given tolerance.

  This check is performed once, before the start of the ARMC procedure. Setting this parameter to `inf` disables the check.

`param.validation.enforce_constraints`

  Parameter

  • **Type**: bool

  • **Default Value**: False

  Whether to enforce the constraints for the initial user-specified parameters.

  This option checks if the initially supplied parameters are compatible with all the supplied constraints; an error will be raised if this is not the case. Note that the constraints will always be enforced once the actual ARMC procedure starts.

`param.block.param`

  Parameter

  • **Type**: str

  The name of the forcefield parameter.

  **Important**: Note that this option has no default value; one *must* be provided by the user.

`param.block.unit`

  Parameter

  • **Type**: str

  The unit in which the forcefield parameters are expressed.

  See the CP2K manual for a comprehensive list of all available units.

`param.block.constraints`

  Parameter

  • **Type**: str or list[str]

  A string or list of strings with parameter constraints. Accepted types of constraints are minima/maxima (e.g. $2 > Cd > 0$) and fixed parameter ratios (e.g. $Cd == -1 * Se$). The special `$LIGAND` alias can be used for representing all atoms within a single ligand. For example, `$LIGAND` is equivalent to $2 * O + C + H$ in the case of formate.

`param.block.guess`
Parameter

- **Type** - `dict[str, str]`

Estimate all non-specified forcefield parameters.

If specified, expects a dictionary with the "mode" key, e.g. `{"mode": "uff"}` or `{"mode": "rdf"}`.

**param.block.frozen**

Parameter

- **Type** - `dict`

A sub-block with to-be frozen parameters.

Parameters specified herein will be treated as constants rather than variables. Accepts forcefield parameters (e.g. "Cd Cd" = 1.0) and, optionally, the `guess` key.

### 2.5.3 psf

Settings related to the construction of protein structure files (.psf).

Note that the `psf.str_file`, `psf.rtf_file` and `psf.psf_file` options are all mutually exclusive; only one should be specified. Furthermore, this block is completely optional.

**Examples**

```yaml
psf:
  rtf_file: ligand.rtf
  ligand_atoms: [C, O, H]
```

**psf.str_file**

Parameter

- **Type** - `str` or `list[str]`

- **Default Value** - `None`

The path+filename to one or more stream files.

Used for assigning atom types and charges to ligands.

**psf.rtf_file**

Parameter

- **Type** - `str` or `list[str]`

- **Default Value** - `None`

The path+filename to one or more MATCH-produced rtf files.

Used for assigning atom types and charges to ligands.

**psf.psf_file**
Parameter

- **Type** - *str* or *list[str]*
- **Default Value** - *None*

The path+filename to one or more psf files. Used for assigning atom types and charges to ligands.

**psf.ligandAtoms**

Parameter

- **Type** - *str* or *list[str]*
- **Default Value** - *None*

A list with all atoms within the organic ligands. Used for defining residues.

### 2.5.4 pes

Settings to the construction of potentialy energy surface (PES) descriptors. This settings block accepts an arbitrary number of sub-blocks, each containing the *func* and, optionally, *kwargs* keys.

**Examples**

```python
pes:
    rdf:
        func: FOX.MultiMolecule.init_rdf
        kwargs:
            atom_subset: [Cd, Se, O]
    adf:
        func: FOX.MultiMolecule.init_adf
        kwargs:
            atom_subset: [Cd, Se]
    energy:
        func: FOX.properties.get_attr # i.e. `qmflows.Result(...).energy`
        ref: [-17.0429775897]
        kwargs:
            name: energy
    hirshfeld_charges:
        func: FOX.properties.call_method # i.e. `qmflows.Result(...).get_hirshfeld_charges()
        ref:
            [-0.1116, 0.1930, -0.1680, -0.2606, 0.1702, 0.0598, 0.0575, 0.0598]
        kwargs:
            name: get_hirshfeld_charges
```

**pes.block.func**

Parameter
• Type - str, Callable[[FOX.MultiMolecule], ArrayLike] or Callable[[qmflows.Result], ArrayLike]

A callable for constructing a PES descriptor.

The callable should return an array-like object and, as sole positional argument, take either a FOX. MultiMolecule or qmflows.Results instance. In the latter case one must supply a list of reference PES-descriptor-values to pes.block.ref.

**Important:** Note that this option has no default value; one must be provided by the user.

See Also

FOX.MultiMolecule.init_rdf() Initialize the calculation of radial distribution functions (RDFs).

FOX.MultiMolecule.init_adf() Initialize the calculation of angular distribution functions (ADFs).

pes.block.ref

Parameter

• Type - list[ArrayLike] or None

• Default Value - None

A list of reference values for when func operates on qmflows.Result objects.

If not None, a list of array_like objects must be supplied here, one equal in length to the number of supplied molecules (see job.molecule).

pes.block.kwars

Parameter

• Type - dict[str, object]

• Default Value - {}  

A dictionary with keyword arguments for func.

### 2.5.5 pes_validation

Settings to the construction of potentialy energy surface (PES) validators.

Functions identically w.r.t. to the pes block, the exception being that PES descriptors calculated herein are do not affect the error; they are only calculated for the purpose of validation.

This settings block accepts an arbitrary number of sub-blocks, each containing the func and, optionally, kwars keys.

**Examples**

```python
pes_validation:
  adf:
    func: FOX.MultiMolecule.init_adf
    kwars:
      atom_subset: [Cd, Se]
```
mol_subset: `!!python/object/apply:builtins.slice`  # i.e. slice(None, None, 10)
- null
- null
- 10

```

pes_validation.block.func

Parameter

• **Type**: `str` or `Callable[[FOX.MultiMolecule], ArrayLike]

A callable for constructing a PES validators.

The callable should return an array-like object and, as sole positional argument, take either a `FOX.MultiMolecule` or `qmflows.Results` instance. In the latter case one **must** supply a list of reference PES-descriptor-values to `pes_validation.block.ref`.

The structure of this block is identical to its counterpart in `pes.block.func`.

**Important**: Note that this option has no default value; one **must** be provided by the user.

See Also

* `FOX.MultiMolecule.init_rdf()` Initialize the calculation of radial distribution functions (RDFs).
* `FOX.MultiMolecule.init_adf()` Initialize the calculation of angular distribution functions (ADFs).

pes_validation.block.ref

Parameter

• **Type**: `list[ArrayLike]` or `None`
• **Default Value**: `None`

A list of reference values for when `func` operates on `qmflows.Result` objects.

If not `None`, a list of `array_like` objects must be supplied here, one equal in length to the number of supplied molecules (see `job.molecule`).

pes_validation.block.kwars

Parameter

• **Type**: `dict[str, object]`
• **Default Value**: `{}`

A dictionary with keyword arguments for `func`.

The structure of this block is identical to its counterpart in `pes.block.kwars`.

2.5. Monte Carlo Parameters
## 2.5.6 job

Settings related to the running of the various molecular mechanics jobs.

In addition to having two constant keys (*type* and *molecule*) this block accepts an arbitrary number of sub-blocks representing quantum and/or classical mechanical jobs. In the example above there are two of such sub-blocks: *geometry_opt* and *md*. The first step consists of a geometry optimization while the second one runs the actual molecular dynamics calculation. Note that these jobs are executed in the order as provided by the user-input.

### Examples

```python
job:
  type: FOX.armc.PackageManager
  molecule: .../mol.xyz

  geometry_opt:
    type: qmflows.cp2k_mm
    settings:
      prm: .../ligand.prm
      cell_parameters: [50, 50, 50]
      template: qmflows.templates.geometry.specific.cp2k_mm

  md:
    type: qmflows.cp2k_mm
    settings:
      prm: .../ligand.prm
      cell_parameters: [50, 50, 50]
      template: qmflows.templates.md.specific.cp2k_mm
```

### job.type

Parameter

- **Type** - *str* or *FOX.armc.PackageManagerABC* subclass
- **Default Value** - "FOX.armc.PackageManager"

The type of Auto-FOX package manager.

Used for managing and running the actual jobs.

### See Also


### job.molecule

Parameter

- **Type** - *str* or *list*[str]

One or more .xyz files with reference (QM) potential energy surfaces.
**Important:** Note that this option has no default value; one *must* be provided by the user.

**job.block.type**

Parameter

- **Type** - `str` or `qmflows.packages.Package` instance
- **Default Value** - "qmflows.cp2k_mm"

An instance of a QMFlows Package.

**See Also**

`qmflows.cp2k_mm` An instance of CP2KMM.

**job.block.settings**

Parameter

- **Type** - `dict` or `list[dict]`
- **Default Value** - `{}`

The job settings as used by `type`.

In the case of PES-averaged ARMC one can supply a list of dictionaries, each one representing the settings for its counterpart in `job.molecule`.

If a `template` is specified then this block may or may not be redundant, depending on its completeness.

**job.block.template**

Parameter

- **Type** - `dict` or `str`
- **Default Value** - `{}`

A Settings template for updating `settings`.

The template can be provided either as a dictionary or, alternatively, an import path pointing to a pre-existing dictionary. For example, "qmflows.templates.md.specific.cp2k_mm" is equivalent to import qmflows; template = qmflows.templates.md.specific.cp2k_mm.

**See Also**

`qmflows.templates.md` Templates for molecular dynamics (MD) calculations.

`qmflows.templates.geometry` Templates for geometry optimization calculations.
2.5.7 monte_carlo

Settings related to the Monte Carlo procedure itself.

Examples

```python
monte_carlo:
    type: FOX.armc.ARMC
    iter_len: 50000
    sub_iter_len: 10
    logfile: armc.log
    hdf5_file: armc.hdf5
    path: .
    folder: MM_MD_workdir
    keep_files: False
```

**monte_carlo.type**

Parameter

- **Type** - str or FOX.armc.MonteCarloABC subclass
- **Default Value** - "FOX.armc.ARMC"

The type of Monte Carlo procedure.

**See Also**

FOX.armc.ARMC The Addaptive Rate Monte Carlo class.

FOX.armc.ARMCPT An ARMC subclass implementing a parallel tempering procedure.

**monte_carlo.iter_len**

Parameter

- **Type** - int
- **Default Value** - 50000

The total number of ARMC iterations $\kappa \omega$.

**monte_carlo.sub_iter_len**

Parameter

- **Type** - int
- **Default Value** - 100

The length of each ARMC subiteration $\omega$.

**monte_carlo.logfile**

Parameter

- **Type** - str
• Default Value - "armc.log"

The name of the ARMC logfile.

**monte_carlo.hdf5_file** Parameter

• Type - str
• Default Value - "armc.hdf5"

The name of the ARMC .hdf5 file.

**monte_carlo.path** Parameter

• Type - str
• Default Value - "."

The path to the ARMC working directory.

**monte_carlo.folder** Parameter

• Type - str
• Default Value - "MM_MD_workdir"

The name of the ARMC working directory.

**monte_carlo.keep_files** Parameter

• Type - bool
• Default Value - "False"

Whether to keep all raw output files or not.

## 2.5.8 phi

Settings related to the ARMC $\phi$ parameter.

### Examples

```python
phi:
    type: FOX.armc.PhiUpdater
    gamma: 2.0
    a_target: 0.25
    phi: 1.0
    func: numpy.add
    kwargs: {}
```

**phi.type**
Parameter

- **Type**: `str` or `FOX.armc.PhiUpdaterABC` subclass
- **Default Value**: "FOX.armc.PhiUpdater"

The type of phi updater.
The phi updater is used for storing, keeping track of and updating $\phi$.

See Also

*FOX.armc.PhiUpdater* A class for applying and updating $\phi$.

### phi.gamma

Parameter

- **Type**: `float` or `list[float]`
- **Default Value**: 2.0

The constant $\gamma$.

See (2.4). Note that a list must be supplied when running the ARMC parallel tempering procedure

**(monte_carlo.type = FOX.armc.ARMCP)**

### phi.a_target

Parameter

- **Type**: `float` or `list[float]`
- **Default Value**: 0.25

The target acceptance rate $\alpha_t$.

See (2.4). Note that a list must be supplied when running the ARMC parallel tempering procedure

**(monte_carlo.type = FOX.armc.ARMCP)**

### phi.phi

Parameter

- **Type**: `float` or `list[float]`
- **Default Value**: 1.0

The initial value of the variable $\phi$.

See (2.3) and (2.4). Note that a list must be supplied when running the ARMC parallel tempering procedure

**(monte_carlo.type = FOX.armc.ARMCP)**

### phi.func

Parameter

- **Type**: `str` or `Callable[[float, float], float]`
- **Default Value**: "numpy.add"

The callable for updating $\phi$.
The passed callable should be able to take two floats as arguments and return a new float.

See Also
**numpy.add()**  Add arguments element-wise.

**phi.kwarg**

Parameter

- **Type**: dict
- **Default Value**: {}

A dictionary with further keyword arguments for *phi.func*.

### 2.6 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 `FOX.MultiMolecule` instance.

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

>>> mol = MultiMolecule.from_xyz(example_xyz)

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

### 2.6.1 API

`FOX.io.read_xyz.read_multi_xyz(filename, return_comment=True, unit='angstrom')`

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.
- **unit** *(str)* – The unit of the to-be returned array.

**Returns**

- *m* *n* *3* `np.ndarray` *(np.float64, dict [str, list [int]])* and
- (optional) *m* `np.ndarray` *(str)* –
  - 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, read_comment=False)`

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** (path-like object) – The path+filename of an .xyz file.
- **bonds** (np.ndarray[np.int64], shape (k, 3)) – 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, optional) – 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.
- **read_comments** (bool) – If True, extract all comment lines from the passed .xyz file and store them under properties.comments.

Returns A molecule constructed from filename.

Return type FOX.MultiMolecule

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.

### 2.7 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 = 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)
            sigma (Angstrom)  epsilon (kj/mol)
Atom pairs
Cd Cd         3.95          2.097554
Cd Se         2.50          4.759017
Cd O          2.20          3.360966
Se Se         4.20          2.976106
Se O          3.65          0.992538
O O           2.15          6.676584
```
2.7.1 Index

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>estimate_lj()</code></td>
<td>Estimate the Lennard-Jones ( \sigma ) and ( \varepsilon ) parameters using an RDF.</td>
</tr>
<tr>
<td><code>get_free_energy()</code></td>
<td>Convert a distribution function into a free energy function.</td>
</tr>
</tbody>
</table>

2.7.2 API

FOX.ff.lj_param.\texttt{estimate}_\texttt{lj}(\texttt{rdf}, \texttt{temperature}=298.15, \texttt{sigma}_\texttt{estimate}=\textquote{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:

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

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 (pandas.DataFrame)} – A radial distribution function. The columns should consist of atom-pairs.
- \texttt{temperature (float)} – The temperature in Kelvin.
- \texttt{sigma_estimate (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.\texttt{get_free}_\texttt{energy}(\texttt{distribution}, \texttt{temperature}=298.15, \texttt{unit}=\textquote{kcal/mol}, \texttt{inf_replace}=\textquote{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:

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

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

Parameters
• **distribution** (*array-like*) – A distribution function (e.g. an RDF) as an array-like object.

• **temperature** (*float*) – The temperature in Kelvin.

• **inf_replace** (*float*, optional) – A value used for replacing all instances of infinity (np.inf).

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

**Returns** An array-like object with a free-energy function (kJ/mol) of `distribution`.

**Return type** `pandas.DataFrame`

**See also:**

`MultiMolecule.init_rdf()` Initialize the calculation of radial distribution functions (RDFs).

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

## 2.8 PSFContainer

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

### 2.8.1 Index

**PSFContainer([filename, title, atoms, ...])**

A container for managing protein structure files.

### 2.8.2 API

**class** `FOX.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()

**filename**
A 1D array with a single string as filename.

  Type: 1 * numpy.ndarray[|str|

**title**
A 1D array of strings holding the title block.

  Type: n * numpy.ndarray[|str|

**atoms**
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"

  Type: n * 8 * pandas.DataFrame

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

  Type: n * 2 * numpy.ndarray[int]

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

  Type: n * 3 * numpy.ndarray[int]

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

  Type: n * 4 * numpy.ndarray[int]

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

  Type: n * 4 * numpy.ndarray[int]

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

  Type: n * 1 * numpy.ndarray[int]

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

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

Type \( n \times 2 \) numpy.ndarray[int]

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

Type Mapping[str,object]

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

Type Mapping[str,object]

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

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

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

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

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

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

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

property impropers
Get PSFContainer.impropers or assign an array-like object as a 2D array.
property donors
Get `PSFContainer.donors` or assign an array-like object as a 2D array.

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

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

property segment_name
Get or set the "segment name" column in `PSFContainer.atoms`.

property residue_id
Get or set the "residue ID" column in `PSFContainer.atoms`.

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

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

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

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

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

update_atom_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`.

update_atom_type
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"].

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

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

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

Parameters `mol` (plams.Molecule) – A PLAMS Molecule.
generate_dihedrals(mol)
Update PSFContainer.dihedrals with the indices of all proper dihedral angle-defining atoms from mol.

Parameters
- **mol** (*plams.Molecule*) – A PLMS Molecule.

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

Parameters
- **mol** (*plams.Molecule*) – A PLMS Molecule.

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>Atom.properties [&quot;pdb_info&quot;][&quot;ResidueNumber&quot;]</td>
<td>1</td>
</tr>
<tr>
<td>&quot;residue name&quot;</td>
<td>Atom.properties [&quot;pdb_info&quot;][&quot;ResidueName&quot;]</td>
<td>&quot;COR&quot;</td>
</tr>
<tr>
<td>&quot;atom name&quot;</td>
<td>Atom.symbol</td>
<td></td>
</tr>
<tr>
<td>&quot;atom type&quot;</td>
<td>Atom.properties [&quot;symbol&quot;]</td>
<td>Atom.symbol</td>
</tr>
<tr>
<td>&quot;charge&quot;</td>
<td>Atom.properties [&quot;charge_float&quot;]</td>
<td>Atom.properties [&quot;charge&quot;] &amp; 0.0</td>
</tr>
<tr>
<td>&quot;mass&quot;</td>
<td>Atom.mass</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 PLMS Molecule.
- **id_map** (Mapping[int, Any], 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.

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 PLMS Molecule.
- **copy_mol** (bool) – If True, create a copy of mol instead of modifying it inplace.
• **pdb_file** *(str or IO[str])* – A filename or a file-like object.

## 2.9 PRMContainer

A class for reading and generating .prm parameter files.

### 2.9.1 Index

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRMContainer([atoms, bonds, angles,...])</td>
<td>A class for managing prm files.</td>
</tr>
<tr>
<td>PRMContainer.read(file[, bytes_decoding])</td>
<td>Construct a new instance from this object’s class by reading the content of file.</td>
</tr>
<tr>
<td>PRMContainer.write([file, bytes_encoding])</td>
<td>Write the content of this instance to file.</td>
</tr>
<tr>
<td>PRMContainer.overlay_mapping(prm_name, param)</td>
<td>Update a set of parameters, <strong>prm_name</strong>, with those provided in <strong>param_df</strong>.</td>
</tr>
<tr>
<td>PRMContainer.overlay_cp2k_settings(cp2k_settings)</td>
<td>Extract forcefield information from PLAMS-style CP2K settings.</td>
</tr>
</tbody>
</table>

### 2.9.2 API

**class** FOX.PRMContainer *(atoms=None, bonds=None, angles=None, dihedrals=None, impropers=None, nbfix=None, hbond=None, nonbonded_header=None, nonbonded=None, improper=None)*

A class for managing prm files.

**Examples**

```python
>>> from FOX import PRMContainer

>>> input_file = str(...)  
>>> output_file = str(...)  

>>> prm = PRMContainer.read(input_file)  
>>> prm.write(output_file)
```

- **impropers**
  A dataframe holding improper diehdral-related parameters.
- **atoms**
  A dataframe holding atomic parameters.
- **bonds**
  A dataframe holding bond-related parameters.
- **angles**
  A dataframe holding angle-related parameters.
- **dihedrals**
  A dataframe holding proper dihedral-related parameters.
- **nonbonded**
  A dataframe holding non-bonded atomic parameters.
**nbfix**

A dataframe holding non-bonded pair-wise atomic parameters.

**classmethod PRMContainer.read**(file, bytes_decoding=None, **kwargs)

Construct a new instance from this object’s class by reading the content of file.

**Parameters**

- **file**(str, bytes, os.PathLike or IO) – A path- or file-like object.
- **bytes_decoding**(str, optional) – The type of encoding to use when reading from file when it will be/is be opened in bytes mode. This value should be left empty otherwise.
- ****kwargs**(Any) – Further keyword arguments for open(). Only relevant if file is a path-like object.

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

**Return type** nanoutils.AbstractFileContainer

**PRMContainer.write**(file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>, bytes_encoding=None, **kwargs)

Write the content of this instance to file.

**Parameters**

- **file**(str, bytes, os.PathLike or IO) – A path- or file-like object. Defaults to sys.stdout if not specified.
- **bytes_encoding**(str, optional) – The type of encoding to use when writing to file when it will be/is be opened in bytes mode. This value should be left empty otherwise.
- ****kwargs**(Any) – Further keyword arguments for open(). Only relevant if file is a path-like object.

**Return type** None

**PRMContainer.overlay_mapping**(prm_name, param, 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** *(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"].

PRMContainer.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
global cp2kparser  # https://github.com/nlesc-nano/CP2K-Parser
global filename = str(...)  >>> cp2k_settings: dict = cp2kparser.read_input(filename)
```

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

### 2.10 Properties

Functions for calculating/extracting various properties.

Each function can be used to calculate the respective property as is, or to extract it from a passed qmflows.Result instance.

```python
>>> from FOX.properties import get_bulk_modulus
>>> from qmflows.packages import Result
>>> import numpy as np

```
An example for how \texttt{get\_bulk\_modulus()} can be used in conjunction with the \textit{ARMC yaml input}. Note that additional CP2K \texttt{print} keys are required in order for it to export the necessary properties.

\begin{verbatim}
job:
   type: FOX.armc.PackageManager
   molecule: mol.xyz

   md:
      template: qmflows.md.specific.cp2k_mm
      settings:
         cell_parameters: [50, 50, 50]
         input:
            motion:
               print:
                  cell on:
                     filename: ''
                  forces on:
                     filename: ''
            md:
               ensemble: NVE
               thermostat:
                  print:
                     temperature on:
                        filename: ''

pes:
   bulk_modulus:
      func: FOX.properties.get_bulk_modulus.from_result
      ref: [1.0]
      kwargs:
         reduce: mean
\end{verbatim}

2.10.1 Index

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{FromResult(func, name[, module, doc])}</td>
<td>A class for wrapping \texttt{Callable} objects.</td>
</tr>
<tr>
<td>\texttt{get_attr(obj, name[, default, reduce, axis])}</td>
<td>\texttt{getattr()} with support for keyword argument.</td>
</tr>
<tr>
<td>\texttt{call_method(obj, name, *args[, reduce, axis])}</td>
<td>Call the \texttt{name} method of \texttt{obj}.</td>
</tr>
<tr>
<td>\texttt{get_pressure(forces, coords, volume[...])}</td>
<td>Calculate the pressure from the passed \texttt{forces}.</td>
</tr>
<tr>
<td>\texttt{get_bulk_modulus(pressure, volume[...])}</td>
<td>Calculate the bulk modulus via differentiation of pressure w.r.t. \texttt{volume}.</td>
</tr>
</tbody>
</table>
2.10.2 API

class FOX.properties.FromResult (func, name, module=None, doc=None)
An abstract base class for wrapping Callable objects.

Besides __call__(), instances have access to the from_result() method, which is used for applying the wrapped callable to a qmflows.Result instance.

Parameters
• func(Callable[..., Any]) – The to-be wrapped function.
• name(str) – The __name__ attribute of the to-be created instance.
• module(str) – The __module__ attribute of the to-be created instance. If None, set it to "__main__".
• doc(str, optional) – The __doc__ attribute of the to-be created instance. If None, extract the docstring from func.

REDUCTION_NAMES : Mapping[str, Callable[[np.ndarray], np.float64]] = ...  
A mapping that maps from_result() aliases to callbacks.

In addition to the examples below, all reducable ufuncs from numpy and scipy.special are available.

```
>>> from types import MappingProxyType
>>> import numpy as np
>>> import scipy.special

>>> REDUCTION_NAMES = MappingProxyType({
...     'min': np.min,
...     'max': np.max,
...     'mean': np.mean,
...     'sum': np.sum,
...     'product': np.product,
...     'var': np.var,
...     'std': np.std,
...     'ptp': np.ptp,
...     'norm': np.linalg.norm,
...     'argmin': np.argmin,
...     'argmax': np.argmax,
...     'all': np.all,
...     'any': np.any,
...     'add': np.add.reduce,
...     'eval_legendre': scipy.special.eval_legendre.reduce,
...     ...
... })
```

property __call__
Get the underlying function.

abstract from_result (result, reduce=None, axis=None, *, return_unit=NotImplemented, **kwargs)
Call self using argument extracted from result.

Parameters
• result(qmflows.Result) – The Result instance that self should operator on.
• reduce(str or Callable[[Any], Any], optional) – A callback for reducing the output of self. Alternatively, one can provide on of the string aliases from REDUCTION_NAMES.
• **axis** (int or Sequence[int], optional) – The axis along which the reduction should take place. If None, use all axes.
• **return_unit** (str) – The unit of the to-be returned quantity.
• **kwargs** (Any) – Further keyword arguments for __call__().

Returns The output of __call__().

Return type Any

FOX.properties.get_attr(obj, name, default=<null>, reduce=None, axis=None)

getattr() with support for keyword argument.

Parameters

• **obj** (object) – The object in question.
• **name** (str) – The name of the to-be extracted attribute.
• **default** (Any) – An object that is to-be returned if obj does not have the name attribute.
• **reduce** (str or Callable[[Any], Any], optional) – A callback for reducing the extracted attribute. Alternatively, one can provide on of the string aliases from FromResult.REDUCTION_NAMES.
• **axis** (int or Sequence[int], optional) – The axis along which the reduction should take place. If None, use all axes.

Returns The extracted attribute.

Return type Any

See also:

getattr() Get a named attribute from an object.

FOX.properties.call_method(obj, name, *args, reduce=None, axis=None, **kwargs)

Call the name method of obj.

Parameters

• **obj** (object) – The object in question.
• **name** (str) – The name of the to-be extracted method.
• **args/**kwargs (Any) – Positional and/or keyword arguments for the (to-be called) extracted method.
• **reduce** (str or Callable[[Any], Any], optional) – A callback for reducing the output of the called function. Alternatively, one can provide on of the string aliases from FromResult.REDUCTION_NAMES.
• **axis** (int or Sequence[int], optional) – The axis along which the reduction should take place. If None, use all axes.

Returns The output of the extracted method.

Return type Any

FOX.properties.get_pressure(forces, coords, volume, temp=298.15, forces_unit='ha/bohr', coords_unit='bohr', volume_unit='bohr', return_unit='ha/bohr^3')

Calculate the pressure from the passed forces.

\[
P = \frac{N k_B T}{V} + \frac{1}{6V} \sum_i \sum_{j \neq i} r_{ij}^3 f_{ij}
\]
Parameters

- **forces** *(np.ndarray[np.float64], shape (n_mol, n_atom, 3)) – A 3D array containing the forces of all molecules within the trajectory.*

- **coords** *(np.ndarray[np.float64], shape (n_mol, n_atom, 3)) – A 3D array containing the coordinates of all molecules within the trajectory.*

- **volume** *(np.ndarray[np.float64], shape (n_mol,)) – A 1D array containing the cell volumes across the trajectory.*

- **temp** *(np.ndarray[np.float64], shape (n_mol,)) – A 1D array of the temperatures across the trajectory.*

- **forces_unit** *(str) – The unit of the forces.*

- **coords_unit** *(str) – The unit of the coords.*

- **volume_unit** *(str) – The unit of the volume. The passed unit will automatically cubed, e.g. Angstrom -> Angstrom**3.*

- **return_unit** *(str) – The unit of the to-be returned pressure.*

Returns A 1D array with all pressures across the trajectory.

Return type np.ndarray[np.float64], shape (n_mol,)

Note: Using `get_pressure.from_result()` requires the passed qmflows.CP2K_Result to have access to the following files for each argument:

- **forces**: cp2k-frc-1.xyz
- **coords**: cp2k-pos-1.xyz
- **volume**: cp2k-1.cell
- **temp**: cp2k-1.ener

FOX.properties.get_bulk_modulus(pressure, volume, pressure_unit='ha/bohr^3', volume_unit='bohr', return_unit='ha/bohr^3')

Calculate the bulk modulus via differentiation of pressure w.r.t. volume.

\[ B = -V \frac{\delta P}{\delta V} \]

Parameters

- **pressure** *(np.ndarray[np.float64]) – A 1D array of pressures used for defining \( \delta P \). Must be of equal length as volume.*

- **volume** *(np.ndarray[np.float64]) – A 1D array of volumes used for defining \( \delta V \). Must be of equal length as pressure.*

- **pressure_unit** *(str) – The unit of the pressure.*

- **volume_unit** *(str) – The unit of the volume. The passed unit will automatically cubed, e.g. Angstrom -> Angstrom**3.*

- **return_unit** *(str) – The unit of the to-be returned pressure.*

Returns The bulk modulus \( B \). Returned as either a scalar or array, depending on the dimensionality volume_ref.

Return type np.float64 or np.ndarray[np.float64]
Note: Using `getBulkModulus/from_result()` requires the passed `qmflows.CP2K_Result` to have access to the following files for each argument:

- **pressure**: cp2k-frc-1.xyz, cp2k-pos-1.xyz, cp2k-1.cell & cp2k-1.ener
- **volume**: cp2k-1.cell

Furthermore, in order to get sensible results both the pressure and cell volume must be variable.

## 2.11 Recipes

Various recipes implemented in Auto-FOX.

<table>
<thead>
<tr>
<th>FOX.recipes.param</th>
<th>A set of functions for analyzing and plotting ARMC results.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOX.recipes.psf</td>
<td>A set of functions for creating .psf files.</td>
</tr>
<tr>
<td>FOX.recipes.ligands</td>
<td>A set of functions for analyzing ligands.</td>
</tr>
<tr>
<td>FOX.recipes.time_resolution</td>
<td>A set of functions for calculating time-resolved distribution functions.</td>
</tr>
</tbody>
</table>

### 2.11.1 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)
```

![Graphs](image)
A small workflow for calculating 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'
```
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()}
```

Index

get_best(hdf5_file, name[, i, err_dset]) Return the PES descriptor or ARMC property which yields the lowest error.

overlay_descriptor(hdf5_file[, name, i, ...]) Return the PES descriptor which yields the lowest error and overlay it with the reference PES descriptor.

plot_descriptor(descriptor[, show_fig, ...]) Plot a DataFrame or iterable consisting of one or more DataFrames.

API

FOX.recipes.get_best(hdf5_file, name, i=0, err_dset='aux_error') 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.
- **err_dset** (str) – The name of the dataset containing the errors. Generally speaking one should pick either "aux_error" or "validation/aux_error".

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

Return type pandas.DataFrame or pd.Series
FOX.recipes.overlay_descriptor(hdf5_file, name='rdf', i=0, err_dset='aux_error')

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.
• err_dset (str) – The name of the dataset containing the errors. Generally speaking one should pick either "aux_error" or "validation/aux_error".

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]

FOX.recipes.plot_descriptor(descriptor, show_fig=True, kind='line', sharex=True, sharey=False, **kwargs)

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.

2.11.2 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.
>>> 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(...)
>>> ligand: Molecule = Molecule(...)
>>> rtf_file : str = ...
>>> psf_file : str = ...

>>> 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 ($C_{18}H_{33}O_2$).

>>> 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.

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

>>> ...

(continues on next page)
>> 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.generate_psf** *(qd, ligand, rtf_file=None, 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.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. Usefull for debugging. If False, raise a MoleculeError.

---

2.11. Recipes
**FOX.recipes.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

### 2.11.3 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.
Or the potential of mean force (i.e. Boltzmann-inverted RDF).

```python
>>> 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
>>> keep_lig = [0, 1, 2, 3]  # Keep these ligands; discard the rest
>>> lig_centra_subset = lig_centra[:, keep_lig]

# Add dummy atoms to the ligand-center of mass and calculate the RDF
>>> mol_new2: MultiMolecule = mol.add_atoms(lig_centra_subset, symbols='Xx')
>>> rdf: pd.DataFrame = mol_new2.init_rdf(atom_subset=['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))
>>> del idx_dict[1]  # Delete the core

# 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 = np.ndarray = get_multi_lig_center(mol, idx_dict.values())
>>> mol_new = MultiMolecule = mol.add_atoms(lig_centra, symbols=symbols)
>>> rdf = mol_new.init_rdf(atom_subset=set(symbols))
```

Index

<table>
<thead>
<tr>
<th>Method</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 <code>mol</code>.</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 <code>mol</code>.</td>
</tr>
</tbody>
</table>

API

`FOX.recipes.get_lig_center` (mol, start, step=None, stop=None, mass_weighted=True)

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 centre 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`
FOX.recipes.get_multi_lig_center(mol, idx_iter, mass_weighted=True)

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

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 center 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

2.11.4 FOX.recipes.time_resolution

A set of functions for calculating time-resolved distribution functions.

Index

| time_resolved_rdf(mol[, start, stop, step]) | Calculate the time-resolved radial distribution function (RDF). |
| time_resolved_adf(mol[, start, stop, step]) | Calculate the time-resolved angular distribution function (ADF). |

API

FOX.recipes.time_resolved_rdf(mol[, start=0, stop=None, step=500, **kwargs])

Calculate the time-resolved radial distribution function (RDF).

Examples

```python
>>> from FOX import MultiMolecule, example_xyz
>>> from FOX.recipes import time_resolved_rdf

# Calculate each RDF over the course of 500 frames
>>> time_step = 500
>>> mol = MultiMolecule.from_xyz(example_xyz)

>>> rdf_list = time_resolved_rdf(
...     mol, step=time_step, atom_subset=['Cd', 'Se']
... )
```

Parameters

• mol (MultiMolecule) – The trajectory in question.
• `start` (int) – The initial frame.
• `stop` (int, optional) – The final frame. Set to `None` to iterate over all frames.
• `step` (int) – The number of frames per individual RDF. Note that lower `step` values will result in increased numerical noise.
• `**kwargs` (Any) – Further keyword arguments for `init_rdf()`.

Returns A list of dataframes, each containing an RDF calculated over the course of `step` frames.

Return type `List[pandas.DataFrame]`

See also:

`init_rdf()` Calculate the radial distribution function.

FOX.recipes.time_resolved_adf(mol, start=0, stop=None, step=500, **kwargs)
Calculate the time-resolved angular distribution function (ADF).

Examples

```python
>>> from FOX import MultiMolecule, example_xyz
>>> from FOX.recipes import time_resolved_adf

# Calculate each ADF over the course of 500 frames
>>> from FOX import MultiMolecule, example_xyz
>>> mol = MultiMolecule.from_xyz(example_xyz)

>>> rdf_list = time_resolved_adf(...
    mol, step=time_step, atom_subset=['Cd', 'Se'])
```

Parameters

• `mol` (`MultiMolecule`) – The trajectory in question.
• `start` (int) – The initial frame.
• `stop` (int, optional) – The final frame. Set to `None` to iterate over all frames.
• `step` (int) – The number of frames per individual RDF. Note that lower `step` values will result in increased numerical noise.
• `**kwargs` (Any) – Further keyword arguments for `init_adf()`.

Returns A list of dataframes, each containing an ADF calculated over the course of `step` frames.

Return type `List[pandas.DataFrame]`

See also:

`init_adf()` Calculate the angular distribution function.
2.12 cp2k_to_prm

A TypedMapping subclass converting CP2K settings to .prm-compatible values.

2.12.1 Index

<table>
<thead>
<tr>
<th>PRMMapping(*args, **kwargs)</th>
<th>A TypedMapping providing tools for converting CP2K settings to .prm-compatible values.</th>
</tr>
</thead>
</table>

2.12.2 API

class FOX.io.cp2k_to_prm.PRMMapping(*args, **kwargs)  
A TypedMapping providing tools for converting CP2K settings to .prm-compatible values.

    name  
    The name of the PRMContainer attribute.  
    Type str

    columns  
    The names relevant PRMContainer DataFrame columns.  
    Type tuple[int]

    key_path  
    The path of CP2K Settings keys leading to the property of interest.  
    Type tuple[str]

    key  
    The key(s) within PRMMapping.key_path containing the actual properties of interest, e.g. "epsilon" and "sigma".  
    Type tuple[str]

    unit  
    The desired output unit.  
    Type tuple[str]

    default_unit  
    The default unit as utilized by CP2K.  
    Type tuple[str, optional]

    post_process  
    Callables for post-processing the value of interest. Set a particular callable to None to disable post-processing.  
    Type tuple[Callable[[float], float], optional]

FOX.io.cp2k_to_prm.CP2K_TO_PRM : MappingProxyType[str, PRMMapping]  
A Mapping containing PRMMapping instances.
2.13 Index

<table>
<thead>
<tr>
<th><code>ParamMappingABC</code></th>
<th>A <code>Mapping</code> for storing and updating forcefield parameters.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ParamMapping</code></td>
<td>A <code>Mapping</code> for storing and updating forcefield parameters.</td>
</tr>
</tbody>
</table>

2.14 API

```python
class FOX.armc.ParamMappingABC(data, move_range, func[, ...])
```

A `Mapping` for storing and updating forcefield parameters.

Besides the implementation of the `Mapping` protocol, this class has access to four main methods:

- `__call__()` or `move()` move a random parameter by a random step size.
- `identify_move()` identify the parameter and move step size.
- `clip_move()` clip the move.
- `apply_constraints()` apply further constraints to the move.

Note that `__call__()` will internally call all other three methods.

Examples

```python
>>> import pandas as pd
>>> df = pd.DataFrame(..., index=pd.MultiIndex(...))
>>> param = ParamMapping(df, ...)
>>> idx = param.move()
```

- `move_range`  
  An 1D array with all allowed move steps.
  Type: `np.ndarray[np.float64], shape (n,)`

- `func`  
  The callable used for applying $\phi$ to the auxiliary error. The callable should take an two floats as arguments and return a new float.
  Type: `Callable`

- `_net_charge`  
  The net charge of the molecular system. Only applicable if the "charge" is among the passed parameters.
  Type: `float`, optional

- `FILL_VALUE`: `ClassVar[Mapping[Literal[min, max, count, frozen, guess, unit], numpy.generic]]`
  Fill values for when optional keys are absent.

- `add_param(idx, value, **kwargs)`  
  Add a new parameter to this instance.

Parameters
• **idx** (tuple[str, str, str]) – The index of the new parameter. Must be compatible with pd.DataFrame.loc.

• **value** (float) – The value of the new parameter.

• **kwargs** (Any) – Values for ParamMappingABC.metadata.

abstract **identify_move** (param)
Identify the to-be moved parameter and the size of the move.

Parameters

• **param** (str) – The name of the parameter-containing column.

Returns
The index of the to-be moved parameter, it’s value and the size of the move.

Return type
tuple[tuple[str, str, str], float, float]

**clip_move** (idx, value)
An optional function for clipping the value of value.

Parameters

• **idx** (tuple[str, str, str]) – The index of the moved parameter.

• **value** (float) – The value of the moved parameter.

Returns
The newly clipped value of the moved parameter.

Return type
float

**apply_constraints** (idx, value, param)
An optional function for applying further constraints based on idx and value.

Should perform an inplace update of this instance.

Parameters

• **idx** (tuple[str, str, str]) – The index of the moved parameter.

• **value** (float) – The value of the moved parameter.

• **param** (str) – The name of the parameter-containing column.

Returns
Any exceptions raised during this functions’ call.

Return type
Exception, optional

**to_struct_array** ()
Stack all Series in this instance into a single structured array.

class FOX.armc.ParamMapping (data, move_range=array([[0.9, 0.905, 0.91, 0.915, 0.92, 0.925, 0.93, 0.935, 0.94, 0.945, 0.95, 0.955, 0.96, 0.965, 0.97, 0.975, 0.98, 0.985, 0.99, 0.995, 1.005, 1.01, 1.015, 1.02, 1.025, 1.03, 1.035, 1.04, 1.045, 1.05, 1.055, 1.06, 1.065, 1.07, 1.075, 1.08, 1.085, 1.09, 1.095, 1.1 ])), **kwargs)
A Mapping for storing and updating forcefield parameters.

Besides the implementation of the Mapping protocol, this class has access to four main methods:

• __call__() or move() move a random parameter by a random step size.

• identify_move() identify the parameter and move step size.

• clip_move() clip the move.

• apply_constraints() apply further constraints to the move.
Note that \_\_call\_\_() will internally call all other three methods.

Examples

```python
>>> import pandas as pd
>>> df = pd.DataFrame(..., index=pd.MultiIndex(...))
>>> param = ParamMapping(df, ...)
>>> idx = param.move()
```

**move_range**

An 1D array with all allowed move steps.

Type: np.ndarray[np.float64], shape \((n,)\)

**func**

The callable used for applying \(\phi\) to the auxiliary error. The callable should take two floats as arguments and return a new float.

Type: Callable

**net_charge**

The net charge of the molecular system. Only applicable if the "charge" is among the passed parameters.

Type: float, optional

**CHARGE_LIKE**

A set of charge-like parameters which require a parameter re-normalization after every move.

```
CHARGE_LIKE: ClassVar[FrozenSet[str]] = frozenset({'charge'})
```

**identify_move**(param_idx)

Identify and return a random parameter and move size.

Parameters:

- **param_idx** (int) – The name of the parameter-containing column.

Returns:

The index of the to-be moved parameter, its value and the size of the move.

Return type: tuple[tuple[str, str, str], float, float]

**clip_move**(idx, value)

Ensure that value falls within a user-specified range.

Parameters:

- **idx** (tuple[str, str, str]) – The index of the moved parameter.
- **value** (float) – The value of the moved parameter.

Returns:

The newly clipped value of the moved parameter.

Return type: float

**apply_constraints**(idx, value, param_idx)

Apply further constraints based on idx and value.

Perform an in-place update of this instance.

Parameters:

- **idx** (tuple[str, str, str]) – The index of the moved parameter.
- **value** (float) – The value of the moved parameter.
• **param_idx** *(int)* – The name of the parameter-containing column.

## 2.15 Index

<table>
<thead>
<tr>
<th>PackageManagerABC(data[, hook])</th>
<th>A class for managing qmflows-style jobs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PackageManager(data[, hook])</td>
<td>A class for managing qmflows-style jobs.</td>
</tr>
</tbody>
</table>

## 2.16 API

class FOX.armc.PackageManagerABC *(data, hook=None, **kwargs)*  
A class for managing qmflows-style jobs.

- **property hook**  
  Get or set the *hook* attribute.

- **property data**  
  A property containing this instance’s underlying dict.
  The getter will simply return the attribute’s value. The setter will validate and assign any mapping or iterable containing of key/value pairs.

- **keys()**  
  Return a set-like object providing a view of this instance’s keys.

- **items()**  
  Return a set-like object providing a view of this instance’s key/value pairs.

- **values()**  
  Return an object providing a view of this instance’s values.

- **get(key, default=None)**  
  Return the value for *key* if it’s available; return *default* otherwise.

- **abstract static assemble_job**(job, **kwargs)**  
  Assemble a PkgDict into an actual job.

- **abstract clear_jobs**(**kwargs)**  
  Delete all jobs located in _job_cache.

- **abstract update_settings**(dct, **kwargs)**  
  Update the Settings embedded in this instance using *dct*.

class FOX.armc.PackageManager *(data, hook=None)*  
A class for managing qmflows-style jobs.

- **static assemble_job**(job, old_results=None, name=None)  
  *(scheduled)* Create a PromisedObject from a qmflow Package instance.

- **static clear_jobs()**  
  Delete all jobs.

- **update_settings**(dct, new_keys=True)  
  Update all forcefield parameter blocks in this instance’s CP2K settings.
2.17 Index

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MonteCarloABC(molecule, package_manager, param)</code></td>
<td>The base <em>MonteCarloABC</em> class.</td>
</tr>
<tr>
<td><code>ARMC(phi[, iter_len, sub_iter_len])</code></td>
<td>The Addaptive Rate Monte Carlo class (<em>ARMC</em>).</td>
</tr>
<tr>
<td><code>ARMCPT([swapper])</code></td>
<td>An <em>ARMC</em> subclass implementing a parallel tempering procedure.</td>
</tr>
</tbody>
</table>

2.18 API

```python
class FOX.armc.MonteCarloABC(molecule, package_manager, param, keep_files=False, hdf5_file='armc.hdf5', logger=None, pes_post_process=None, **kwargs)
```

The base *MonteCarloABC* class.

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

- **property pes_post_process**
  Get or set post-processing functions.

- **property logger**
  Get or set the logger.

- **keys()**
  Return a set-like object providing a view of this instance’s keys.

- **items()**
  Return a set-like object providing a view of this instance’s key/value pairs.

- **values()**
  Return an object providing a view of this instance’s values.

- **get(key, default=None)**
  Return the value for key if it's available; return default otherwise.

- **add_pes_evaluator(name, func, args=(), kwargs=mappingproxy({}), validation=False, ref=None)**
  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`.
- **validation** *(bool)* – Whether the PES-descriptor is used exclusively for validation or
  not.

**property clear_jobs**
Delete all cp2k output files.

**run_jobs()**
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. Will return
`None` if one of the jobs crashed

**Return type** `list[FOX.MultiMolecule]`, optional

**move** *(idx=None)*
Update a random parameter in `self.param` by a random value from `self.move.range`.

Performs in-place 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
```

(continues on next page)
### Parameters `idx` (`int`, optional)
- The column key for `param_mapping["param"]`.

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

### Return type
- `tuple[float, ...]`

### `get_pes_descriptors` (*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 `get_first_key` (`bool`) – Keep both the files and the job_cache if this is the first ARMC iteration. Useful 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]]`, `dict[str, np.ndarray[np.float64]]` and `list[FOX.MultiMolecule]`

```python
class FOX.armc.ARMCArmc(phi, iter_len=50000, sub_iter_len=100, **kwargs)  # ARMC
The Addaptive Rate Monte Carlo class (ARMC).

A subclass of `MonteCarloABC`.

**iter_len**
The total number of ARMC iterations $\kappa \omega$.

Type `int`

**super_iter_len**
The length of each ARMC subiteration $\kappa$.

Type `int`

**sub_iter_len**
The length of each ARMC subiteration $\omega$.

Type `int`

**phi**
A PhiUpdater instance.

Type `PhiUpdaterABC`

**kwargs**
Keyword arguments for the `MonteCarlo` superclass.

Type `Any`
```
acceptance()

Create an empty 1D boolean array for holding the acceptance.


to_yaml_dict(*, path='.', folder='MM_MD_workdir', logfile='armc.log', psf=None)

Convert an ARMC instance into a .yaml readable by ARMC.from_yaml.

Returns A dictionary.

Return type dict[str, Any]

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__().
- **acceptance (np.ndarray[np.bool_])** – An array with the acceptance over the course of the latest super-iteration
- **key_new (tuple[float, ...])** – A tuple with the latest set of forcefield parameters.

Returns The latest set of parameters.

Return type tuple[float, ...]

property apply_phi

Apply \( \phi \) to value.

to_hdf5(mol_list, accept, aux_new, aux_validation, pes_new, pes_validation, kappa, omega)

Construct a dictionary with the hdf5_kwarg and pass it to to_hdf5().

Parameters

- **mol_list (list[FOX.MultiMolecule], optional)** – An iterable consisting molecules instances (or None).
- **accept (bool)** – Whether or not the latest set of parameters was accepted.
- **aux_new (np.ndarray[np.float64])** – The latest auxiliary error.
- **aux_validation (np.ndarray[np.float64])** – The latest auxiliary error from the validation.
- **pes_new (dict[str, np.ndarray[np.float64]])** – A dictionary of PES descriptors.
- **pes_validation (dict[str, np.ndarray[np.float64]])** – A dictionary of PES descriptors from the validation.
- **kappa (int)** – The super-iteration, \( \kappa \), in ARMC.__call__().
- **omega (int)** – The sub-iteration, \( \omega \), in ARMC.__call__().

Returns A dictionary with the hdf5_kwarg argument for to_hdf5().

Return type dict[str, Any]

get_aux_error(pes_dict, validation=False)

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

The auxiliary error is constructed using the PES descriptors in values with respect to self.ref.
The default function is equivalent to:
\[
\Delta \varepsilon_{QM-MM} = \sum_{i=1}^{N} \frac{|r_i^{QM} - r_i^{MM}|^2}{r_i^{QM}}
\]

Parameters:

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

Returns:
An array with \(m \times n\) auxiliary errors

Return type:
np.ndarray[np.float64], shape \((m, n)\)

```
restart()
```
Restart a previously started Addaptive Rate Monte Carlo procedure.

```
class FOX.armc.ARMCP (swapper=<function swap_random>, **kwargs)
```
An ARMC subclass implementing a parallel tempering procedure.

```
acceptance ()
```
Create an empty 2D boolean array for holding the acceptance.

```
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__().
- **acceptance** (np.ndarray[np.bool_]) – An array with the acceptance over the course of the latest super-iteration.
- **key_new** (tuple[float, ...]) – A tuple with the latest set of forcefield parameters.

Returns:
The latest set of parameters.

Return type:
tuple[float, ...]

```
to_yaml_dict (*, path='.', folder='MM_MD_workdir', logfile='armc.log', psf=None)
```
Convert an ARMC instance into a .yaml readable by ARMC.from_yaml.

Returns:
A dictionary.

Return type:
dict[str, Any]

### 2.19 Index

- **PhiUpdaterABC** (phi, gamma, a_target, func, ...) A class for applying and updating \(\phi\).
- **PhiUpdater** ([phi, gamma, a_target, func]) A class for applying and updating \(\phi\).
2.20 API

class FOX.armc.PhiUpdaterABC(\(\phi, \gamma, a\_target, func, **kwargs\))
A class for applying and updating \(\phi\).

Has two main methods:

- \_\_call\_\_\_() for applying \(\phi\) to the passed value.
- \_update() for updating the value of \(\phi\).

Examples

```python
>>> import numpy as np

>>> value = np.ndarray(...)  # ...
>>> phi = PhiUpdater(...)  # ...

>>> phi(value)
>>> phi.update(...)  
```

\(\phi\)

The variable \(\phi\).

Type `np.ndarray[np.float64]`

\(\gamma\)

The constant \(\gamma\).

Type `np.ndarray[np.float64]`

\(a\_target\)

The target acceptance rate \(\alpha_t\).

Type `np.ndarray[np.float64]`

\(func\)

The callable used for applying \(\phi\) to the auxiliary error. The callable should take an array-like object and a `numpy.ndarray` as arguments and return a new array.

Type `Callable[array-like, np.ndarray]`

property shape

Return the shape of \(\phi\).

Serves as a wrapper around the shape attribute of \(\phi\). Note that \(\phi, \gamma, a\_target\) all have the same shape.

to\_yaml\_dict()  
Convert this instance into a .yaml-compatible dictionary.

abstract update(acceptance, **kwargs)

An abstract method for updating \(\phi\) based on the values of \(\gamma\) and \(acceptance\).

Parameters

- **acceptance** (ArrayLike[np.bool_]) – An array-like object consisting of booleans.
- ****kwargs (Any) – Further keyword arguments which can be customized in the methods of subclasses.
class FOX.armc.PhiUpdater (phi=1.0, gamma=2.0, a_target=0.25, func=<ufunc 'add'>, **kwargs)
A class for applying and updating \(\phi\).

Has two main methods:

• __call__() for applying \(\phi\) to the passed value.
• update() for updating the value of \(\phi\).

Examples

```python
>>> import numpy as np
>>> value = np.ndarray(...)  # Example usage
>>> phi = PhiUpdater(...)  # Example usage
>>> phi(value)  # Example usage
>>> phi.update(...)  # Example usage
```

phi
The variable \(\phi\).

Type `np.ndarray[np.float64]`

gamma
The constant \(\gamma\).

Type `np.ndarray[np.float64]`

da_target
The target acceptance rate \(\alpha_t\).

Type `np.ndarray[np.float64]`

func
The callable used for applying \(\phi\) to the auxiliary error. The callable should take an array-like object and a `numpy.ndarray` as arguments and return a new array.

Type `Callable[[array-like, ndarray], ndarray]`

update (acceptance, *, logger=None)
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:

\[
\phi_{k+1}\omega = \phi_{(k-1)\omega} \* \gamma \text{sgn}(\alpha_t - \pi_{(k-1)})
\]

Parameters

• `acceptance` (ArrayLike[np.bool_]) – An array-like object consisting of booleans.
• `logger` (logging.Logger, optional) – A logger for reporting the updated value.
PYTHON MODULE INDEX

f
FOX.ff.lj_param, 78
FOX.io.cp2k_to_prm, 103
FOX.io.read prm, 85
FOX.io.read_psf, 80
FOX.properties, 87
FOX.recipes.ligands, 98
FOX.recipes.param, 92
FOX.recipes.psf, 95
FOX.recipes.time_resolution, 101
Symbols

__call__() (FOX.properties.FromResult property), 89
__net_charge (FOX.armc.ParamMapping attribute), 106
__net_charge (FOX.armc.ParamMappingABC attribute), 104

A

a_target (FOX.armc.PhiUpdater attribute), 114
a_target (FOX.armc.PhiUpdaterABC attribute), 113
a_target (phi attribute), 76
acceptance() (FOX.armc.ARMC method), 110
acceptance() (FOX.armc.ARMCPT method), 112
add_atoms() (FOX.MultiMolecule method), 17
add_param() (FOX.armc.ParamMappingABC method), 104
add_pes_evaluator() (FOX.armc.MonteCarloABC method), 108
ALIGNED (FOX.armc.MonteCarloABC attribute), 108
all() (FOX.MultiMolecule method), 26
allow_non_existent (param.validation attribute), 66
any() (FOX.MultiMolecule method), 26
apply_constraints() (FOX.armc.ParamMapping method), 106
apply_constraints() (FOX.armc.ParamMappingABC method), 105
apply_phi() (FOX.armc.ARMC property), 111
argmax() (FOX.MultiMolecule method), 26
argmin() (FOX.MultiMolecule method), 26
argpartition() (FOX.MultiMolecule method), 27
argsort() (FOX.MultiMolecule method), 27
ARMC (class in FOX.armc), 110
ARMCPT (class in FOX.armc), 112
as_mass_weighted() (FOX.MultiMolecule method), 56
as_mol() (FOX.MultiMolecule method), 55
as_mol2() (FOX.MultiMolecule method), 55
as_Molecule() (FOX.MultiMolecule method), 56
as_pdb() (FOX.MultiMolecule method), 55
as_xyz() (FOX.MultiMolecule method), 55
assemble_job() (FOX.armc.PackageManager static method), 107
assemble_job() (FOX.armc.PackageManagerABC static method), 107
astype() (FOX.MultiMolecule method), 27
atnum() (FOX.MultiMolecule property), 28
atom1() (FOX.MultiMolecule property), 28
atom12() (FOX.MultiMolecule property), 28
atom2() (FOX.MultiMolecule property), 28
atoms (FOX.MultiMolecule attribute), 16
atoms_view (FOX.MultiMolecule attribute), 39

B

base (FOX.MultiMolecule attribute), 28
BEHAVED (FOX.MultiMolecule attribute), 34
bonds (FOX.MultiMolecule attribute), 16
bonds (FOX.MultiMolecule attribute), 28
byteswap() (FOX.MultiMolecule method), 28

C

C_CONTIGUOUS (FOX.MultiMolecule attribute), 34
call_method() (in module FOX.properties), 90
CARRAY (FOX.MultiMolecule attribute), 34
CHARGE_LIKE (FOX.armc.ParamMapping attribute), 106
charge_tolerance (param.validation attribute), 67
choose() (FOX.MultiMolecule method), 29
clear_jobs() (FOX.armc.MonteCarloABC property), 109
clear_jobs() (FOX.armc.PackageManager static method), 107
clear_jobs() (FOX.armc.PackageManagerABC method), 107
clip() (FOX.MultiMolecule method), 29
clip_move() (FOX.armc.ParamMapping method), 106
clip_move() (FOX.armc.ParamMappingABC method), 105
colums (FOX.io.cp2k_to_prm.PRMMapping attribute), 103
compress() (FOX.MultiMolecule method), 30
conj() (FOX.MultiMolecule method), 30
get_free_energy() (in module FOX.ff.lj_param), 79
get_lig_center() (in module FOX.recipes), 100
get_multi_lig_center() (in module FOX.recipes), 100
get_pair_dict() (FOX.MultiMolecule method), 24
get_pes_descriptors()
  (FOX.armc.MonteCarloABC method), 110
get_pressure() (in module FOX.properties), 90
get_rmsd() (FOX.MultiMolecule method), 22
get_rmsf() (FOX.MultiMolecule method), 22
get_time_averaged_velocity()
  (FOX.MultiMolecule method), 21
get_vacf() (FOX.MultiMolecule method), 25
get_velocity() (FOX.MultiMolecule method), 25
guess (param.block attribute), 67
go\_field() (FOX.MultiMolecule method), 36
horse() (FOX.armc.ParamMapping method), 106
identify_move() (FOX.armc.ParamMapping method), 105
imag (FOX.MultiMolecule attribute), 37
init\_adf() (FOX.MultiMolecule method), 25
init\_average\_velocity() (FOX.MultiMolecule method), 19
init\_power\_spectrum() (FOX.MultiMolecule method), 24
init\_rdf() (FOX.MultiMolecule method), 23
init\_rmsd() (FOX.MultiMolecule method), 20
init\_rmsf() (FOX.MultiMolecule method), 20
init\_shell\_search() (FOX.MultiMolecule method), 22
init\_time\_averaged\_velocity()
  (FOX.MultiMolecule method), 20
item() (FOX.MultiMolecule method), 37
items() (FOX.armc.MonteCarloABC method), 108
items() (FOX.armc.PackageManagerABC method), 107
itemset() (FOX.MultiMolecule method), 38
itemsize (FOX.MultiMolecule attribute), 38
iter\_len (FOX.armc.ARMC attribute), 110
iter\_len (monte\_carlo attribute), 74

keep\_files () (monte\_carlo attribute), 75
key (FOX.io.cp2k\_to\_prm.PRMMapping attribute), 103

keys () (FOX.armc.MonteCarloABC method), 108
keys () (FOX.armc.PackageManagerABC method), 107
kwargs (param attribute), 66
kwargs (pes.block attribute), 70
kwargs (pes\_validation.block attribute), 71
kwargs (phi attribute), 77

ligand\_atoms (psf attribute), 69
logfile (monte\_carlo attribute), 74
logger () (FOX.armc.MonteCarloABC method), 108

mass () (FOX.MultiMolecule property), 39
max () (FOX.MultiMolecule method), 39
min () (FOX.MultiMolecule method), 39
module
  FOX.ff.lj_param, 78
  FOX.io.cp2k\_to\_prm, 103
  FOX.io.read\_prm, 85
  FOX.io.read\_psf, 80
  FOX.properties, 87
  FOX.armc.ligands, 98
  FOX.armc.param, 92
  FOX.armc.psf, 95
  FOX.armc.time\_resolution, 101
molecule (job attribute), 72
molecule () (FOX.armc.MonteCarloABC attribute), 108
MonteCarloABC (class in FOX.armc), 108
move () (FOX.armc.MonteCarloABC method), 109
move\_range () (FOX.armc.ParamMapping attribute), 106
move\_range () (FOX.armc.MonteCarloABC attribute), 104
move\_range (param attribute), 66
MultiMolecule (class in FOX), 16

name (FOX.io.cp2k\_to\_prm.PRMMapping attribute), 103
nbytes (FOX.MultiMolecule attribute), 40
ndim (FOX.MultiMolecule attribute), 40
newbyteorder () (FOX.MultiMolecule method), 40
nonzero () (FOX.MultiMolecule method), 41

order () (FOX.MultiMolecule property), 41
overlay_cp2k_settings() (FOX.PRMContainer method), 87
overlay_descriptor() (in module FOX.recipes), 94
overlay_mapping() (FOX.PRMContainer method), 86
OWNDATA (FOX.MultiMolecule attribute), 34

P
PackageManager (class in FOX.armc), 107
PackageManagerABC (class in FOX.armc), 107
param (param.block attribute), 67
ParamMapping (class in FOX.armc), 105
ParamMappingABC (class in FOX.armc), 104
partition() (FOX.MultiMolecule method), 41
path (monte_carlo attribute), 75
pes_post_process() (FOX.armc.MonteCarloABC property), 108
phi (FOX.armc.ARMC attribute), 110
phi (FOX.armc.PhiUpdater attribute), 114
phi (Phi attribute), 76
PhiUpdater (class in FOX.armc), 113
PhiUpdaterABC (class in FOX.armc), 113
plot_descriptor() (in module FOX.recipes), 95
post_process (FOX.io.cp2k_to_prm.PRMMapping attribute), 103
PRMMapping (class in FOX.io.cp2k_to_prm), 103
prod() (FOX.MultiMolecule method), 42
properties (FOX.MultiMolecule attribute), 16
psf_file (psf attribute), 68
ptp() (FOX.MultiMolecule method), 42
put() (FOX.MultiMolecule method), 42

R
radius() (FOX.MultiMolecule property), 42
random_slice() (FOX.MultiMolecule method), 18
ravel() (FOX.MultiMolecule method), 42
read() (FOX.PRMContainer class method), 86
read_multi_xyz() (in module FOX.io.read_xyz), 77
real (FOX.MultiMolecule attribute), 43
ref (pes.block attribute), 70
ref (pes_validation.block attribute), 71
repeat() (FOX.MultiMolecule method), 43
reset_origin() (FOX.MultiMolecule method), 18
reshape() (FOX.MultiMolecule method), 43
residue_argsort() (FOX.MultiMolecule method), 19
resize() (FOX.MultiMolecule method), 43
restart() (FOX.armc.ARMC method), 112
round() (FOX.MultiMolecule method), 17
rtf_file (psf attribute), 68
run_jobs() (FOX.armc.MonteCarloABC method), 109

S
searchsorted() (FOX.MultiMolecule method), 45
setfield() (FOX.MultiMolecule method), 45
setflags() (FOX.MultiMolecule method), 45
settings (job.block attribute), 73
shape (FOX.MultiMolecule attribute), 47
shape() (FOX.armc.PhiUpdaterABC property), 113
size (FOX.MultiMolecule attribute), 47
sort() (FOX.MultiMolecule method), 18
squeeze() (FOX.MultiMolecule method), 48
std() (FOX.MultiMolecule method), 48
str_file (psf attribute), 68
strides (FOX.MultiMolecule attribute), 48
sub_iter_len (FOX.armc.ARMC attribute), 110
sub_iter_len (monte_carlo attribute), 74
sum() (FOX.MultiMolecule method), 49
super_iter_len (FOX.armc.ARMC attribute), 110
swapaxes() (FOX.MultiMolecule method), 49
symbol() (FOX.MultiMolecule property), 49

T
T (FOX.MultiMolecule attribute), 26
take() (FOX.MultiMolecule method), 49
template (job.block attribute), 73
time_resolved_adf() (in module FOX.recipes), 102
time_resolved_rdf() (in module FOX.recipes), 101
to_hdf5() (FOX.armc.ARMC method), 111
to_struct_array()
(FOX.armc.ParamMappingABC method), 105
to_yaml_dict() (FOX.armc.ARMC method), 111
to_yaml_dict() (FOX.armc.ARMCPT method), 112
to_yaml_dict() (FOX.armc.PhiUpdaterABC method), 113
tobytes() (FOX.MultiMolecule method), 50
tofile() (FOX.MultiMolecule method), 50
tolist() (FOX.MultiMolecule method), 50
tostring() (FOX.MultiMolecule method), 51
trace() (FOX.MultiMolecule method), 52
transpose() (FOX.MultiMolecule method), 52
type (job attribute), 72
type (job.block attribute), 73
type (monte_carlo attribute), 74
type (param attribute), 65
type (phi attribute), 75

U
unit (FOX.io.cp2k_to_prm.PRMMapping attribute), 103
unit (param.block attribute), 67
update() (FOX.armc.PhiUpdater method), 114
update() (FOX.armc.PhiUpdaterABC method), 113
update_settings() (FOX.armc.PackageManager method), 107
update_settings()
  (FOX.armc.PackageManagerABC method), 107
UPDATEIFCOPY (FOX.MultiMolecule attribute), 34

V
values() (FOX.armc.MonteCarloABC method), 108
values() (FOX.armc.PackageManagerABC method), 107
var() (FOX.MultiMolecule method), 52
view() (FOX.MultiMolecule method), 53

W
write() (FOX.PRMContainer method), 86
WRITEABLE (FOX.MultiMolecule attribute), 34
WRITEBACKIFCOPY (FOX.MultiMolecule attribute), 34

X
x() (FOX.MultiMolecule property), 54

Y
y() (FOX.MultiMolecule property), 54

Z
z() (FOX.MultiMolecule property), 54