# Auto-FOX Documentation

***Release 0.10.2***

**B. F. van Beek**

**Aug 31, 2022**

# CONTENTS

Contents:

CHAPTER

ONE

# AUTOMATED FORCEFIELD OPTIMIZATION EXTENSION 0.10.2

**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:

```
>>> 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 stronly recomended 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 pip`
  - 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 auto-FOX --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`
- RDKit is required for a number of .psf-related recipes.
  - Anaconda: `conda install -name FOX -y -c conda-forge rdkit`
  - PyPi: `pip install rdkit`

CHAPTER

TWO

# AUTO-FOX DOCUMENTATION

## 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 particale 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.

$$g(r) = \frac{V}{N_a * N_b} \sum_{i=1}^{N_a} \sum_{j=1}^{N_b} \langle *placeholder* \rangle$$

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.

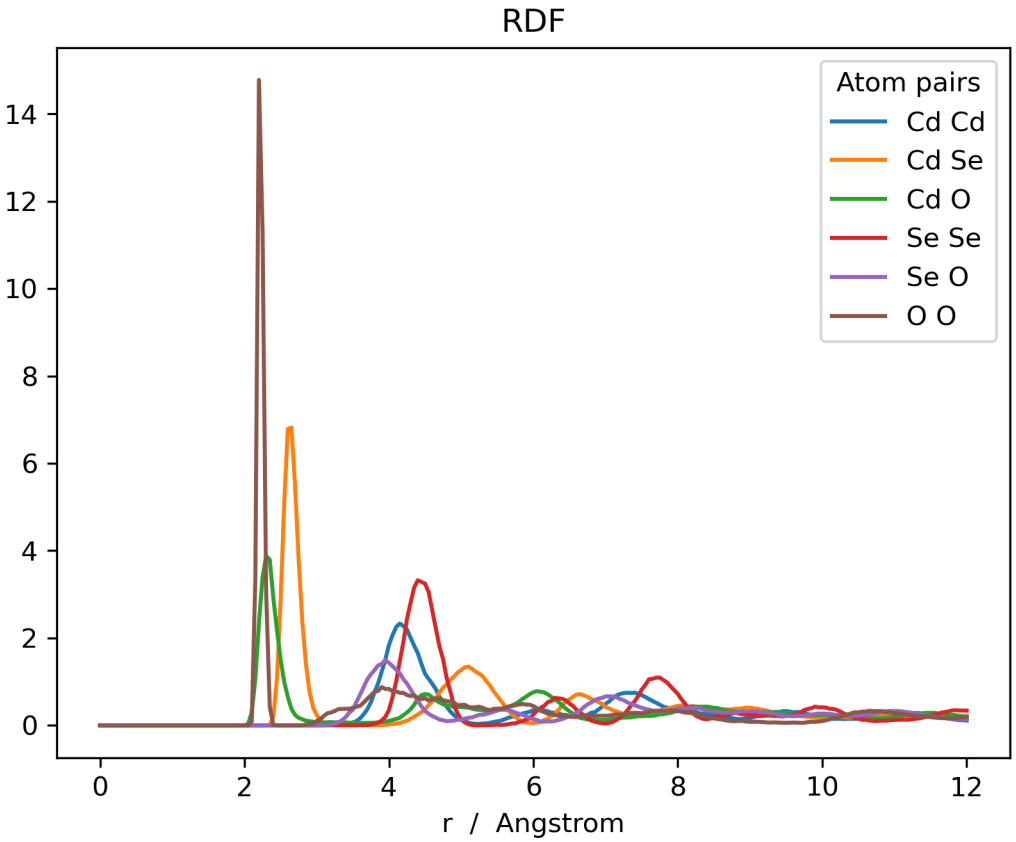
Below is an example RDF and ADF of a CdSe quantum dot pacified with formate ligands. The RDF is printed for all possible combinations of cadmium, selenium and oxygen (Cd_Cd, Cd_Se, Cd_O, Se_Se, Se_O and O_O).

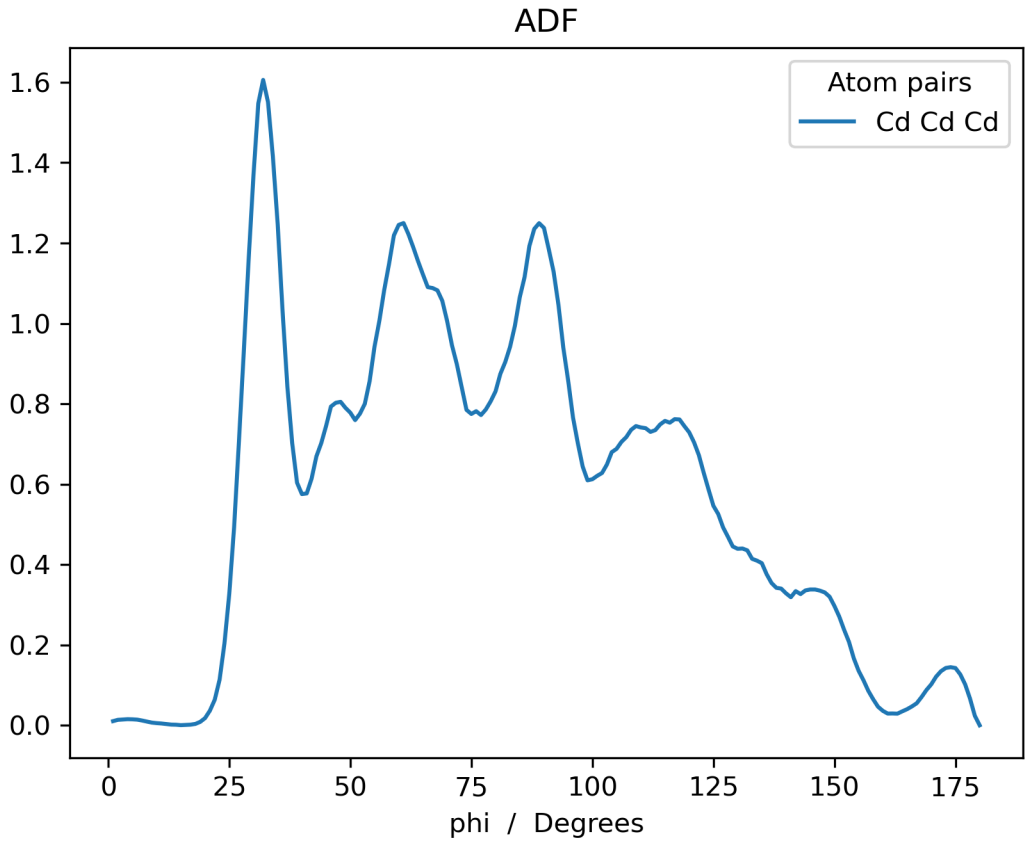
```
>>> from FOX import MultiMolecule, example_xyz

>>> mol = MultiMolecule.from_xyz(example_xyz)

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

>>> rdf.plot(title='RDF')
>>> adf.plot(title='ADF')
>>> adf_weighted.plot(title='Distance-weighted ADF')
```

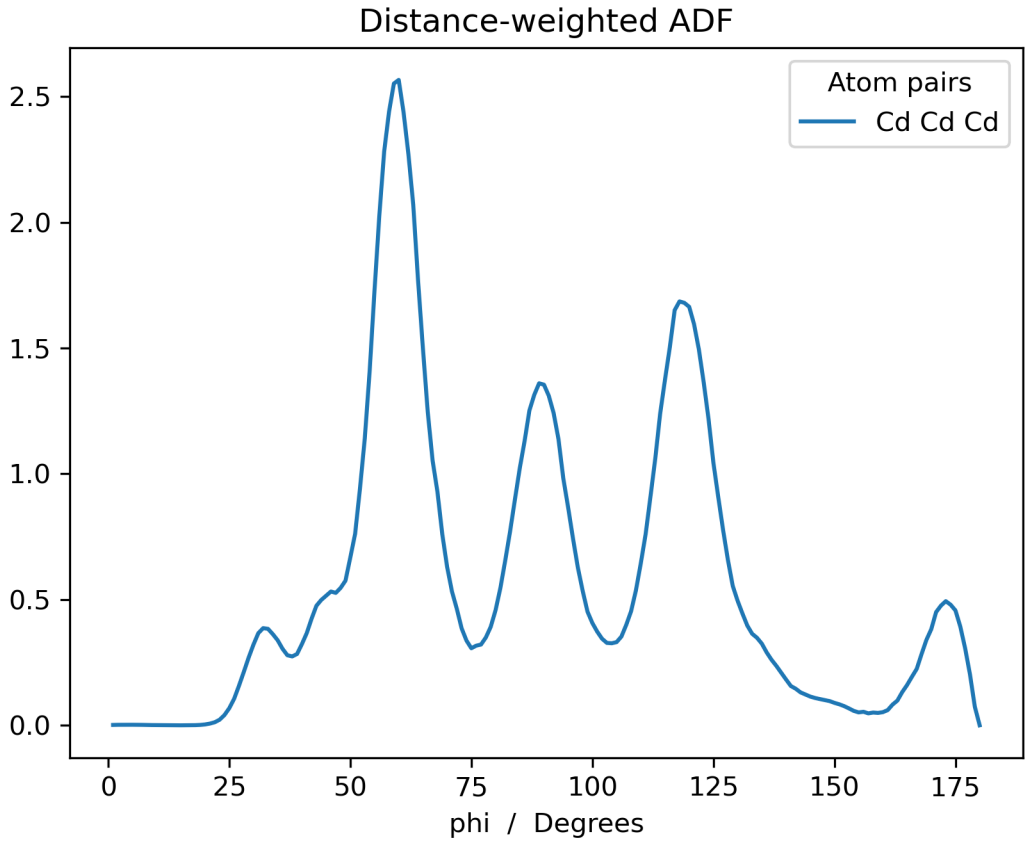
One can take into account a systems periodicity by settings the molecules' `lattice` vectors and specifying the axes along which the system is periodic.

The lattice vectors can be provided in one of two formats:

- A $(3, 3)$ matrix.


RDF
Atom pairs
Cd Cd
Cd Se
Cd O
Se Se
Se O
O O
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
r / Angstrom


ADF
Atom pairs
Cd Cd Cd
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
25
50
75
100
125
150
175
phi / Degrees


Distance-weighted ADF
Atom pairs
Cd Cd Cd
2.5
2.0
1.5
1.0
0.5
0.0
0
25
50
75
100
125
150
175
phi / Degrees

- A $(N_{mol}, 3, 3)$-shaped tensor if they vary across the trajectory.

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

>>> lattice = np.array(...)
>>> mol = MultiMolecule.from_xyz(...)
>>> mol.lattice = lattice

# Periodic along the x, y and/or z axes
>>> rdf = mol.init_rdf(atom_subset=('Cd', 'Se', 'O'), periodic="xy")
>>> adf = mol.init_adf(r_max=8, atom_subset=('Cd',), periodic="xyz")
```

### 2.1.1 API

## 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, moleculair indices in a MD trajectory.

$$\rho^{\mathrm{RMSD}}(t) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\mathbf{r}_i(t) - \mathbf{r}_i^{\mathrm{ref}}\right)^2}$$

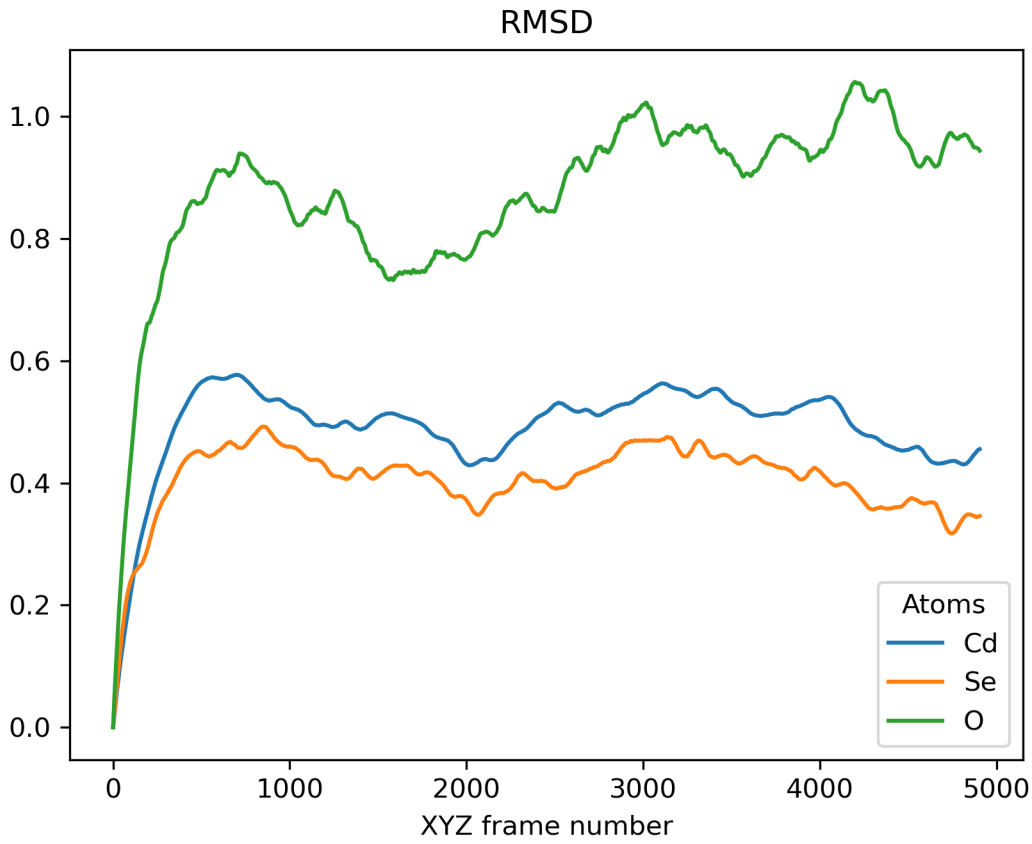
Given a trajectory, `mol`, stored as a `FOX.MultiMolecule` instance, the RMSD can be calculated with the `FOX.MultiMolecule.init_rmsd()` method using the following command:

```
>>> 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.

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

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


RMSD
1.0
0.8
0.6
0.4
0.2
0.0
0
1000
2000
3000
4000
5000
XYZ frame number
Atoms
Cd
Se
O

### 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.

$$\rho_i^{\mathrm{RMSF}} = \sqrt{\left\langle \left( \mathbf{r}_i - \langle \mathbf{r}_i \rangle \right)^2 \right\rangle}$$

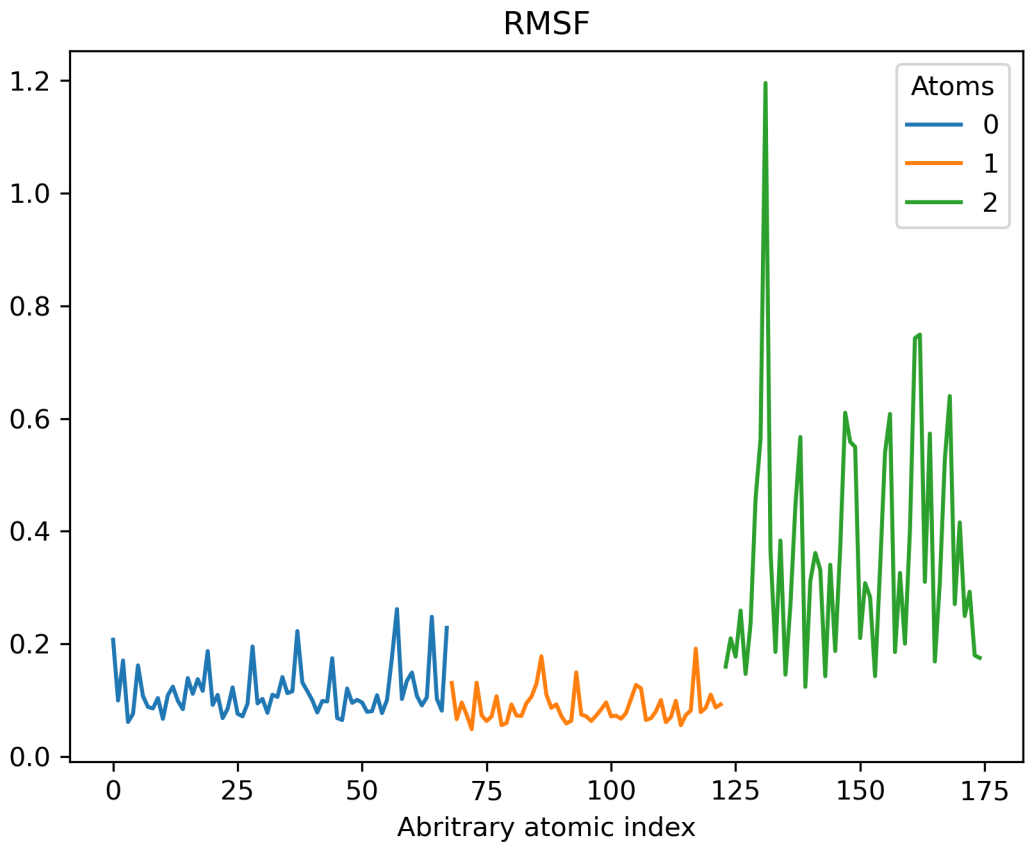
Given a trajectory, `mol`, stored as a `FOX.MultiMolecule` instance, the RMSF can be calculated with the `FOX.MultiMolecule.init_rmsf()` method using the following command:

```
>>> 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.

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

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



### 2.2.3 The atom_subset argument

In the above two examples `atom_subset=None` was used 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

## 2.3 The MultiMolecule Class

The API of the `FOX.MultiMolecule` class.

### 2.3.1 API FOX.MultiMolecule

## 2.4 Addaptive Rate Monte Carlo

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

The implemented algorithm can be summarized as following:

### 2.4.1 The algorithm

1. A trial state, $S_l$, is generated by moving a random parameter retrieved from a user-specified parameter set (*e.g.* atomic charge).
2. It is checked whether or not the trial state has been previously visited.
   - If `True`, retrieve the previously calculated PES.
   - If `False`, calculate a new PES with the generated parameters $S_l$.

$$p(k \leftarrow l) = \begin{cases} 1, & \Delta\varepsilon_{QM-MM}(S_k) \;\; \Delta\varepsilon_{QM-MM}(S_l) \\ 0, & \Delta\varepsilon_{QM-MM}(S_k) \;\; \Delta\varepsilon_{QM-MM}(S_l) \end{cases} \tag{2.1}$$

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).

$$\Delta\varepsilon_{QM-MM} = \frac{\sum_i^N \left| r_i^{QM} - r_i^{MM} \right|^2}{\sum_i^N r_i^{QM}} \tag{2.2}$$

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.

$$\begin{aligned} \Delta\varepsilon_{QM-MM}(S_k) + \phi \quad &\text{if} \quad \Delta\varepsilon_{QM-MM}(S_k) \;\; \Delta\varepsilon_{QM-MM}(S_l) \\ \Delta\varepsilon_{QM-MM}(S_l) + \phi \quad &\text{if} \quad \Delta\varepsilon_{QM-MM}(S_k) \;\; \Delta\varepsilon_{QM-MM}(S_l) \end{aligned} \tag{2.3}$$

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 begining of every super-iteration $\kappa$. In this example the total number of iterations, $\kappa\omega$, is divided into $\kappa$ super- and $\omega$ sub-iterations.

$$\phi_{\kappa\omega} = \phi_{(\kappa-1)\omega} * \gamma^{\text{sgn}(\alpha_t - \overline{\alpha}_{(\kappa-1)})} \quad \kappa = 1, 2, 3, ..., N \tag{2.4}$$

### 2.4.2 Parameters

```yaml
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
        frozen:
            C_1: 0.4524
    lennard_jones:
        -   unit: kjmol
            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
```

(continues on next page)

(continued from previous page)

```
        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 descriped in the `pes` block. Provided below is an example where the radial dsitribution function (RDF) is used as PES descriptor, more specifically the RDF constructed from all possible combinations of cadmium, selenium and oxygen atoms.

```
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 are used as PES descriptors. In this example the RDF is construced for all combinations of cadmium, selenium and oxygen atoms (Cd, Se & O), whereas the ADF is construced for all combinations of cadmium and selenium atoms (Cd & Se).

```
pes:
    rdf:
        func: FOX.MultiMolecule.init_rdf
        kwargs:
```

(continues on next page)

(continued from previous page)

```
            atom_subset: [Cd, Se, O]

    adf:
        func: FOX.MultiMolecule.init_adf
        kwargs:
            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 thee 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 *kwargs* key. The *kwargs* key is entirely optional and can be skipped if desired.

An example of a custom, albit rather nonsensical, PES descriptor involving the `numpy.sum()` function is provided below:

```
pes:
  numpy_sum:
      func: numpy.sum
      kwargs:
          axis: 0
```

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

```
>>> import numpy
>>> from FOX import MultiMolecule

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

>>> mol = MultiMolecule(...)
>>> func(mol, **kwargs)
```

### 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
```

(continues on next page)

(continued from previous page)

```
      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 appropiate 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.

```
param:
    charge:
        constraints:
            - Cd == -2 * $LIGAND
            - 0 < Cd < 2
            - -2 < Se < 0
```

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 responsibiliy to ensure that the initial charge is actually integer.

---

### 2.4.5 Parameter Guessing

```
param:
    lennard_jones:
        - unit: kjmol
          param: epsilon
          Cs Cs: 0.1882
          Cs Pb: 0.7227
          Pb Pb: 2.7740
          guess: rdf
        - unit: nm
          param: sigma
          frozen:
              guess: uff
```

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

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 construcetd 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]
```

(continues on next page)

(continued from previous page)

```
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

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

| *psf* | Description |
|---|---|
| *psf.str_file* | The path+filename to one or more stream file. |
| *psf.rtf_file* | The path+filename to one or more MATCH-produced rtf file. |
| *psf.psf_file* | The path+filename to one or more psf files. |
| *psf.ligand_atoms* | All atoms within a ligand. |

| *pes* | Description |
|---|---|
| *pes.block.func* | The callable for performing the Monte Carlo moves. |
| *pes.block.ref* | A list of reference values for when *func* operates on `qmflows.Result` objects. |
| *pes.block.kwargs* | A dictionary with keyword arguments for *pes.block.func*. |
| *pes.block.err_func* | A function for computing the auxilary error of the specified PES descriptor. |

| *pes_validation* | Description |
|---|---|
| *pes_validation.block.func* | The callable for performing the Monte Carlo validation. |
| *pes_validation.block.ref* | A list of reference values for when *func* operates on `qmflows.Result` objects. |
| *pes_validation.block.kwargs* | A dictionary with keyword arguments for *pes_validation.block.func*. |

| *job* | Description |
|---|---|
| *job.type* | The type of package manager. |
| *job.molecule* | One or more .xyz files with reference (QM) potential energy surfaces. |
| *job.lattice* | One or more CP2K .cell files with the lattice vectors of each mol in *job.molecule*. |
| *job.block.type* | An instance of a QMFlows `Package`. |
| *job.block.settings* | The job settings as used by *job.block.type*. |
| *job.block.template* | A settings template for updating *job.block.settings*. |

| *monte_carlo* | Description |
|---|---|
| *monte_carlo.type* | The type of Monte Carlo procedure. |
| *monte_carlo.iter_len* | The total number of ARMC iterations $\kappa\omega$. |
| *monte_carlo.sub_iter_len* | The length of each ARMC subiteration $\omega$. |
| *monte_carlo.logfile* | The name of the ARMC logfile. |
| *monte_carlo.hdf5_file* | The name of the ARMC .hdf5 file. |
| *monte_carlo.path* | The path to the ARMC working directory. |
| *monte_carlo.folder* | The name of the ARMC working directory. |
| *monte_carlo.keep_files* | Whether to keep *all* raw output files or not. |

| *phi* | Description |
|---|---|
| *phi.type* | The type of phi updater. |
| *phi.gamma* | The constant $\gamma$. |
| *phi.a_target* | The target acceptance rate $\alpha_t$. |
| *phi.phi* | The initial value of the variable $\phi$. |
| *phi.func* | The callable for updating *phi*. |
| *phi.kwargs* | A dictionary with keyword arguments for *phi.func*. |

### 2.5.2 param

All forcefield-parameter related options.

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

**Examples**

```
param:
    type: FOX.armc.ParamMapping
    move_range:
        start: 0.005
        stop: 0.1
        step: 0.005
```

(continues on next page)

(continued from previous page)

```
        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.**type**

**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**

*FOX.armc.ParamMapping*
A **ParamMappingABC** subclass.

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 a arguments and return a new one.

**See Also**

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

param.**kwargs**

**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 completelly optional.

**Examples**

```
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.**ligand_atoms**

**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 containg the *func* and, optionally, *kwargs* keys.

**Examples**

```
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.**kwargs**

**Parameter**

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

A dictionary with keyword arguments for *func*.

pes.block.**err_func**

**Parameter**

- **Type** - str or Callable[[ArrayLike, ArrayLike], float]
- **Default Value** - *FOX.armc.default_error_func*

A function for computing the auxilary error of the specified PES descriptor. The callable should be able to take two array-like objects as arguments and return a scalar.

---

**See Also**

***FOX.armc.mse_normalized()* & *FOX.armc.mse_normalized_v2()***
Return a normalized mean square error (MSE) over the flattened input.

*FOX.armc.mse_normalized_weighted()* & *FOX.armc.mse_normalized_weighted_v2()*
Return a normalized mean square error (MSE) over the flattened subarrays of the input.

*FOX.armc.mse_normalized_max()*
Return a maximum normalized mean square error (MSE) over the flattened subarrays of the input.

### 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 containg the *func* and, optionally, *kwargs* keys.

**Examples**

```
pes_validation:
    adf:
        func: FOX.MultiMolecule.init_adf
        kwargs:
            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 identintical 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.**kwargs**

**Parameter**

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

A dictionary with keyword arguments for *func*.

The structure of this block is identintical to its counterpart in *pes.block.kwargs*.

Passing a list of dictionaries allows one the use different kwargs for different jobs in PES-averaged ARMC or ARMCPT:

```yaml
job:
    molecule:
        - mol_CdSeO.xyz
        - mol_CdSeN.xyz

pes_validation:
    rdf:
        func: FOX.MultiMolecule.init_rdf
        kwargs:
            - atom_subset: [Cd, Se, O]
            - atom_subset: [Cd, Se, N]
```

### 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**

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

(continues on next page)

(continued from previous page)

```
    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**

*FOX.armc.PackageManager*
A **PackageManagerABC** subclass.

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.**lattice**

**Parameter**

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

One or more CP2K .cell files with the lattice vectors of each mol in *job.molecule*.

This option should be specified is one is performing calculations on periodic systems.

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, alternativelly, 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**

```
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**

```
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.ARMCPT*)

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.ARMCPT*)

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.ARMCPT*)

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.**kwargs**

**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 fileformat 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-effiecient, 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.

```
>>> 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.

## 2.7 FOX.ff.lj_param

A module for estimating Lennard-Jones parameters.

**Examples**

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

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

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

>>> print(param)
            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

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

### 2.7.2 API

FOX.ff.lj_param.**estimate_lj**(*rdf*, *temperature=298.15*, *sigma_estimate='base'*)

Estimate the Lennard-Jones $\sigma$ and $\varepsilon$ parameters using an RDF.

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

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

The $\sigma$ and $\varepsilon$ parameters are estimated as following:

- $\sigma$: The radii at which the first inflection point or peak base occurs in **rdf**.
- $\varepsilon$: The minimum value in of the **rdf** ree energy multiplied by $-1$.
- All values are calculated per atom pair specified in **rdf**.

**Parameters**

- **rdf** (pandas.DataFrame) – A radial distribution function. The columns should consist of atom-pairs.
- **temperature** (float) – The temperature in Kelvin.
- **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 **rdf** are used as index.

**Return type**

pandas.DataFrame

**See also:**

**MultiMolecule.init_rdf()**

Initialize the calculation of radial distribution functions (RDFs).

*get_free_energy()*

Convert a distribution function into a free energy function.

FOX.ff.lj_param.**get_free_energy**(*distribution*, *temperature=298.15*, *unit='kcal/mol'*, *inf_replace=nan*)

Convert a distribution function into a free energy function.

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

$$F(g(r)) = -RT * \ln(g(r))$$

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

**Parameters**

- **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

### 2.8.2 API

## 2.9 PRMContainer

A class for reading and generating .prm parameter files.

### 2.9.1 Index

### 2.9.2 API

## 2.10 Properties

## 2.11 Recipes

Various recipes implemented in Auto-FOX.

| | |
|---|---|
| *FOX.recipes.param* | A set of functions for analyzing and plotting ARMC results. |

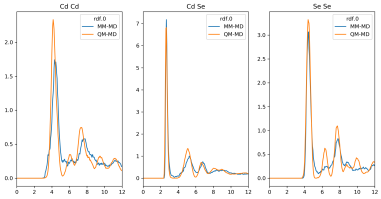
### 2.11.1 FOX.recipes.param

A set of functions for analyzing and plotting ARMC results.

**Examples**

A general overview of the functions within this module.

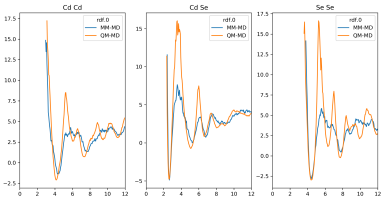
```
>>> import pandas as pd
>>> from FOX.recipes import get_best, overlay_descriptor, plot_descriptor

>>> hdf5_file: str = ...

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

# Compare the RDF to its reference RDF and plot
>>> rdf_dict = overlay_descriptor(hdf5_file, name='rdf')
>>> plot_descriptor(rdf_dict)
```




**Examples**

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

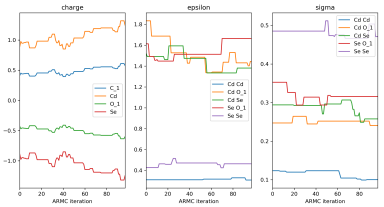
```
>>> import pandas as pd
>>> from FOX import get_free_energy
>>> from FOX.recipes import get_best, overlay_descriptor, plot_descriptor

>>> 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

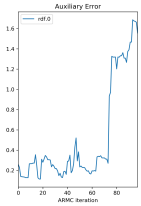
A workflow for plotting parameters as a function of ARMC iterations.

```
>>> 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.

```
>>> ...

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

>>> plot_descriptor(err_dict)
```




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

```
>>> ...

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

>>> plot_descriptor(err_slice_dict)
```

#### Index

#### API

### 2.11.2 FOX.recipes.psf

### 2.11.3 FOX.recipes.ligands

### 2.11.4 FOX.recipes.time_resolution

### 2.11.5 FOX.recipes.similarity

## 2.12 cp2k_to_prm

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

### 2.12.1 Index

| | |
|---|---|
| *PRMMapping* | A TypedMapping providing tools for converting CP2K settings to .prm-compatible values. |
| *CP2K_TO_PRM* | |

### 2.12.2 API

**class** FOX.io.cp2k_to_prm.**PRMMapping**

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* containg 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

| | |
|---|---|
| *ParamMappingABC*(data, move_range, func[, ...]) | A Mapping for storing and updating forcefield parameters. |
| *ParamMapping*(data[, move_range, func]) | A Mapping for storing and updating forcefield parameters. |

## 2.14 API

**class** FOX.armc.**ParamMappingABC**(*data*, *move_range*, *func*, *constraints=None*, *is_independent=False*, ***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**

```
>>> 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 = mappingproxy({'min': -inf, 'max': inf, 'count': -1, 'frozen': False, 'guess': False, 'unit': ''})**

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_idx*)

Identify the to-be moved parameter and the size of the move.

**Parameters**
**param_idx** (str) – The name of the parameter-containg 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*, *param_idx*)

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.
- **param_idx** (str) – The name of the parameter-containg column.

**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-containg 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.

**constraints_to_str**()

Convert the constraints into a human-readably pandas.Series.

**get_cp2k_dicts**()

Get dictionaries with CP2K parameters that are parsable by QMFlows.

**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 ]])*, *func=<ufunc 'multiply'>*, ***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**

```
>>> 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

**CHARGE_LIKE = frozenset({'charge'})**

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

**identify_move**(*param_idx*)

Identify and return a random parameter and move size.

**Parameters**
**param_idx** (int) – The name of the parameter-containg 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*, *param_idx*)

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

**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**.

Performs 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_idx** (int) – The name of the parameter-containg column.

## 2.15 Index

| | |
|---|---|
| *PackageManagerABC*(data[, hook]) | A class for managing qmflows-style jobs. |
| *PackageManager*(data[, hook]) | A class for managing qmflows-style jobs. |

## 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_seq*)

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_seq*)

Update all forcefield parameter blocks in this instance's CP2K settings.

## 2.17 Index

| | |
|---|---|
| *MonteCarloABC*(molecule, package_manager, param) | The base *MonteCarloABC* class. |
| *ARMC*(phi[, iter_len, sub_iter_len]) | The Addaptive Rate Monte Carlo class (*ARMC*). |
| *ARMCPT*([swapper]) | An *ARMC* subclass implementing a parallel tempering procedure. |

## 2.18 API

**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 **value** 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*, *err_func*, *args=()*, *kwargs=mappingproxy({})*, *validation=False*, *ref=None*)

Add a callable to this instance for constructing PES-descriptors.

**Examples**

```
>>> 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.
- **err_func** (Callable) – The function for computing the auxilary error.
- **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 inplace update of the `'param'` column in **self.param**. By default the move is applied in a multiplicative manner. **self.job.md_settings** and **self.job.preopt_settings** are updated to reflect the change in parameters.

**Examples**

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

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

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

**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. Usefull for manual inspection in case cp2k hard-crashes at this point.

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

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

**class** FOX.armc.**ARMC**(*phi*, *iter_len=50000*, *sub_iter_len=100*, ***kwargs*)

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:

$$\Delta\varepsilon_{QM-MM} = \frac{\sum_i^N |r_i^{QM} - r_i^{MM}|^2}{r_i^{QM}}$$

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

**Returns**
An array with $m * n$ auxilary errors

**Return type**
np.ndarray[np.float64], shape $(m, n)$

**restart**()

Restart a previously started Addaptive Rate Monte Carlo procedure.

**class** FOX.armc.**ARMCPT**(*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**

```
>>> 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, ndarray], ndarray]

**property shape**

Return the shape of *phi*.

Serves as a wrapper around the shape attribute of *phi*. Note that *phi*, *gamma* and *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**

```
>>> 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, ndarray], ndarray]

**update**(*acceptance*, *, *logger=None*)

Update the variable $\phi$.

$\phi$ is updated based on the target accepatance rate, $\alpha_t$, and the acceptance rate, **acceptance**, of the current super-iteration:

$$\phi_{\kappa\omega} = \phi_{(\kappa-1)\omega} * \gamma^{\text{sgn}(\alpha_t - \overline{\alpha}_{(\kappa-1)})}$$

**Parameters**

- **acceptance** (ArrayLike[np.bool_]) – An array-like object consisting of booleans.
- **logger** (logging.Logger, optional) – A logger for reporting the updated value.

## 2.21 err_funcs

A module with ARMC error functions.

### 2.21.1 Index

| | |
|---|---|
| *mse_normalized*(qm, mm) | Return a normalized mean square error (MSE) over the flattened input. |
| *mse_normalized_weighted*(qm, mm) | Return a normalized mean square error (MSE) over the flattened subarrays of the input. |
| *mse_normalized_max*(qm, mm) | Return the maximum normalized mean square error (MSE) over the flattened subarrays of the input. |
| *mse_normalized_v2*(qm, mm) | Return a normalized mean square error (MSE) over the flattened input. |
| *mse_normalized_weighted_v2*(qm, mm) | Return a normalized mean square error (MSE) over the flattened subarrays of the input. |
| *default_error_func*(qm, mm) | Return a normalized mean square error (MSE) over the flattened input. |

### 2.21.2 API

FOX.armc.**mse_normalized**(*qm*, *mm*)

Return a normalized mean square error (MSE) over the flattened input.

FOX.armc.**mse_normalized_weighted**(*qm*, *mm*)

Return a normalized mean square error (MSE) over the flattened subarrays of the input.

>1D array-likes are herein treated as stacks of flattened arrays.

FOX.armc.**mse_normalized_max**(*qm*, *mm*)

Return the maximum normalized mean square error (MSE) over the flattened subarrays of the input.

>1D array-likes are herein treated as stacks of flattened arrays.

FOX.armc.**mse_normalized_v2**(*qm*, *mm*)

Return a normalized mean square error (MSE) over the flattened input.

Normalize before squaring the error.

FOX.armc.**mse_normalized_weighted_v2**(*qm*, *mm*)

Return a normalized mean square error (MSE) over the flattened subarrays of the input.

>1D array-likes are herein treated as stacks of flattened arrays.

Normalize before squaring the error.

FOX.armc.**err_normalized**(*qm*, *mm*)

Return a normalized wrror over the flattened input.

Normalize before taking the exponent - 1 of the error.

FOX.armc.**err_normalized_weighted**(*qm*, *mm*)

Return a normalized error over the flattened subarrays of the input.

>1D array-likes are herein treated as stacks of flattened arrays.

FOX.armc.**default_error_func = FOX.armc.mse_normalized**

An alias for FOX.arc.mse_normalized().

# PYTHON MODULE INDEX

## f

# INDEX

## Symbols



## A



## C



## D



## E



## F

## G



## H



## I



## K



## L



## M



## N



## P



## R

## S



## T



## U



## V