openforcefield Documentation

Release 0.7.0

Open Force Field Consortium

Jul 03, 2020

CONTENTS

A modern, extensible library for molecular mechanics force field science from the Open Force Field Initiative

CHAPTER

ONE

USER GUIDE

1.1 Installation

1.1.1 Installing via *conda*

The simplest way to install the Open Forcefield Toolkit is via the conda package manager. Packages are provided on the omnia Anaconda Cloud channel for Linux, OS X, and Win platforms. The openforcefield Anaconda Cloud page has useful instructions and download statistics.

If you are using the anaconda scientific Python distribution, you already have the `conda` package manager installed. If not, the quickest way to get started is to install the miniconda distribution, a lightweight minimal installation of Anaconda Python.

On `linux`, you can install the Python 3 version into `$HOME/miniconda3` with (on `bash` systems):

```
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
$ bash ./Miniconda3-latest-Linux-x86_64.sh -b -p $HOME/miniconda3
$ source ~/miniconda3/etc/profile.d/conda.sh
$ conda activate base
```

On `osx`, you want to use the `osx` binary

```
$ curl https://repo.anaconda.com/miniconda/Miniconda3-latest-MacOSX-x86_64.sh -O
$ bash ./Miniconda3-latest-MacOSX-x86_64.sh -b -p $HOME/miniconda3
$ source ~/miniconda3/etc/profile.d/conda.sh
$ conda activate base
```

You may want to add the new `source ~/miniconda3/etc/profile.d/conda.sh` line to your `~/.bashrc` file to ensure Anaconda Python can enabled in subsequent terminal sessions. `conda activate base` will need to be run in each subsequent terminal session to return to the environment where the toolkit will be installed.

Note that `openforcefield` will be installed into this local Python installation, so that you will not need to worry about disrupting existing Python installations.

Note: Installation via the conda package manager is the preferred method since all dependencies are automatically fetched and installed for you.

1.1.2 Required dependencies

The `openforcefield` toolkit makes use of the Omnia and Conda Forge free and open source community package repositories:

```
$ conda config --add channels omnia --add channels conda-forge
$ conda update --all
```

This only needs to be done once.

Note: If automation is required, provide the `--yes` argument to `conda update` and `conda install` comamnds. More information on the `conda` command-line API can be found in the conda online documentation.

Release build

You can install the latest stable release build of `openforcefield` via the `conda` package with

```
$ conda config --add channels omnia --add channels conda-forge
$ conda install openforcefield
```

This version is recommended for all users not actively developing new forcefield parameterization algorithms.

Note: The conda package manager will install dependencies from binary packages automatically, including difficult-to-install packages such as OpenMM, numpy, and scipy. This is really the easiest way to get started.

Upgrading your installation

To update an earlier `conda` installation of `openforcefield` to the latest release version, you can use `conda update`:

```
$ conda update openforcefield
```

Optional dependencies

This toolkit can optionally make use of the OpenEye toolkit if the user has a license key installed. Academic laboratories intending to release results into the public domain can obtain a free license key, while other users (including academics intending to use the software for purposes of generating protected intellectual property) must pay to obtain a license.

To install the OpenEye toolkits (provided you have a valid license file):

```
$ conda install --yes -c openeye openeye-toolkits
```

No essential `openforcefield` release capabilities *require* the OpenEye toolkit, but the Open Force Field developers make use of it in parameterizing new open source force fields. It is known that there are certain differences in toolkit behavior between RDKit and OpenEye when reading a small fraction of molecules, and we encourage you to report any unexpected behavior that may be caused by toolkit differences to our issue tracker.

1.1.3 Alternative method: Single-file installer

As of release 0.4.1, single-file installers are available for each Open Force Field Toolkit release. These are provided primarily for users who do not have access to the Anaconda cloud for installing packages. These installers have few requirements beyond a Linux or OSX operating system and will, in one command, produce a functional Python executable containing the Open Force Field Toolkit, as well as all required dependencies. The installers are very similar to the widely-used Miniconda `*.sh` files. Accordingly, installation using the "single-file installer" does not require root access.

The installers are between 200 and 300 MB each, and can be downloaded from the "Assets" section of the Toolkit's GitHub Releases page. They are generated using a workflow leveraging the "conda constructor" utility.

Please report any installer difficulties to the OFF Toolkit Issue tracker, as we hope to make this a major distribution channel for the toolkit moving forward.

Installation

Download the appropriate installer (`openforcefield-<X.Y.Z>-py37-<your platform>-x86_64.sh`) from the "Assets" section at the bottom of the desired release. Then, install the toolkit with the following command:

```
$ bash openforcefield-<X.Y.Z>-py37-<your platform>-x86_64.sh
```

and follow the prompts.

Note: You must have write access to the installation directory. This is generally somewhere in the user's home directory. When prompted, we recommend NOT making modifications to your `bash_profile`.

Warning: We recommend that you do not install this package as root. Conda is intended to support on-the-fly creation of several independent environments, and managing a multi-user conda installation is somewhat involved.

Usage

Any time you want to use this conda environment in a terminal, run

```
$ source <install_directory>/etc/profile.d/conda.sh
$ conda activate base
```

Once the base environment is activated, your system will default to use python (and other executables) from the newly installed conda environment.

Installing optional OpenEye toolkits

We're waiting on permission to redistribute the OpenEye toolkits inside the single-file installer, so for now the installers only ship with the open-source backend (RDKit+AmberTools). With this in mind, the conda environment created by the installer *contains the conda package manager itself*, which can be used to install the OpenEye toolkits if you have access to the Anaconda cloud.

```
$ conda install -c openeye openeye-toolkits
```

Note: The OpenEye Toolkits conda package still requires a valid OpenEye license file to run.

1.2 Release History

Releases follow the `major.minor.micro` scheme recommended by PEP440, where

- `major` increments denote a change that may break API compatibility with previous `major` releases
- `minor` increments add features but do not break API compatibility
- `micro` increments represent bugfix releases or improvements in documentation

1.2.1 0.7.0 - Charge Increment Model, Proper Torsion interpolation, and new Molecule methods

This is a relatively large release, motivated by the idea that changing existing functionality is bad so we shouldn't do it too often, but when we do change things we should do it all at once.

Here's a brief rundown of what changed, migration tips, and how to find more details in the full release notes below:

- To provide more consistent partial charges for a given molecule, existing conformers are now disregarded by default by `Molecule.assign_partial_charges`. Instead, new conformers are generated for use in semiempirical calculations. Search for `use_conformers`.
- Formal charges are now always returned as `simtk.unit.Quantity` objects, with units of elementary charge. To convert them to integers, use `from simtk import unit` and `atom.formal_charge.value_in_unit(unit.elementary_charge)` or `mol.total_charge.value_in_unit(unit.elementary_charge)`. Search `atom.formal_charge`.
- The OpenFF Toolkit now automatically reads and writes partial charges in SDF files. Search for `atom.dprop.PartialCharges`.
- The OpenFF Toolkit now has different behavior for handling multi-molecule and multi-conformer SDF files. Search `multi-conformer`.
- The OpenFF Toolkit now distinguishes between partial charges that are all-zero and partial charges that are unknown. Search `partial_charges = None`.
- `Topology.to_openmm` now assigns unique atoms names by default. Search `ensure_unique_atom_names`.
- Molecule equality checks are now done by graph comparison instead of SMILES comparison. Search `Molecule.are_isomorphic`.
- The `ChemicalEnvironment` module was almost entirely removed, as it is an outdated duplicate of some Chemper functionality. Search `ChemicalEnvironment`.
- `TopologyMolecule.topology_particle_start_index` has been removed from the `TopologyMolecule` API, since atoms and virtualsites are no longer contiguous in the `Topology` particle indexing system. Search `topology_particle_start_index`.
- `compute_wiberg_bond_orders` has been renamed to `assign_fractional_bond_orders`.

There are also a number of new features, such as:

- Support for `ChargeIncrementModel` sections in force fields.
- Support for `ProperTorsion` `k` interpolation in force fields using fractional bond orders.
- Support for AM1-Mulliken, Gasteiger, and other charge methods using the new `assign_partial_charges` methods.
- Support for AM1-Wiberg bond order calculation using either the OpenEye or RDKit/AmberTools backends and the `assign_fractional_bond_orders` methods.
- Initial (limited) interoperability with QCArchive, via `Molecule.to_qcschema` and `from_qcschema`.
- A `Molecule.visualize` method.
- Several additional `Molecule` methods, including state enumeration and mapped SMILES creation.

Major Feature: Support for the SMIRNOFF ChargeIncrementModel tag

The ChargeIncrementModel tag in the SMIRNOFF specification provides analagous functionality to AM1-BCC, except that instead of AM1-Mulliken charges, a number of different charge methods can be called, and

instead of a fixed library of two-atom charge corrections, an arbitrary number of SMIRKS-based, N-atom charge corrections can be defined in the SMIRNOFF format.

The initial implementation of the SMIRNOFF `ChargeIncrementModel` tag accepts keywords for `version`, `partial_charge_method`, and `number_of_conformers`. `partial_charge_method` can be any string, and it is up to the `ToolkitWrapper`'s `compute_partial_charges` methods to understand what they mean. For geometry-independent `partial_charge_method` choices, `number_of_conformers` should be set to zero.

SMIRKS-based parameter application for `ChargeIncrement` parameters is different than other SMIRNOFF sections. The initial implementation of `ChargeIncrementModelHandler` follows these rules:

- an atom can be subject to many `ChargeIncrement` parameters, which combine additively.
- a `ChargeIncrement` that matches a set of atoms is overwritten only if another `ChargeIncrement` matches the same group of atoms, regardless of order. This overriding follows the normal SMIRNOFF hierarchy.

To give a concise example, what if a molecule `A-B(-C)-D` were being parametrized, and the force field defined `ChargeIncrement` SMIRKS in the following order?

1) `[A:1]-[B:2]`
2) `[B:1]-[A:2]`
3) `[A:1]-[B:2]-[C:3]`
4) `[*:1]-[B:2](-[*:3])-[*:4]`
5) `[D:1]-[B:2](-[*:3])-[*:4]`

In the case above, the ChargeIncrement from parameters 1 and 4 would NOT be applied to the molecule, since another parameter matching the same set of atoms is specified further down in the parameter hierarchy (despite those subsequent matches being in a different order).

Ultimately, the ChargeIncrement contributions from parameters 2, 3, and 5 would be summed and applied.

It's also important to identify a behavior that these rules were written to *avoid*: if not for the "regardless of order" clause in the second rule, parameters 4 and 5 could actually have been applied six and two times, respectively (due to symmetry in the SMIRKS and the use of wildcards). This situation could also arise as a result of molecular symmetry. For example, a methyl group could match the SMIRKS `[C:1]([H:2])([H:3])([H:4])` six ways (with different orderings of the three hydrogen atoms), but the user would almost certainly not intend for the charge increments to be applied six times. The "regardless of order" clause was added specifically to address this.

In short, the first time a group of atoms becomes involved in a `ChargeIncrement` together, the System gains a new parameter "slot". Only another `ChargeIncrement` which applies to the exact same group of atoms (in any order) can take over the "slot", pushing the original `ChargeIncrement` out.

Major Feature: Support for ProperTorsion k value interpolation

Chaya Stern's work showed that we may be able to produce higher-quality proper torsion parameters by taking into account the "partial bond order" of the torsion's central bond. We now have the machinery to compute AM1-Wiberg partial bond orders for entire molecules using the `assign_fractional_bond_orders` methods of either `OpenEyeToolkitWrapper` or `AmberToolsToolkitWrapper`. The thought is that, if some simple electron population analysis shows that a certain aromatic bond's order is 1.53, maybe rotations about that bond can be described well by interpolating 53% of the way between the single and double bond k values.

Full details of how to define a torsion-interpolating SMIRNOFF force fields are available in the ProperTorsions section of the SMIRNOFF specification.

Behavior changed

- PR #508: In order to provide the same results for the same chemical species, regardless of input conformation, `Molecule` `assign_partial_charges`, `compute_partial_charges_am1bcc`, and `assign_fractional_bond_orders` methods now default to ignore input conformers and generate new conformer(s) of the molecule before running semiempirical calculations. Users can override this behavior by specifying the keyword argument `use_conformers=molecule.conformers`.
- PR #281: Closes Issue #250 by adding support for partial charge I/O in SDF. The partial charges are stored as a property in the SDF molecule block under the tag `<atom.dprop.PartialCharge>`.
- PR #281: If a `Molecule`'s `partial_charges` attribute is set to `None` (the default value), calling `to_openeye` will now produce a OE molecule with partial charges set to `nan`. This would previously produce an OE molecule with partial charges of 0.0, which was a loss of information, since it wouldn't be clear whether the original OFFMol's partial charges were REALLY all-zero as opposed to `None`. OpenEye toolkit wrapper methods such as `from_smiles` and `from_file` now produce OFFMols with `partial_charges = None` when appropriate (previously these would produce OFFMols with all-zero charges, for the same reasoning as above).
- PR #281: `Molecule` `to_rdkit` now sets partial charges on the RDAtom's `PartialCharges` property (this was previously set on the `partial_charges` property). If the `Molecule`'s partial_charges attribute is `None`, this property will not be defined on the RDAtoms.
- PR #281: Enforce the behavior during SDF I/O that a SDF may contain multiple *molecules*, but that the OFF Toolkit does not assume that it contains multiple *conformers of the same molecule*. This is an important distinction, since otherwise there is ambiguity around whether properties of one entry in a SDF are shared among several molecule blocks or not, or how to resolve conflicts if properties are defined differently for several "conformers" of chemically-identical species (More info here). If the user requests the OFF Toolkit to write a multi-conformer `Molecule` to SDF, only the first conformer will be written. For more fine-grained control of writing properties, conformers, and partial charges, consider using `Molecule.to_rdkit` or `Molecule.to_openeye` and using the functionality offered by those packages.
- PR #281: Due to different constraints placed on the data types allowed by external toolkits, we make our best effort to preserve `Molecule` `properties` when converting molecules to other packages, but users should be aware that no guarantee of data integrity is made. The only data format for keys and values in the property dict that we will try to support through a roundtrip to another toolkit's Molecule object is `string`.
- PR #574: Removed check that all partial charges are zero after assignment by `quacpac` when AM1BCC used for charge assignment. This check fails erroneously for cases in which the partial charge assignments are correctly all zero, such as for `N#N`. It is also an unnecessary check given that `quacpac` will reliably indicate when it has failed to assign charges.
- PR #597: Energy-minimized sample systems with Parsley 1.1.0.
- PR #558: The `Topology` particle indexing system now orders `TopologyVirtualSites` after all atoms.
- PR #469: When running `Topology.to_openmm`, unique atom names are generated if the provided atom names are not unique (overriding any existing atom names). This uniqueness extends only to atoms in the same molecule. To disable this behavior, set the kwarg `ensure_unique_atom_names=False`.
- PR #472: `Molecule.__eq__` now uses the new `Molecule.are_isomorphic` to perform the similarity checking.
- PR #472: The `Topology.from_openmm` and `Topology.add_molecule` methods now use the `Molecule.are_isomorphic` method to match molecules.
- PR #551: Implemented the `ParameterHandler.get_parameter` function (would previously return `None`).

API-breaking changes

- PR #471: Closes Issue #465. `atom.formal_charge` and `molecule.total_charge` now return `simtk.unit.Quantity` objects instead of integers. To preserve backward compatibility, the setter for `atom.formal_charge` can accept either a `simtk.unit.Quantity` or an integer.
- PR #601: Removes almost all of the previous `ChemicalEnvironment` API, since this entire module was simply copied from Chemper several years ago and has fallen behind on updates. Currently only `ChemicalEnvironment.get_type`, `ChemicalEnvironment.validate`, and an equivalent classmethod `ChemicalEnvironment.validate_smirks` remain. Also, please comment on this GitHub issue if you HAVE been using the previous extra functionality in this module and would like us to prioritize creation of a Chemper conda package.
- PR #558: Removes `TopologyMolecule.topology_particle_start_index`, since the `Topology` particle indexing system now orders `TopologyVirtualSites` after all atoms. `TopologyMolecule.topology_atom_start_index` and `TopologyMolecule.topology_virtual_site_start_index` are still available to access the appropriate values in the respective topology indexing systems.
- PR #508: `OpenEyeToolkitWrapper.compute_wiberg_bond_orders` is now `OpenEyeToolkitWrapper.assign_fractional_bond_orders`. The `charge_model` keyword is now `bond_order_model`. The allowed values of this keyword have changed from `am1` and `pm3` to `am1-wiberg` and `pm3-wiberg`, respectively.
- PR #508: `Molecule.compute_wiberg_bond_orders` is now `Molecule.assign_fractional_bond_orders`.
- PR #595: Removed functions `openforcefield.utils.utils.temporary_directory` and `openforcefield.utils.utils.temporary_cd` and replaced their behavior with `tempfile.TemporaryDirectory()`.

New features

- PR #471: Closes Issue #208 by implementing support for the `ChargeIncrementModel` tag in the SMIRNOFF specification.
- PR #471: Implements `Molecule.assign_partial_charges`, which calls one of the newly-implemented `OpenEyeToolkitWrapper.assign_partial_charges`, and `AmberToolsToolkitWrapper.assign_partial_charges`. `strict_n_conformers` is a optional boolean keyword argument indicating whether an `IncorrectNumConformersError` should be raised if an invalid number of conformers is supplied during partial charge calculation. For example, if two conformers are supplied, but `partial_charge_method="AM1BCC"` is also set, then there is no clear use for the second conformer. The previous behavior in this case was to raise a warning, and to preserve that behavior, `strict_n_conformers` defaults to a value of `False`.
- PR #471: Adds keyword argument `raise_exception_types` (default: `[Exception]`) to `ToolkitRegistry.call`. The default value will provide the previous OpenFF Toolkit behavior, which is that the first ToolkitWrapper that can provide the requested method is called, and it either returns on success or raises an exception. This new keyword argument allows the ToolkitRegistry to *ignore* certain exceptions, but treat others as fatal. If `raise_exception_types = []`, the ToolkitRegistry will attempt to call each ToolkitWrapper that provides the requested method and if none succeeds, a single `ValueError` will be raised, with text listing the errors that were raised by each ToolkitWrapper.
- PR #601: Adds `RDKitToolkitWrapper.get_tagged_smarts_connectivity` and `OpenEyeToolkitWrapper.get_tagged_smarts_connectivity`, which allow the use of either toolkit for smirks/tagged smarts validation.
- PR #600: Adds `ForceField.__getitem__` to look up `ParameterHandler` objects based on their string names.
- PR #508: Adds `AmberToolsToolkitWrapper.assign_fractional_bond_orders`.

- PR #469: The Molecule class adds Molecule.has_unique_atom_names and Molecule.has_unique_atom_names.
- PR #472: Adds to the Molecule class Molecule.are_isomorphic and Molecule.is_isomorphic_with and Molecule.hill_formula and Molecule.to_hill_formula and Molecule.to_qcschema and Molecule.from_qcschema and Molecule.from_mapped_smiles and Molecule.from_pdb_and_smiles and Molecule.canonical_order_atoms and Molecule.remap

  Note: The to_qcschema method accepts an extras dictionary which is passed into the validated qcelemental.models.Molecule object.

- PR #506: The Molecule class adds Molecule.find_rotatable_bonds
- PR #521: Adds Molecule.to_inchi and Molecule.to_inchikey and Molecule.from_inchi

  Warning: InChI was not designed as an molecule interchange format and using it as one is not recommended. Many round trip tests will fail when using this format due to a loss of information. We have also added support for fixed hydrogen layer nonstandard InChI which can help in the case of tautomers, but overall creating molecules from InChI should be avoided.

- PR #529: Adds the ability to write out to XYZ files via Molecule.to_file Both single frame and multiframe XYZ files are supported. Note reading from XYZ files will not be supported due to the lack of connectivity information.
- PR #535: Extends the the API for the Molecule.to_smiles to allow for the creation of cmiles identifiers through combinations of isomeric, explicit hydrogen and mapped smiles, the default settings will return isomeric explicit hydrogen smiles as expected.

  Warning: Atom maps can be supplied to the properties dictionary to modify which atoms have their map index included, if no map is supplied all atoms will be mapped in the order they appear in the Molecule.

- PR #563: Adds test_forcefields/ion_charges.offxml, giving LibraryCharges for monatomic ions.
- PR #543: Adds 3 new methods to the Molecule class which allow the enumeration of molecule states. These are Molecule.enumerate_tautomers, Molecule.enumerate_stereoisomers, Molecule.enumerate_protomers

  Warning: Enumerate protomers is currently only available through the OpenEye toolkit.

- PR #573: Adds quacpac error output to quacpac failure in Molecule.compute_partial_charges_am1bcc.
- PR #560: Added visualization method to the the Molecule class.
- PR #620: Added the ability to register parameter handlers via entry point plugins. This functionality is accessible by initializing a ForceField with the load_plugins=True keyword argument.
- PR #582: Added fractional bond order interpolation Adds *return_topology* kwarg to Forcefield.create_openmm_system, which returns the processed topology along with the system when True (default False).

Tests added

- PR #558: Adds tests ensuring that the new Topology particle indexing system are properly implemented, and that TopologyVirtualSites reference the correct TopologyAtoms.
- PR #469: Added round-trip SMILES test to add coverage for `Molecule.from_smiles`.
- PR #469: Added tests for unique atom naming behavior in `Topology.to_openmm`, as well as tests of the `ensure_unique_atom_names=False` kwarg disabling this behavior.
- PR #472: Added tests for `Molecule.hill_formula` and `Molecule.to_hill_formula` for the various supported input types.
- PR #472: Added round-trip test for `Molecule.from_qcschema` and `Molecule.to_qcschema`.
- PR #472: Added tests for `Molecule.is_isomorphic_with` and `Molecule.are_isomorphic` with various levels of isomorphic graph matching.
- PR #472: Added toolkit dependent tests for `Molecule.canonical_order_atoms` due to differences in the algorithms used.
- PR #472: Added a test for `Molecule.from_mapped_smiles` using the molecule from issue #412 to ensure it is now fixed.
- PR #472: Added a test for `Molecule.remap`, this also checks for expected error when the mapping is not complete.
- PR #472: Added tests for `Molecule.from_pdb_and_smiles` to check for a correct combination of smiles and PDB and incorrect combinations.
- PR #509: Added test for `Molecule.chemical_environment_matches` to check that the complete set of matches is returned.
- PR #509: Added test for `Forcefield.create_openmm_system` to check that a protein system can be created.
- PR #506: Added a test for the molecule identified in issue #513 as losing aromaticity when converted to rdkit.
- PR #506: Added a verity of toolkit dependent tests for identifying rotatable bonds while ignoring the user requested types.
- PR #521: Added toolkit independent round-trip InChI tests which add coverage for `Molecule.to_inchi` and `Molecule.from_inchi`. Also added coverage for bad inputs and `Molecule.to_inchikey`.
- PR #529: Added to XYZ file coverage tests.
- PR #563: Added *LibraryCharges* parameterization test for monatomic ions in `test_forcefields/ion_charges.offxml`.
- PR #543: Added tests to assure that state enumeration can correctly find molecules tautomers, stereoisomers and protomers when possible.
- PR #573: Added test for `quacpac` error output for `quacpac` failure in `Molecule.compute_partial_charges_am1bcc`.
- PR #579: Adds regression tests to ensure RDKit can be be used to write multi-model PDB files.
- PR #582: Added fractional bond order interpolation tests, tests for `ValidatedDict`.

Bugfixes

- PR #558: Fixes a bug where `TopologyVirtualSite.atoms` would not correctly apply `TopologyMolecule` atom ordering on top of the reference molecule ordering, in cases where the same molecule appears multiple times, but in a different order, in the same Topology.
- Issue #460: Creates unique atom names in `Topology.to_openmm` if the existing ones are not unique. The lack of unique atom names had been causing problems in workflows involving downstream tools that expect unique atom names.
- Issue #448: We can now make molecules from mapped smiles using `Molecule.from_mapped_smiles` where the order will correspond to the indeing used in the smiles. Molecules can also be re-indexed at any time using the `Molecule.remap`.
- Issue #462: We can now instance the `Molecule` from a QCArchive entry record instance or dictionary representation.
- Issue #412: We can now instance the `Molecule` using `Molecule.from_mapped_smiles`. This resolves an issue caused by RDKit considering atom map indices to be a distinguishing feature of an atom, which led to erroneous definition of chirality (as otherwise symmetric substituents would be seen as different). We anticipate that this will reduce the number of times you need to type `allow_undefined_stereo=True` when processing molecules that do not actually contain stereochemistrty.
- Issue #513: The `Molecule.to_rdkit` now re-sets the aromaticity model after sanitizing the molecule.
- Issue #500: The `Molecule.find_rotatable_bonds` has been added which returns a list of rotatable `Bond` instances for the molecule.
- Issue #491: We can now parse large molecules without hitting a match limit cap.
- Issue #474: We can now convert molecules to InChI and InChIKey and from InChI.
- Issue #523: The `Molecule.to_file` method can now correctly write to `MOL` files, in line with the supported file type list.
- Issue #568: The `Molecule.to_file` can now correctly write multi-model PDB files when using the RDKit backend toolkit.

Examples added

- PR #591 and PR #533: Adds an example notebook and utility to compute conformer energies. This example is made to be reverse-compatible with the 0.6.0 OpenFF Toolkit release.
- PR #472: Adds an example notebook QCarchive_interface.ipynb which shows users how to instance the `Molecule` from a QCArchive entry level record and calculate the energy using RDKit through QCEngine.

1.2.2 0.6.0 - Library Charges

This release adds support for a new SMIRKS-based charge assignment method, Library Charges. The addition of more charge assignment methods opens the door for new types of experimentation, but also introduces several complex behaviors and failure modes. Accordingly, we have made changes to the charge assignment infrastructure to check for cases when partial charges do not sum to the formal charge of the molecule, or when no charge assignment method is able to generate charges for a molecule. More detailed explanation of the new errors that may be raised and keywords for overriding them are in the "Behavior Changed" section below.

With this release, we update `test_forcefields/tip3p.offxml` to be a working example of assigning LibraryCharges. However, we do not provide any force field files to assign protein residue `LibraryCharges`. If you are interested in translating an existing protein FF to SMIRNOFF format or developing a new one, please feel free to contact us on the Issue tracker or open a Pull Request.

New features

- PR #433: Closes Issue #25 by adding initial support for the LibraryCharges tag in the SMIRNOFF specification using `LibraryChargeHandler`. For a molecule to have charges assigned using Library Charges, all of its atoms must be covered by at least one `LibraryCharge`. If an atom is covered by multiple `LibraryCharge` s, then the last `LibraryCharge` matched will be applied (per the hierarchy rules in the SMIRNOFF format).

  This functionality is thus able to apply per-residue charges similar to those in traditional protein force fields. At this time, there is no concept of "residues" or "fragments" during parametrization, so it is not possible to assign charges to *some* atoms in a molecule using `LibraryCharge` s, but calculate charges for other atoms in the same molecule using a different method. To assign charges to a protein, LibraryCharges SMARTS must be provided for the residues and protonation states in the molecule, as well as for any capping groups and post-translational modifications that are present.

  It is valid for `LibraryCharge` SMARTS to *partially* overlap one another. For example, a molecule consisting of atoms `A-B-C` connected by single bonds could be matched by a SMIRNOFF `LibraryCharges` section containing two `LibraryCharge` SMARTS: `A-B` and `B-C`. If listed in that order, the molecule would be assigned the `A` charge from the `A-B` `LibraryCharge` element and the `B` and `C` charges from the `B-C` element. In testing, these types of partial overlaps were found to frequently be sources of undesired behavior, so it is recommended that users define whole-molecule `LibraryCharge` SMARTS whenever possible.
- PR #455: Addresses Issue #393 by adding `ParameterHandler.attribute_is_cosmetic` and `ParameterType.attribute_is_cosmetic`, which return True if the provided attribute name is defined for the queried object but does not correspond to an allowed value in the SMIRNOFF spec.

Behavior changed

- PR #433: If a molecule can not be assigned charges by any charge-assignment method, an `openforcefield.typing.engines.smirnoff.parameters.UnassignedMoleculeChargeException` will be raised. Previously, creating a system without either `ToolkitAM1BCCHandler` or the `charge_from_molecules` keyword argument to `ForceField.create_openmm_system` would produce a system where the molecule has zero charge on all atoms. However, given that we will soon be adding more options for charge assignment, it is important that failures not be silent. Molecules with zero charge can still be produced by setting the `Molecule.partial_charges` array to be all zeroes, and including the molecule in the `charge_from_molecules` keyword argument to `create_openmm_system`.
- PR #433: Due to risks introduced by permitting charge assignment using partially-overlapping `LibraryCharge` s, the toolkit will now raise a `openforcefield.typing.engines.smirnoff.parameters.NonIntegralMoleculeChargeException` if the sum of partial charges on a molecule are found to be more than 0.01 elementary charge units different than the molecule's formal charge. This exception can be overridden by providing the `allow_nonintegral_charges=True` keyword argument to `ForceField.create_openmm_system`.

Tests added

- PR #430: Added test for Wiberg Bond Order implemented in OpenEye Toolkits. Molecules taken from DOI:10.5281/zenodo.3405489 . Added by Sukanya Sasmal.
- PR #569: Added round-trip tests for more serialization formats (dict, YAML, TOML, JSON, BSON, messagepack, pickle). Note that some are unsupported, but the tests raise the appropriate error.

Bugfixes

- PR #431: Fixes an issue where `ToolkitWrapper` objects would improperly search for functionality in the `GLOBAL_TOOLKIT_REGISTRY`, even though a specific `ToolkitRegistry` was requested for an operation.
- PR #439: Fixes Issue #438, by replacing call to NetworkX `Graph.node` with call to `Graph.nodes`, per 2.4 migration guide.

Files modified

- PR #433: Updates the previously-nonfunctional `test_forcefields/tip3p.offxml` to a functional state by updating it to the SMIRNOFF 0.3 specification, and specifying atomic charges using the `LibraryCharges` tag.

1.2.3 0.5.1 - Adding the parameter coverage example notebook

This release contains a new notebook example, check_parameter_coverage.ipynb, which loads sets of molecules, checks whether they are parameterizable, and generates reports of chemical motifs that are not. It also fixes several simple issues, improves warnings and docstring text, and removes unused files.

The parameter coverage example notebook goes hand-in-hand with the release candidate of our initial force field, openff-1.0.0-RC1.offxml , which will be temporarily available until the official force field release is made in October. Our goal in publishing this notebook alongside our first major refitting is to allow interested users to check whether there is parameter coverage for their molecules of interest. If the force field is unable to parameterize a molecule, this notebook will generate reports of the specific chemistry that is not covered. We understand that many organizations in our field have restrictions about sharing specific molecules, and the outputs from this notebook can easily be cropped to communicate unparameterizable chemistry without revealing the full structure.

The force field release candidate is in our new refit force field package, openforcefields. This package is now a part of the Open Force Field Toolkit conda recipe, along with the original smirnoff99Frosst line of force fields.

Once the `openforcefields` conda package is installed, you can load the release candidate using:

```
ff = ForceField('openff-1.0.0-RC1.offxml')
```

The release candidate will be removed when the official force field, `openff-1.0.0.offxml`, is released in early October.

Complete details about this release are below.

Example added

- PR #419: Adds an example notebook check_parameter_coverage.ipynb which shows how to use the toolkit to check a molecule dataset for missing parameter coverage, and provides functionality to output tagged SMILES and 2D drawings of the unparameterizable chemistry.

New features

- PR #419: Unassigned valence parameter exceptions now include a list of tuples of `TopologyAtom` which were unable to be parameterized (`exception.unassigned_topology_atom_tuples`) and the class of the `ParameterHandler` that raised the exception (`exception.handler_class`).
- PR #425: Implements Trevor Gokey's suggestion from Issue #411, which enables pickling of `ForceFields` and `ParameterHandlers`. Note that, while XML representations of ``ForceField``s are stable and conform to the SMIRNOFF specification, the pickled ``ForceField``s that this functionality enables are not guaranteed to be compatible with future toolkit versions.

Improved documentation and warnings

- PR #425: Addresses Issue #410, by explicitly having toolkit warnings print `Warning:` at the beginning of each warning, and adding clearer language to the warning produced when the OpenEye Toolkits can not be loaded.
- PR #425: Addresses Issue #421 by adding type/shape information to all Molecule partial charge and conformer docstrings.
- PR #425: Addresses Issue #407 by providing a more extensive explanation of why we don't use RDKit's mol2 parser for molecule input.

Bugfixes

- PR #419: Fixes Issue #417 and Issue #418, where `RDKitToolkitWrapper.from_file` would disregard the `allow_undefined_stereo` kwarg and skip the first molecule when reading a SMILES file.

Files removed

- PR #425: Addresses Issue #424 by deleting the unused files `openforcefield/typing/engines/smirnoff/gbsaforces.py` and `openforcefield/tests/test_smirnoff.py`. `gbsaforces.py` was only used internally and `test_smirnoff.py` tested unsupported functionality from before the 0.2.0 release.

1.2.4 0.5.0 - GBSA support and quality-of-life improvements

This release adds support for the GBSA tag in the SMIRNOFF specification. Currently, the `HCT`, `OBC1`, and `OBC2` models (corresponding to AMBER keywords `igb=1`, `2`, and `5`, respectively) are supported, with the `OBC2` implementation being the most flexible. Unfortunately, systems produced using these keywords are not yet transferable to other simulation packages via ParmEd, so users are restricted to using OpenMM to simulate systems with GBSA.

OFFXML files containing GBSA parameter definitions are available, and can be loaded in addition to existing parameter sets (for example, with the command `ForceField('test_forcefields/smirnoff99Frosst.offxml', 'test_forcefields/GBSA_OBC1-1.0.offxml')`). A manifest of new SMIRNOFF-format GBSA files is below.

Several other user-facing improvements have been added, including easier access to indexed attributes, which are now accessible as `torsion.k1`, `torsion.k2`, etc. (the previous access method `torsion.k` still works as well). More details of the new features and several bugfixes are listed below.

New features

- PR #363: Implements `GBSAHandler`, which supports the GBSA tag in the SMIRNOFF specification. Currently, only GBSAHandlers with `gb_model="OBC2"` support setting non-default values for the `surface_area_penalty` term (default `5.4*calories/mole/angstroms**2`), though users can zero the SA term for `OBC1` and `HCT` models by setting `sa_model="None"`. No model currently supports setting `solvent_radius` to any value other than `1.4*angstroms`. Files containing experimental SMIRNOFF-format implementations of `HCT`, `OBC1`, and `OBC2` are included with this release (see below). Additional details of these models, including literature references, are available on the SMIRNOFF specification page.

  > **Warning:** The current release of ParmEd can not transfer GBSA models produced by the Open Force Field Toolkit to other simulation packages. These GBSA forces are currently only computable using OpenMM.

- PR #363: When using `Topology.to_openmm()`, periodic box vectors are now transferred from the Open Force Field Toolkit Topology into the newly-created OpenMM Topology.
- PR #377: Single indexed parameters in `ParameterHandler` and `ParameterType` can now be get/set through normal attribute syntax in addition to the list syntax.
- PR #394: Include element and atom name in error output when there are missing valence parameters during molecule parameterization.

Bugfixes

- PR #385: Fixes Issue #346 by having `OpenEyeToolkitWrapper.compute_partial_charges_am1bcc` fall back to using standard AM1-BCC if AM1-BCC ELF10 charge generation raises an error about "trans COOH conformers"
- PR #399: Fixes issue where `ForceField` constructor would ignore `parameter_handler_classes` kwarg.
- PR #400: Makes link-checking tests retry three times before failing.

Files added

- PR #363: Adds `test_forcefields/GBSA_HCT-1.0.offxml`, `test_forcefields/GBSA_OBC1-1.0.offxml`, and `test_forcefields/GBSA_OBC2-1.0.offxml`, which are experimental implementations of GBSA models. These are primarily used in validation tests against OpenMM's models, and their version numbers will increment if bugfixes are necessary.

1.2.5 0.4.1 - Bugfix Release

This update fixes several toolkit bugs that have been reported by the community. Details of these bugfixes are provided below.

It also refactors how `ParameterType` and `ParameterHandler` store their attributes, by introducing `ParameterAttribute` and `IndexedParameterAttribute`. These new attribute-handling classes provide a consistent backend which should simplify manipulation of parameters and implementation of new handlers.

Bug fixes

- PR #329: Fixed a bug where the two `BondType` parameter attributes `k` and `length` were treated as indexed attributes. (`k` and `length` values that correspond to specific bond orders will be indexed under `k_bondorder1`, `k_bondorder2`, etc when implemented in the future)
- PR #329: Fixed a bug that allowed setting indexed attributes to single values instead of strictly lists.
- PR #370: Fixed a bug in the API where `BondHandler`, `ProperTorsionHandler` , and `ImproperTorsionHandler` exposed non-functional indexed parameters.
- PR #351: Fixes Issue #344, in which the main `FrozenMolecule` constructor and several other Molecule-construction functions ignored or did not expose the `allow_undefined_stereo` keyword argument.
- PR #351: Fixes a bug where a molecule which previously generated a SMILES using one cheminformatics toolkit returns the same SMILES, even though a different toolkit (which would generate a different SMILES for the molecule) is explicitly called.
- PR #354: Fixes the error message that is printed if an unexpected parameter attribute is found while loading data into a `ForceField` (now instructs users to specify `allow_cosmetic_attributes` instead of `permit_cosmetic_attributes`)
- PR #364: Fixes Issue #362 by modifying `OpenEyeToolkitWrapper.from_smiles` and `RDKitToolkitWrapper.from_smiles` to make implicit hydrogens explicit before molecule creation. These functions also now raise an error if the optional keyword `hydrogens_are_explicit=True` but the SMILES are interpreted by the backend cheminformatic toolkit as having implicit hydrogens.
- PR #371: Fixes error when reading early SMIRNOFF 0.1 spec files enclosed by a top-level `SMIRFF` tag.

Note: The enclosing `SMIRFF` tag is present only in legacy files. Since developing a formal specification, the only acceptable top-level tag value in a SMIRNOFF data structure is `SMIRNOFF`.

Code enhancements

- PR #329: `ParameterType` was refactored to improve its extensibility. It is now possible to create new parameter types by using the new descriptors `ParameterAttribute` and `IndexedParameterAttribute`.
- PR #357: Addresses Issue #356 by raising an informative error message if a user attempts to load an OpenMM topology which is probably missing connectivity information.

Force fields added

- PR #368: Temporarily adds `test_forcefields/smirnoff99frosst_experimental.offxml` to address hierarchy problems, redundancies, SMIRKS pattern typos etc., as documented in issue #367. Will ultimately be propagated to an updated forcefield in the `openforcefield/smirnoff99frosst` repo.
- PR #371: Adds `test_forcefields/smirff99Frosst_reference_0_1_spec.offxml`, a SMIRNOFF 0.1 spec file enclosed by the legacy `SMIRFF` tag. This file is used in backwards-compatibility testing.

1.2.6 0.4.0 - Performance optimizations and support for SMIRNOFF 0.3 specification

This update contains performance enhancements that significantly reduce the time to create OpenMM systems for topologies containing many molecules via `ForceField.create_openmm_system`.

This update also introduces the SMIRNOFF 0.3 specification. The spec update is the result of discussions about how to handle the evolution of data and parameter types as further functional forms are added to the SMIRNOFF spec.

We provide methods to convert SMIRNOFF 0.1 and 0.2 forcefields written with the XML serialization (`.offxml`) to the SMIRNOFF 0.3 specification. These methods are called automatically when loading a serialized SMIRNOFF data representation written in the 0.1 or 0.2 specification. This functionality allows the toolkit to continue to read files containing SMIRNOFF 0.2 spec force fields, and also implements backwards-compatibility for SMIRNOFF 0.1 spec force fields.

Warning: The SMIRNOFF 0.1 spec did not contain fields for several energy-determining parameters that are exposed in later SMIRNOFF specs. Thus, when reading SMIRNOFF 0.1 spec data, the toolkit must make assumptions about the values that should be added for the newly-required fields. The values that are added include 1-2, 1-3 and 1-5 scaling factors, cutoffs, and long-range treatments for nonbonded interactions. Each assumption is printed as a warning during the conversion process. Please carefully review the warning messages to ensure that the conversion is providing your desired behavior.

SMIRNOFF 0.3 specification updates

- The SMIRNOFF 0.3 spec introduces versioning for each individual parameter section, allowing asynchronous updates to the features of each parameter class. The top-level `SMIRNOFF` tag, containing information like `aromaticity_model`, `Author`, and `Date`, still has a version (currently 0.3). But, to allow for independent development of individual parameter types, each section (such as `Bonds`, `Angles`, etc) now has its own version as well (currently all 0.3).
- All units are now stored in expressions with their corresponding values. For example, distances are now stored as `1.526*angstrom`, instead of storing the unit separately in the section header.
- The current allowed value of the `potential` field for `ProperTorsions` and `ImproperTorsions` tags is no longer `charmm`, but is rather `k*(1+cos(periodicity*theta-phase))`. It was pointed out to us that CHARMM-style torsions deviate from this formula when the periodicity of a torsion term is 0, and we do not intend to reproduce that behavior.
- SMIRNOFF spec documentation has been updated with tables of keywords and their defaults for each parameter section and parameter type. These tables will track the allowed keywords and default behavior as updated versions of individual parameter sections are released.

Performance improvements and bugfixes

- PR #329: Performance improvements when creating systems for topologies with many atoms.
- PR #347: Fixes bug in charge assignment that occurs when charges are read from file, and reference and charge molecules have different atom orderings.

New features

- PR #311: Several new experimental functions.
  - Adds `convert_0_2_smirnoff_to_0_3`, which takes a SMIRNOFF 0.2-spec data dict, and updates it to 0.3. This function is called automatically when creating a `ForceField` from a SMIRNOFF 0.2 spec OFFXML file.
  - Adds `convert_0_1_smirnoff_to_0_2`, which takes a SMIRNOFF 0.1-spec data dict, and updates it to 0.2. This function is called automatically when creating a `ForceField` from a SMIRNOFF 0.1 spec OFFXML file.
  - NOTE: The format of the "SMIRNOFF data dict" above is likely to change significantly in the future. Users that require a stable serialized ForceField object should use the output of `ForceField.to_string('XML')` instead.
  - Adds `ParameterHandler` and `ParameterType` `add_cosmetic_attribute` and `delete_cosmetic_attribute` functions. Once created, cosmetic attributes can be accessed and modified as attributes of the underlying object (eg. `ParameterType.my_cosmetic_attrib = 'blue'`) These functions are experimental, and we are interested in feedback on how cosmetic attribute handling could be improved. (See Issue #338) Note that if a new cosmetic attribute is added to an object without using these functions, it will not be recognized by the toolkit and will not be written out during serialization.
  - Values for the top-level `Author` and `Date` tags are now kept during SMIRNOFF data I/O. If multiple data sources containing these fields are read, the values are concatenated using "AND" as a separator.

API-breaking changes

- `ForceField.to_string` and `ForceField.to_file` have had the default value of their `discard_cosmetic_attributes` kwarg set to False.
- `ParameterHandler` and `ParameterType` constructors now expect the `version` kwarg (per the SMIRNOFF spec change above) This requirement can be skipped by providing the kwarg `skip_version_check=True`
- `ParameterHandler` and `ParameterType` functions no longer handle `X_unit` attributes in SMIRNOFF data (per the SMIRNOFF spec change above).
- The scripts in `utilities/convert_frosst` are now deprecated. This functionality is important for provenance and will be migrated to the `openforcefield/smirnoff99Frosst` repository in the coming weeks.
- `ParameterType._SMIRNOFF_ATTRIBS` is now `ParameterType._REQUIRED_SPEC_ATTRIBS`, to better parallel the structure of the `ParameterHandler` class.
- `ParameterType._OPTIONAL_ATTRIBS` is now `ParameterType._OPTIONAL_SPEC_ATTRIBS`, to better parallel the structure of the `ParameterHandler` class.
- Added class-level dictionaries `ParameterHandler._DEFAULT_SPEC_ATTRIBS` and `ParameterType._DEFAULT_SPEC_ATTRIBS`.

1.2.7 0.3.0 - API Improvements

Several improvements and changes to public API.

New features

- PR #292: Implement `Topology.to_openmm` and remove `ToolkitRegistry.toolkit_is_available`
- PR #322: Install directories for the lookup of OFFXML files through the entry point group `openforcefield.smirnoff_forcefield_directory`. The `ForceField` class doesn't search in the `data/forcefield/` folder anymore (now renamed `data/test_forcefields/`), but only in `data/`.

API-breaking Changes

- PR #278: Standardize variable/method names
- PR #291: Remove `ForceField.load/to_smirnoff_data`, add `ForceField.to_file/string` and `ParameterHandler.add_parameters`. Change behavior of `ForceField.register_X_handler` functions.

Bugfixes

- PR #327: Fix units in tip3p.offxml (note that this file is still not loadable by current toolkit)
- PR #325: Fix solvent box for provided test system to resolve periodic clashes.
- PR #325: Add informative message containing Hill formula when a molecule can't be matched in `Topology.from_openmm`.
- PR #325: Provide warning or error message as appropriate when a molecule is missing stereochemistry.
- PR #316: Fix formatting issues in GBSA section of SMIRNOFF spec
- PR #308: Cache molecule SMILES to improve system creation speed
- PR #306: Allow single-atom molecules with all zero coordinates to be converted to OE/RDK mols
- PR #313: Fix issue where constraints are applied twice to constrained bonds

1.2.8 0.2.2 - Bugfix release

This release modifies an example to show how to parameterize a solvated system, cleans up backend code, and makes several improvements to the README.

Bugfixes

- PR #279: Cleanup of unused code/warnings in main package `__init__`
- PR #259: Update T4 Lysozyme + toluene example to show how to set up solvated systems
- PR #256 and PR #274: Add functionality to ensure that links in READMEs resolve successfully

1.2.9 0.2.1 - Bugfix release

This release features various documentation fixes, minor bugfixes, and code cleanup.

Bugfixes

- PR #267: Add neglected `<ToolkitAM1BCC>` documentation to the SMIRNOFF 0.2 spec
- PR #258: General cleanup and removal of unused/inaccessible code.
- PR #244: Improvements and typo fixes for BRD4:inhibitor benchmark

1.2.10 0.2.0 - Initial RDKit support

This version of the toolkit introduces many new features on the way to a 1.0.0 release.

New features

- Major overhaul, resulting in the creation of the SMIRNOFF 0.2 specification and its XML representation
- Updated API and infrastructure for reference SMIRNOFF `ForceField` implementation
- Implementation of modular `ParameterHandler` classes which process the topology to add all necessary forces to the system.
- Implementation of modular `ParameterIOHandler` classes for reading/writing different serialized SMIRNOFF forcefield representations
- Introduction of `Molecule` and `Topology` classes for representing molecules and biomolecular systems
- New `ToolkitWrapper` interface to RDKit, OpenEye, and AmberTools toolkits, managed by `ToolkitRegistry`
- API improvements to more closely follow PEP8 guidelines
- Improved documentation and examples

1.2.11 0.1.0

This is an early preview release of the toolkit that matches the functionality described in the preprint describing the SMIRNOFF v0.1 force field format: [DOI].

New features

This release features additional documentation, code comments, and support for automated testing.

Bugfixes

Treatment of improper torsions

A significant (though currently unused) problem in handling of improper torsions was corrected. Previously, non-planar impropers did not behave correctly, as six-fold impropers have two potential chiralities. To remedy this, SMIRNOFF impropers are now implemented as three-fold impropers with consistent chirality. However, current force fields in the SMIRNOFF format had no non-planar impropers, so this change is mainly aimed at future work.

1.3 The SMIRks Native Open Force Field (SMIRNOFF) specification

SMIRNOFF is a specification for encoding molecular mechanics force fields from the Open Force Field Initiative based on direct chemical perception using the broadly-supported SMARTS language, utilizing atom tagging extensions from SMIRKS.

1.3.1 Authors and acknowledgments

The SMIRNOFF specification was designed by the Open Force Field Initiative.

Primary contributors include:

- Caitlin C. Bannan (University of California, Irvine) <bannanc@uci.edu>
- Christopher I. Bayly (OpenEye Software) <bayly@eyesopen.com>
- John D. Chodera (Memorial Sloan Kettering Cancer Center) <john.chodera@choderalab.org>
- David L. Mobley (University of California, Irvine) <dmobley@uci.edu>

SMIRNOFF and its reference implementation in the `openforcefield` toolkit was heavily inspired by the ForceField class from the OpenMM molecular simulation package, and its associated XML format, developed by Peter K. Eastman (Stanford University).

1.3.2 Representations and encodings

A force field in the SMIRNOFF format can be encoded in multiple representations. Currently, only an XML representation is supported by the reference implementation of the openforcefield toolkit.

XML representation

A SMIRNOFF force field can be described in an XML representation, which provides a human- and machine-readable form for encoding the parameter set and its typing rules. This document focuses on describing the XML representation of the force field.

- By convention, XML-encoded SMIRNOFF force fields use an `.offxml` extension if written to a file to prevent confusion with other file formats.
- In XML, numeric quantities appear as strings, like `"1"` or `"2.3"`.
- Integers should always be written without a decimal point, such as `"1"`, `"9"`.
- Non-integral numbers, such as parameter values, should be written with a decimal point, such as `"1.23"`, `"2."`.

- In XML, certain special characters that occur in valid SMARTS/SMIRKS patterns (such as ampersand symbols `&`) must be specially encoded. See this list of XML and HTML character entity references for more details.

Future representations: JSON, MessagePack, YAML, and TOML

We are considering supporting JSON, MessagePack, YAML, and TOML representations as well.

1.3.3 Reference implementation

A reference implementation of the SMIRNOFF XML specification is provided in the openforcefield toolkit.

1.3.4 Support for molecular simulation packages

The reference implementation currently generates parameterized molecular mechanics systems for the GPU-accelerated OpenMM molecular simulation toolkit. Parameterized systems can subsequently be converted for use in other popular molecular dynamics simulation packages (including AMBER, CHARMM, NAMD, Desmond, and LAMMPS) via ParmEd and InterMol. See Converting SMIRNOFF parameterized systems to other simulation packages for more details.

1.3.5 Basic structure

A reference implementation of a SMIRNOFF force field parser that can process XML representations (denoted by `.offxml` file extensions) can be found in the `ForceField` class of the `openforcefield.typing.engines.smirnoff` module.

Below, we describe the main structure of such an XML representation.

The enclosing <SMIRNOFF> tag

A SMIRNOFF forcefield XML specification always is enclosed in a `<SMIRNOFF>` tag, with certain required attributes provided. The required and permitted attributes defined in the `<SMIRNOFF>` are recorded in the version attribute, which describes the top-level attributes that are expected or permitted to be defined.

```
<SMIRNOFF version="0.3" aromaticity_model="OEAroModel_MDL">
...
</SMIRNOFF>
```

Versioning

The SMIRNOFF force field format supports versioning via the `version` attribute to the root `<SMIRNOFF>` tag, e.g.:

```
<SMIRNOFF version="0.3" aromaticity_model="OEAroModel_MDL">
...
</SMIRNOFF>
```

The version format is `x.y`, where `x` denotes the major version and `y` denotes the minor version. SMIRNOFF versions are guaranteed to be backward-compatible within the *same major version number series*, but it is possible major version increments will break backwards-compatibility.

SMIRNOFF tag version	Required attributes	Optional attributes
0.1	`aromaticity_model`	`Date`, `Author`
0.2	`aromaticity_model`	`Date`, `Author`
0.3	`aromaticity_model`	`Date`, `Author`

The SMIRNOFF tag versions describe the required and allowed force field-wide settings. The list of keywords is as follows:

Aromaticity model

The `aromaticity_model` specifies the aromaticity model used for chemical perception (here, `OEAroModel_MDL`).

Currently, the only supported model is `OEAroModel_MDL`, which is implemented in both the RDKit and the OpenEye Toolkit.

Note: Add link to complete open specification of *OEAroModel_MDL* aromaticity model.

Metadata

Typically, date and author information is included:

```
<Date>2016-05-25</Date>
<Author>J. D. Chodera (MSKCC) charge increment tests</Author>
```

The `<Date>` tag should conform to ISO 8601 date formatting guidelines, such as `2018-07-14` or `2018-07-14T08:50:48+00:00` (UTC time).

Parameter generators

Within the `<SMIRNOFF>` tag, top-level tags encode parameters for a force field based on a SMARTS/SMIRKS-based specification describing the chemical environment the parameters are to be applied to. The file has tags corresponding to OpenMM force terms (`Bonds`, `Angles`, `ProperTorsions`, etc., as discussed in more detail below); these specify functional form and other information for individual force terms.

```
<Angles version="0.3" potential="harmonic">
   ...
</Angles>
```

which introduces the following `Angle` child elements which will use a harmonic potential.

Specifying parameters

Under each of these force terms, there are tags for individual parameter lines such as these:

```
<Angles version="0.3" potential="harmonic">
   <Angle smirks="[a,A:1]-[#6X4:2]-[a,A:3]" angle="109.50*degree" k="100.0*kilocalorie_per_mole/
↪radian**2"/>
   <Angle smirks="[#1:1]-[#6X4:2]-[#1:3]" angle="109.50*degree" k="70.0*kilocalorie_per_mole/radian**2"/
↪>
</Angles>
```

The first of these specifies the `smirks` attribute as `[a,A:1]-[#6X4:2]-[a,A:3]`, specifying a SMIRKS pattern that matches three connected atoms specifying an angle. This particular SMIRKS pattern matches a tetravalent carbon at the center with single bonds to two atoms of any type. This pattern is essentially a SMARTS string with numerical atom tags commonly used in SMIRKS to identify atoms in chemically unique environments—these can be thought of as tagged regular expressions for identifying chemical environments, and atoms within those environments. Here, `[a,A]` denotes any atom—either aromatic (`a`) or aliphatic (`A`), while `[#6X4]` denotes a carbon by element number (`#6`) that with four substituents (`X4`). The symbol `-` joining these groups denotes a single bond. The strings `:1`, `:2`, and `:2` label these atoms as indices 1, 2, and 3, with 2 being the central atom. Equilibrium angles are provided as the `angle` attribute, along with force constants as the `k` attribute (with corresponding units included in the expression).

Note: The reference implementation of the SMIRNOFF specification implemented in the Open Force Field toolkit will, by default, raise an exception if an unexpected attribute is encountered. The toolkit can be configured to accept non-spec keywords, but these are considered "cosmetic" and will not be evaluated. For example, providing an *<Angle>* tag that also specifies a second force constant *k2* will result in an exception, unless the user specifies that "cosmetic" attributes should be accepted by the parser.

SMIRNOFF parameter specification is hierarchical

Parameters that appear later in a SMIRNOFF specification override those which come earlier if they match the same pattern. This can be seen in the example above, where the first line provides a generic angle parameter for any tetravalent carbon (single bond) angle, and the second line overrides this for the specific case of a hydrogen-(tetravalent carbon)-hydrogen angle. This hierarchical structure means that a typical parameter file will tend to have generic parameters early in the section for each force type, with more specialized parameters assigned later.

Multiple SMIRNOFF representations can be processed in sequence

Multiple SMIRNOFF data sources (e.g. multiple OFFXML files) can be loaded by the openforcefield `ForceField` in sequence. If these files each contain unique top-level tags (such as `<Bonds>`, `<Angles>`, etc.), the resulting forcefield will be independent of the order in which the files are loaded. If, however, the same tag occurs in multiple files, the contents of the tags are merged, with the tags read later taking precedence over the parameters read earlier, provided the top-level tags have compatible attributes. The resulting force field will therefore depend on the order in which parameters are read.

This behavior is intended for limited use in appending very specific parameters, such as parameters specifying solvent models, to override standard parameters.

1.3.6 Units

To minimize the potential for unit conversion errors, SMIRNOFF forcefields explicitly specify units in a form readable to both humans and computers for all unit-bearing quantities. Allowed values for units are given in simtk.unit (though in the future this may change to the more widely-used Python pint library). For example, for the `angle` (equilibrium angle) and `k` (force constant) parameters in the `<Angle>` example block above, both attributes are specified as a mathematical expression

```
<Angle smirks="[#1:1]-[#6X4:2]-[#1:3]" angle="109.50*degree" k="70.0*kilocalorie_per_mole/radian**2"/>
```

For more information, see the standard OpenMM unit system.

1.3.7 SMIRNOFF independently applies parameters to each class of potential energy terms

The SMIRNOFF uses direct chemical perception to assign parameters for potential energy terms independently for each term. Rather than first applying atom typing rules and then looking up combinations of the resulting atom types for each force term, the rules for directly applying parameters to atoms is compartmentalized in separate sections. The file consists of multiple top-level tags defining individual components of the potential energy (in addition to charge models or modifiers), with each section specifying the typing rules used to assign parameters for that potential term:

```
<Bonds version="0.3" potential="harmonic">
   <Bond smirks="[#6X4:1]-[#6X4:2]" length="1.526*angstrom" k="620.0*kilocalories_per_mole/angstrom**2"/
↪>
   <Bond smirks="[#6X4:1]-[#1:2]" length="1.090*angstrom" k="680.0*kilocalories_per_mole/angstrom**2"/>
   ...
</Bonds>

<Angles version="0.3" potential="harmonic">
   <Angle smirks="[a,A:1]-[#6X4:2]-[a,A:3]" angle="109.50*degree" k="100.0*kilocalories_per_mole/
↪radian**2"/>
   <Angle smirks="[#1:1]-[#6X4:2]-[#1:3]" angle="109.50*degree" k="70.0*kilocalories_per_mole/radian**2
↪"/>
   ...
</Angles>
```

Each top-level tag specifying a class of potential energy terms has an attribute `potential` for specifying the functional form for the interaction. Common defaults are defined, but the goal is to eventually allow these to be overridden by alternative choices or even algebraic expressions in the future, once more molecular simulation packages support general expressions. We distinguish between functional forms available in all common molecular simulation packages (specified by keywords) and support for general functional forms available in a few packages (especially OpenMM, which supports a flexible set of custom forces defined by algebraic expressions) with an **EXPERIMENTAL** label.

Many of the specific forces are implemented as discussed in the OpenMM Documentation; see especially Section 19 on Standard Forces for mathematical descriptions of these functional forms. Some top-level tags provide attributes that modify the functional form used to be consistent with packages such as AMBER or CHARMM.

1.3.8 Partial charge and electrostatics models

SMIRNOFF supports several approaches to specifying electrostatic models. Currently, only classical fixed point charge models are supported, but future extensions to the specification will support point multipoles, point polarizable dipoles, Drude oscillators, charge equilibration methods, and so on.

<LibraryCharges>: Library charges for polymeric residues and special solvent models

A mechanism is provided for specifying library charges that can be applied to molecules or residues that match provided templates. Library charges are applied first, and atoms for which library charges are applied will be excluded from alternative charging schemes listed below.

For example, to assign partial charges for a non-terminal ALA residue from the AMBER ff14SB parameter set:

```
<LibraryCharges version="0.3">
   <!-- match a non-terminal alanine residue with AMBER ff14SB partial charges -->
   <LibraryCharge name="ALA" smirks="[NX3:1]([#1:2])([#6])[#6H1:3]([#1:4])([#6:5]([#1:6])([#1:7])[#1:
↪8])[#6:9](=[#8:10])[#7]" charge1="-0.4157*elementary_charge" charge2="0.2719*elementary_charge"␣
↪charge3="0.0337*elementary_charge" charge4="0.0823*elementary_charge" charge5="-0.1825*elementary_
↪charge" charge6="0.0603*elementary_charge" charge7="0.0603*elementary_charge" charge8="0.
↪0603*elementary_charge" charge9="0.5973*elementary_charge" charge10="-0.5679*elementary_charge"/>
   ...
</LibraryCharges>
```

In this case, a SMIRKS string defining the residue tags each atom that should receive a partial charge, with the charges specified by attributes `charge1`, `charge2`, etc. The `name` attribute is optional. Note that, for a given template, chemically equivalent atoms should be assigned the same charge to avoid undefined behavior. If the template matches multiple non-overlapping sets of atoms, all such matches will be assigned the provided charges. If multiple templates match the same set of atoms, the last template specified will be used.

Solvent models or excipients can also have partial charges specified via the `<LibraryCharges>` tag. For example, to ensure water molecules are assigned partial charges for TIP3P water, we can specify a library charge entry:

```
<LibraryCharges version="0.3">
   <!-- TIP3P water oxygen with charge override -->
   <LibraryCharge name="TIP3P" smirks="[#1:1]-[#8X2H2+0:2]-[#1:3]" charge1="0.417*elementary_charge"␣
↪charge2="-0.834*elementary_charge" charge3="0.417*elementary_charge"/>
</LibraryCharges>
```

LibraryCharges section tag version	Tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3		`smirks`, `charge` (indexed)	`name`, `id`, `parent_id`

<ChargeIncrementModel>: Small molecule and fragment charges

In keeping with the AMBER force field philosophy, especially as implemented in small molecule force fields such as GAFF, GAFF2, and parm@Frosst, partial charges for small molecules are usually assigned using a quantum chemical method (usually a semiempirical method such as AM1) and a partial charge determination scheme (such as CM2 or RESP), then subsequently corrected via charge increment rules, as in the highly successful AM1-BCC approach.

Here is an example:

```
<ChargeIncrementModel version="0.3" number_of_conformers="1" partial_charge_method="AM1-Mulliken">
  <!-- A fractional charge can be moved along a single bond -->
  <ChargeIncrement smirks="[#6X4:1]-[#6X3a:2]" charge_increment1="-0.0073*elementary_charge" charge_
↪increment2="0.0073*elementary_charge"/>
  <ChargeIncrement smirks="[#6X4:1]-[#6X3a:2]-[#7]" charge_increment1="0.0943*elementary_charge" charge_
↪increment2="-0.0943*elementary_charge"/>
  <ChargeIncrement smirks="[#6X4:1]-[#8:2]" charge_increment1="-0.0718*elementary_charge" charge_
↪increment2="0.0718*elementary_charge"/>
  <!--- Alternatively, fractional charges can be redistributed among any number of bonded atoms -->
  <ChargeIncrement smirks="[N:1]([H:2])([H:3])" charge_increment1="0.02*elementary_charge" charge_
↪increment2="-0.01*elementary_charge" charge_increment3="-0.01*elementary_charge"/>
</ChargeIncrementModel>
```

The sum of formal charges for the molecule or fragment will be used to determine the total charge the molecule or fragment will possess.

`<ChargeIncrementModel>` provides several optional attributes to control its behavior:

- The `number_of_conformers` attribute (default: `"1"`) is used to specify how many conformers will be generated for the molecule (or capped fragment) prior to charging.
- The `partial_charge_method` attribute (default: `"AM1-Mulliken"`) is used to specify how uncorrected partial charges are to be generated. Later additions will add restrained electrostatic potential fitting (RESP) capabilities.

The `<ChargeIncrement>` tags specify how the quantum chemical derived charges are to be corrected to produce the final charges. The `charge_increment#` attributes specify how much the charge on the associated tagged atom index (replacing #) should be modified. The sum of charge increments should equal zero.

Note that atoms for which library charges have already been applied are excluded from charging via `<ChargeIncrementModel>`.

Future additions will provide options for intelligently fragmenting large molecules and biopolymers, as well as a `capping` attribute to specify how fragments with dangling bonds are to be capped to allow these groups to be charged.

<ToolkitAM1BCC>: Temporary support for toolkit-based AM1-BCC partial charges

Warning: Until *<ChargeIncrementModel>* is implemented, support for the *<ToolkitAM1BCC>* tag has been enabled in the toolkit. This tag is not permanent and may be phased out in future versions of the spec.

This tag calculates partial charges using the default settings of the highest-priority cheminformatics toolkit that can perform AM1-BCC charge assignment. Currently, if the OpenEye toolkit is licensed and available, this will use QuacPac configured to generate charges using AM1-BCC ELF10 for each unique molecule in

the topology. Otherwise RDKit will be used for initial conformer generation and the AmberTools antechamber/sqm software will be used for charge calculation.

If this tag is specified for a force field, conformer generation will be performed regardless of whether conformations of the input molecule were provided. If RDKit/AmberTools are used as the toolkit backend for this calculation, only the first conformer is used for AM1-BCC calculation.

The charges generated by this tag may differ depending on which toolkits are available.

Note that atoms for which prespecified or library charges have already been applied are excluded from charging via `<ToolkitAM1BCC>`.

Prespecified charges (reference implementation only)

In our reference implementation of SMIRNOFF in the `openforcefield` toolkit, we also provide a method for specifying user-defined partial charges during system creation. This functionality is accessed by using the `charge_from_molecules` optional argument during system creation, such as in `ForceField.create_openmm_system(topology, charge_from_molecules=molecule_list)`. When this optional keyword is provided, all matching molecules will have their charges set by the entries in `molecule_list`. This method is provided solely for convenience in developing and exploring alternative charging schemes; actual force field releases for distribution will use one of the other mechanisms specified above.

1.3.9 Parameter sections

A SMIRNOFF force field consists of one or more force field term definition sections. For the most part, these sections independently define how a specific component of the potential energy function for a molecular system is supposed to be computed (such as bond stretch energies, or Lennard-Jones interactions), as well as how parameters are to be assigned for this particular term. Each parameter section contains a `version`, which encodes the behavior of the section, as well as the required and optional attributes the top-level tag and SMIRKS-based parameters. This decoupling of how parameters are assigned for each term provides a great deal of flexibility in composing new force fields while allowing a minimal number of parameters to be used to achieve accurate modeling of intramolecular forces.

Below, we describe the specification for each force field term definition using the XML representation of a SMIRNOFF force field.

As an example of a complete SMIRNOFF force field specification, see the prototype SMIRNOFF99Frosst offxml.

Note: Not all parameter sections *must* be specified in a SMIRNOFF force field. A wide variety of force field terms are provided in the specification, but a particular force field only needs to define a subset of those terms.

`<vdW>`

van der Waals force parameters, which include repulsive forces arising from Pauli exclusion and attractive forces arising from dispersion, are specified via the `<vdW>` tag with sub-tags for individual `Atom` entries, such as:

```
<vdW version="0.3" potential="Lennard-Jones-12-6" combining_rules="Lorentz-Berthelot" scale12="0.0"␣
↪scale13="0.0" scale14="0.5" scale15="1.0" switch_width="8.0*angstrom" cutoff="9.0*angstrom" long_
↪range_dispersion="isotropic">
   <Atom smirks="[#1:1]" sigma="1.4870*angstrom" epsilon="0.0157*kilocalories_per_mole"/>
   <Atom smirks="[#1:1]-[#6]" sigma="1.4870*angstrom" epsilon="0.0157*kilocalories_per_mole"/>
   ...
</vdW>
```

For standard Lennard-Jones 12-6 potentials (specified via `potential="Lennard-Jones-12-6"`), the `epsilon` parameter denotes the well depth, while the size property can be specified either via providing the `sigma` attribute, such as `sigma="1.3*angstrom"`, or via the `r_0/2` (`rmin/2`) values used in AMBER force fields (here denoted `rmin_half` as in the example above). The two are related by `r0 = 2^(1/6)*sigma` and conversion is done internally in `ForceField` into the `sigma` values used in OpenMM.

Attributes in the `<vdW>` tag specify the scaling terms applied to the energies of 1-2 (`scale12`, default: 0), 1-3 (`scale13`, default: 0), 1-4 (`scale14`, default: 0.5), and 1-5 (`scale15`, default: 1.0) interactions, as well as the distance at which a switching function is applied (`switch_width`, default: `"1.0*angstrom"`), the cutoff (`cutoff`, default: `"9.0*angstroms"`), and long-range dispersion treatment scheme (`long_range_dispersion`, default: `"isotropic"`).

The `potential` attribute (default: `"none"`) specifies the potential energy function to use. Currently, only `potential="Lennard-Jones-12-6"` is supported:

```
U(r) = 4*epsilon*((sigma/r)^12 - (sigma/r)^6)
```

The `combining_rules` attribute (default: `"none"`) currently only supports `"Lorentz-Berthelot"`, which specifies the geometric mean of `epsilon` and arithmetic mean of `sigma`. Support for other Lennard-Jones mixing schemes will be added later: `Waldman-Hagler`, `Fender-Halsey`, `Kong`, `Tang-Toennies`, `Pena`, `Hudson-McCoubrey`, `Sikora`.

Later revisions will add support for additional potential types (e.g., `Buckingham-exp-6`), as well as the ability to support arbitrary algebraic functional forms using a scheme such as

```
<vdW version="0.3" potential="4*epsilon*((sigma/r)^12-(sigma/r)^6)" scale12="0.0" scale13="0.0" scale14=
↪"0.5" scale15="1" switch_width="8.0*angstrom" cutoff="9.0*angstrom" long_range_dispersion="isotropic">
   <CombiningRules>
      <CombiningRule parameter="sigma" function="(sigma1+sigma2)/2"/>
      <CombiningRule parameter="epsilon" function="sqrt(epsilon1*epsilon2)"/>
   </CombiningRules>
   <Atom smirks="[#1:1]" sigma="1.4870*angstrom" epsilon="0.0157*kilocalories_per_mole"/>
   <Atom smirks="[#1:1]-[#6]" sigma="1.4870*angstrom" epsilon="0.0157*kilocalories_per_mole"/>
   ...
</vdW>
```

If the `<CombiningRules>` tag is provided, it overrides the `combining_rules` attribute.

Later revisions will also provide support for special interactions using the `<AtomPair>` tag:

```
<vdW version="0.3" potential="Lennard-Jones-12-6" combining_rules="Lorentz-Berthelot" scale12="0.0"␣
↪scale13="0.0" scale14="0.5" scale15="1">
   <AtomPair smirks1="[#1:1]" smirks2="[#6:2]" sigma="1.4870*angstrom" epsilon="0.0157*kilocalories_per_
↪mole"/>
   ...
</vdW>
```

vdW section tag version	Tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3	potential="Lennard-Jones-12-6", combining_rules="Lorentz-Berthelot", scale12="0", scale13="0", scale14="0.5", scale15="1.0", cutoff="9.0*angstrom", switch_width="1.0*angstrom", method="cutoff"	smirks, epsilon, (sigma OR rmin_half)	id, parent_id

<Electrostatics>

Electrostatic interactions are specified via the <Electrostatics> tag.

```
<Electrostatics version="0.3" method="PME" scale12="0.0" scale13="0.0" scale14="0.833333" scale15="1.0"/
↪>
```

The method attribute specifies the manner in which electrostatic interactions are to be computed:

- PME - particle mesh Ewald should be used (DEFAULT); can only apply to periodic systems
- reaction-field - reaction-field electrostatics should be used; can only apply to periodic systems
- Coulomb - direct Coulomb interactions (with no reaction-field attenuation) should be used

The interaction scaling parameters applied to atoms connected by a few bonds are

- scale12 (default: 0) specifies the scaling applied to 1-2 bonds
- scale13 (default: 0) specifies the scaling applied to 1-3 bonds
- scale14 (default: 0.833333) specifies the scaling applied to 1-4 bonds
- scale15 (default: 1.0) specifies the scaling applied to 1-5 bonds

Currently, no child tags are used because the charge model is specified via different means (currently library charges or BCCs).

For methods where the cutoff is not simply an implementation detail but determines the potential energy of the system (reaction-field and Coulomb), the cutoff distance must also be specified, and a switch_width if a switching function is to be used.

Electrostatics section tag version	Tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3	scale12="0", scale13="0", scale14="0.833333", scale15="1.0", cutoff="9.0*angstrom", switch_width="0*angstrom", method="PME"	N/A	N/A

<Bonds>

Bond parameters are specified via a <Bonds>...</Bonds> block, with individual <Bond> tags containing attributes specifying the equilibrium bond length (length) and force constant (k) values for specific bonds. For example:

```
<Bonds version="0.3" potential="harmonic">
   <Bond smirks="[#6X4:1]-[#6X4:2]" length="1.526*angstrom" k="620.0*kilocalories_per_mole/angstrom**2"/
↪>
   <Bond smirks="[#6X4:1]-[#1:2]" length="1.090*angstrom" k="680.0*kilocalories_per_mole/angstrom**2"/>
   ...
</Bonds>
```

Currently, only potential="harmonic" is supported, where we utilize the standard harmonic functional form:

```
U(r) = (k/2)*(r-length)^2
```

Later revisions will add support for additional potential types and the ability to support arbitrary algebraic functional forms. If the potential attribute is omitted, it defaults to harmonic.

Note that AMBER and CHARMM define a modified functional form, such that U(r) = k*(r-length)^2, so that force constants would need to be multiplied by two in order to be used in the SMIRNOFF format.

Constrained bonds are handled by a separate <Constraints> tag, which can either specify constraint distances or draw them from equilibrium distances specified in <Bonds>.

Fractional bond orders (EXPERIMENTAL)

> **Warning:** This functionality is not yet implemented and will appear in a future version of the toolkit.

Fractional bond orders can be used to allow interpolation of bond parameters. For example, these parameters:

```
<Bonds version="0.3" potential="harmonic">
    <Bond smirks="[#6X3:1]-[#6X3:2]" k="820.0*kilocalories_per_mole/angstrom**2" length="1.45*angstrom"/
↪>
    <Bond smirks="[#6X3:1]:[#6X3:2]" k="938.0*kilocalories_per_mole/angstrom**2" length="1.40*angstrom"/
↪>
    <Bond smirks="[#6X3:1]=[#6X3:2]" k="1098.0*kilocalories_per_mole/angstrom**2" length="1.35*angstrom
↪"/>
    ...
```

can be replaced by a single parameter line by first invoking the fractional_bondorder_method attribute to specify a method for computing the fractional bond order and fractional_bondorder_interpolation for specifying the procedure for interpolating parameters between specified integral bond orders:

```
<Bonds version="0.3" potential="harmonic" fractional_bondorder_method="Wiberg" fractional_bondorder_
↪interpolation="linear">
    <Bond smirks="[#6X3:1]!#[#6X3:2]" k_bondorder1="820.0*kilocalories_per_mole/angstrom**2" k_
↪bondorder2="1098*kilocalories_per_mole/angstrom**2" length_bondorder1="1.45*angstrom" length_
↪bondorder2="1.35*angstrom"/>
    ...
```

This allows specification of force constants and lengths for bond orders 1 and 2, and then interpolation between those based on the partial bond order.

- `fractional_bondorder_method` defaults to `none`, but the `Wiberg` method is supported.
- `fractional_bondorder_interpolation` defaults to `linear`, which is the only supported scheme for now.

Bonds section tag version	Tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3	`potential="harmonic"`, `fractional_bondorder_method="none"`, `fractional_bondorder_interpolation="linear"`	`smirks`, `length`, `k`	`id`, `parent_id`

<Angles>

Angle parameters are specified via an `<Angles>...</Angles>` block, with individual `<Angle>` tags containing attributes specifying the equilibrium angle (`angle`) and force constant (`k`), as in this example:

```
<Angles version="0.3" potential="harmonic">
   <Angle smirks="[a,A:1]-[#6X4:2]-[a,A:3]" angle="109.50*degree" k="100.0*kilocalories_per_mole/
↪radian**2"/>
   <Angle smirks="[#1:1]-[#6X4:2]-[#1:3]" angle="109.50*degree" k="70.0*kilocalories_per_mole/radian**2
↪"/>
   ...
</Angles>
```

Currently, only `potential="harmonic"` is supported, where we utilize the standard harmonic functional form:

```
U(r) = (k/2)*(theta-angle)^2
```

Later revisions will add support for additional potential types and the ability to support arbitrary algebraic functional forms. If the `potential` attribute is omitted, it defaults to `harmonic`.

Note that AMBER and CHARMM define a modified functional form, such that `U(r) = k*(theta-angle)^2`, so that force constants would need to be multiplied by two in order to be used in the SMIRNOFF format.

Angles section tag version	Tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3	`potential="harmonic"`	`smirks`, `angle`, `k`	`id`, `parent_id`

<ProperTorsions>

Proper torsions are specified via a `<ProperTorsions>...</ProperTorsions>` block, with individual `<Proper>` tags containing attributes specifying the periodicity (`periodicity#`), phase (`phase#`), and barrier height (`k#`).

```
<ProperTorsions version="0.3" potential="k*(1+cos(periodicity*theta-phase))">
   <Proper smirks="[a,A:1]-[#6X4:2]-[#6X4:3]-[a,A:4]" idivf1="9" periodicity1="3" phase1="0.0*degree"␣
↪k1="1.40*kilocalories_per_mole"/>
   <Proper smirks="[#6X4:1]-[#6X4:2]-[#8X2:3]-[#6X4:4]" idivf1="1" periodicity1="3" phase1="0.0*degree"␣
↪k1="0.383*kilocalories_per_mole" idivf2="1" periodicity2="2" phase2="180.0*degree" k2="0.
↪1*kilocalories_per_mole"/>
   ...
</ProperTorsions>
```

Here, child `Proper` tags specify at least `k1`, `phase1`, and `periodicity1` attributes for the corresponding parameters of the first force term applied to this torsion. However, additional values are allowed in the form

k#, phase#, and periodicity#, where all # values must be consecutive (e.g., it is impermissible to specify k1 and k3 values without a k2 value) but # can go as high as necessary.

For convenience, and optional attribute specifies a torsion multiplicity by which the barrier height should be divided (idivf#). The default behavior of this attribute can be controlled by the top-level attribute default_idivf (default: "auto") for <ProperTorsions>, which can be an integer (such as "1") controlling the value of idivf if not specified or "auto" if the barrier height should be divided by the number of torsions impinging on the central bond. For example:

```
<ProperTorsions version="0.3" potential="k*(1+cos(periodicity*theta-phase))" default_idivf="auto">
   <Proper smirks="[a,A:1]-[#6X4:2]-[#6X4:3]-[a,A:4]" periodicity1="3" phase1="0.0*degree" k1="1.
↪40*kilocalories_per_mole"/>
   ...
</ProperTorsions>
```

Currently, only potential="k*(1+cos(periodicity*theta-phase))" is supported, where we utilize the functional form:

```
U = \sum_{i=1}^N k_i * (1 + cos(periodicity_i * phi - phase_i))
```

Note: **AMBER defines a modified functional form**, such that *U = sum_{i=1}^N (k_i/2) * (1 + cos(periodicity_i * phi - phase_i))*, so that barrier heights would need to be divided by two in order to be used in the SMIRNOFF format.

If the potential attribute is omitted, it defaults to k*(1+cos(periodicity*theta-phase)).

Fractional torsion bond orders

Fractional torsion bond orders can be used to allow interpolation and extrapolation of torsion parameters. This is similar to the functionality provided by fractional bond orders detailed above. For example, these parameters:

```
<ProperTorsions version="0.3" potential="k*(1+cos(periodicity*theta-phase))" default_idivf="auto">
    <Proper smirks="[*:1]:[#6X4:2]-[#6X4:3]:[*:4]" periodicity1="2" phase1="0.0 * degree" k1="1.
↪00*kilocalories_per_mole" idivf1="1.0"/>
    <Proper smirks="[*:1]:[#6X4:2]=[#6X4:3]:[*:4]" periodicity1="2" phase1="0.0 * degree" k1="1.
↪80*kilocalories_per_mole" idivf1="1.0"/>
    ...
```

can be replaced by a single parameter line by first defining the fractional_bondorder_method header-level attribute to specify a method for computing the fractional bond order and fractional_bondorder_interpolation for specifying the procedure for interpolating parameters between specified integer bond orders:

```
<ProperTorsions version="0.3" potential="k*(1+cos(periodicity*theta-phase))" default_idivf="auto"␣
↪fractional_bondorder_method="AM1-Wiberg" fractional_bondorder_interpolation="linear">
    <Proper smirks="[*:1]:[#6X4:2]~[#6X4:3]:[*:4]" periodicity1="2" phase1="0.0 * degree" k1_bondorder1=
↪"1.00*kilocalories_per_mole" k1_bondorder2="1.80*kilocalories_per_mole" idivf1="1.0"/>
    ...
```

This allows specification of the barrier height for e.g. bond orders 1 and 2 (single and double bonds), and then interpolation between those based on the partial/fractional bond order. Note that in actual usage partial/fractional bond order may never be exactly 1 or 2, or perhaps even near 2; these values only serve to define the slope of the line used for interpolation. In the example above, we replaced the two proper torsion

terms (one single central bond (-) and one double central bond (=) with a single term giving the barrier heights for bond order 1 and 2. If there are cases where the fractional bond order is 1.5, this can correspond to e.g. an aromatic bond. When barrier heights for more than two integer bond orders are specified, (say, 1, 2, and 3), the interpolation lines are drawn between successive points as a piecewise linear function.

Cases in which the fractional bond order for the central bond is outside of the bond orders specified (e.g. 1 and 2 above), the barrier height `k#` is *extrapolated* using the same slope of the line used for interpolation. This works even when barrier heights for more than two integer bond orders are specified (say, 1, 2, and 3), in which case the piecewise linear extrapolation beyond the bounds uses the slope of the line defined by the nearest two bond orders. In other words, a fractional bond order of 3.2 would yield an interpolated `k#` value determined by the interpolation line between `k#_bondorder2` and `k#_bondorder3`. A fractional bond order of .9 would yield an interpolated `k#` value determined by the interpolation line between `k#_bondorder1` and `k#_bondorder2`.

Some key usage points:

- `fractional_bondorder_method` defaults to `AM1-Wiberg`.
- `fractional_bondorder_interpolation` defaults to `linear`, which is the only supported scheme for now.

<ImproperTorsions>

Improper torsions are specified via an `<ImproperTorsions>...</ImproperTorsions>` block, with individual `<Improper>` tags containing attributes that specify the same properties as `<ProperTorsions>`:

```
<ImproperTorsions version="0.3" potential="k*(1+cos(periodicity*theta-phase))">
    <Improper smirks="[*:1]~[#6X3:2](=[#7X2,#7X3+1:3])~[#7:4]" k1="10.5*kilocalories_per_mole"␣
↪periodicity1="2" phase1="180.*degree"/>
    ...
</ImproperTorsions>
```

Currently, only `potential="charmm"` is supported, where we utilize the functional form:

```
U = \sum_{i=1}^N k_i * (1 + cos(periodicity_i * phi - phase_i))
```

Note: **AMBER defines a modified functional form**, such that *U = sum_{i=1}^N (k_i/2) * (1 + cos(periodicity_i * phi - phase_i))*, so that barrier heights would need to be divided by two in order to be used in the SMIRNOFF format.

If the `potential` attribute is omitted, it defaults to `charmm`.

The improper torsion energy is computed as the average over all three impropers (all with the same handedness) in a trefoil. This avoids the dependence on arbitrary atom orderings that occur in more traditional typing engines such as those used in AMBER. The *second* atom in an improper (in the example above, the trivalent carbon) is the central atom in the trefoil.

ImproperTorsions section tag version	Tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3	`potential="k*(1+cos(periodicity*theta-phase))", default_idivf="auto"`	`smirks, k, phase, periodicity`	`idivf, id, parent_id`

<GBSA>

> **Warning:** The current release of ParmEd can not transfer GBSA models produced by the Open Force Field Toolkit to other simulation packages. These GBSA forces are currently only computable using OpenMM.

Generalized-Born surface area (GBSA) implicit solvent parameters are optionally specified via a `<GBSA>...</GBSA>` using `<Atom>` tags with GBSA model specific attributes:

```
<GBSA version="0.3" gb_model="OBC1" solvent_dielectric="78.5" solute_dielectric="1" sa_model="ACE"␣
↪surface_area_penalty="5.4*calories/mole/angstroms**2" solvent_radius="1.4*angstroms">
  <Atom smirks="[*:1]" radius="0.15*nanometer" scale="0.8"/>
  <Atom smirks="[#1:1]" radius="0.12*nanometer" scale="0.85"/>
  <Atom smirks="[#1:1]~[#7]" radius="0.13*nanometer" scale="0.85"/>
  <Atom smirks="[#6:1]" radius="0.17*nanometer" scale="0.72"/>
  <Atom smirks="[#7:1]" radius="0.155*nanometer" scale="0.79"/>
  <Atom smirks="[#8:1]" radius="0.15*nanometer" scale="0.85"/>
  <Atom smirks="[#9:1]" radius="0.15*nanometer" scale="0.88"/>
  <Atom smirks="[#14:1]" radius="0.21*nanometer" scale="0.8"/>
  <Atom smirks="[#15:1]" radius="0.185*nanometer" scale="0.86"/>
  <Atom smirks="[#16:1]" radius="0.18*nanometer" scale="0.96"/>
  <Atom smirks="[#17:1]" radius="0.17*nanometer" scale="0.8"/>
</GBSA>
```

Supported Generalized Born (GB) models

In the `<GBSA>` tag, `gb_model` selects which GB model is used. Currently, this can be selected from a subset of the GBSA models available in OpenMM:

- `HCT` : Hawkins-Cramer-Truhlar (corresponding to `igb=1` in AMBER): requires parameters `[radius, scale]`
- `OBC1` : Onufriev-Bashford-Case using the GB(OBC)I parameters (corresponding to `igb=2` in AMBER): requires parameters `[radius, scale]`
- `OBC2` : Onufriev-Bashford-Case using the GB(OBC)II parameters (corresponding to `igb=5` in AMBER): requires parameters `[radius, scale]`

If the `gb_model` attribute is omitted, it defaults to `OBC1`.

The attributes `solvent_dielectric` and `solute_dielectric` specify solvent and solute dielectric constants used by the GB model. In this example, `radius` and `scale` are per-particle parameters of the `OBC1` GB model supported by OpenMM.

Surface area (SA) penalty model

The `sa_model` attribute specifies the solvent-accessible surface area model ("SA" part of GBSA) if one should be included; if omitted, no SA term is included.

Currently, only the analytical continuum electrostatics (ACE) model, designated `ACE`, can be specified, but there are plans to add more models in the future, such as the Gaussian solvation energy component of EEF1. If `sa_model` is not specified, it defaults to `ACE`.

The `ACE` model permits two additional parameters to be specified:

- The `surface_area_penalty` attribute specifies the surface area penalty for the `ACE` model. (Default: `5.4 calories/mole/angstroms**2`)
- The `solvent_radius` attribute specifies the solvent radius. (Default: `1.4 angstroms`)

GBSA section tag version	Tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3	`gb_model="OBC1", solvent_dielectric="78.5", solute_dielectric="1", sa_model="ACE", surface_area_penalty="5.4*calories/mole/angstrom**2", solvent_radius="1.4*angstrom"`	`smirks, radius, scale`	`id, parent_id`

<Constraints>

Bond length or angle constraints can be specified through a `<Constraints>` block, which can constrain bonds to their equilibrium lengths or specify an interatomic constraint distance. Two atoms must be tagged in the `smirks` attribute of each `<Constraint>` record.

To constrain the separation between two atoms to their equilibrium bond length, it is critical that a `<Bonds>` record be specified for those atoms:

```
<Constraints version="0.3" >
  <!-- constrain all bonds to hydrogen to their equilibrium bond length -->
  <Constraint smirks="[#1:1]-[*:2]" />
</Constraints>
```

Note that the two atoms must be bonded in the specified `Topology` for the equilibrium bond length to be used.

To specify the constraint distance, or constrain two atoms that are not directly bonded (such as the hydrogens in rigid water models), specify the `distance` attribute (and optional `distance_unit` attribute for the `<Constraints>` tag):

```
<Constraints version="0.3">
  <!-- constrain water O-H bond to equilibrium bond length (overrides earlier constraint) -->
  <Constraint smirks="[#1:1]-[#8X2H2:2]-[#1]" distance="0.9572*angstrom"/>
  <!-- constrain water H...H, calculating equilibrium length from H-O-H equilibrium angle and H-O␣
↪equilibrium bond lengths -->
  <Constraint smirks="[#1:1]-[#8X2H2]-[#1:2]" distance="1.8532*angstrom"/>
</Constraints>
```

Typical molecular simulation practice is to constrain all bonds to hydrogen to their equilibrium bond lengths and enforce rigid TIP3P geometry on water molecules:

```
<Constraints version="0.3">
  <!-- constrain all bonds to hydrogen to their equilibrium bond length -->
  <Constraint smirks="[#1:1]-[*:2]" />
  <!-- TIP3P rigid water -->
  <Constraint smirks="[#1:1]-[#8X2H2:2]-[#1]" distance="0.9572*angstrom"/>
  <Constraint smirks="[#1:1]-[#8X2H2]-[#1:2]" distance="1.8532*angstrom"/>
</Constraints>
```

Constraint section tag version	Required tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3		`smirks`	`distance`

1.3.10 Advanced features

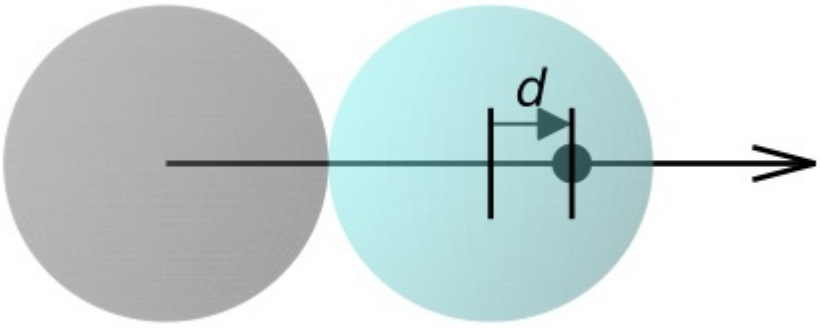
Standard usage is expected to rely primarily on the features documented above and potentially new features. However, some advanced features will also be supported.

`<VirtualSites>`: Virtual sites for off-atom charges

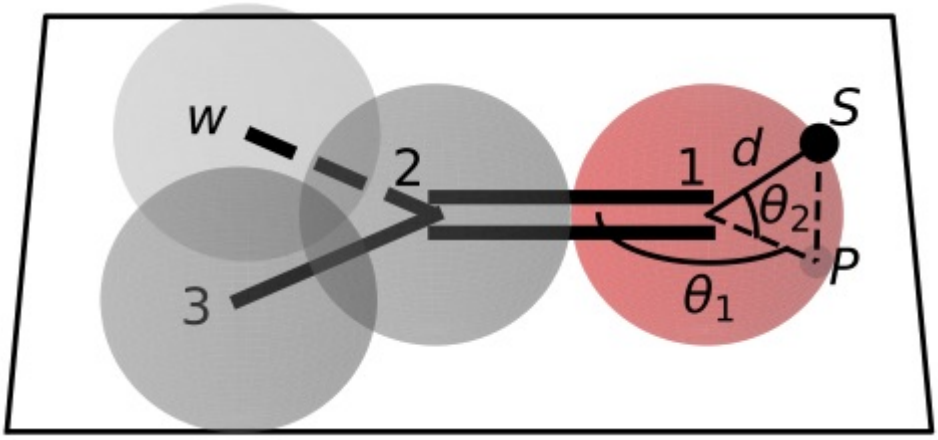
Warning: This functionality is not yet implemented and will appear in a future version of the toolkit

We will implement experimental support for placement of off-atom (off-center) charges in a variety of contexts which may be chemically important in order to allow easy exploration of when these will be warranted. We will support the following different types or geometries of off-center charges (as diagrammed below):

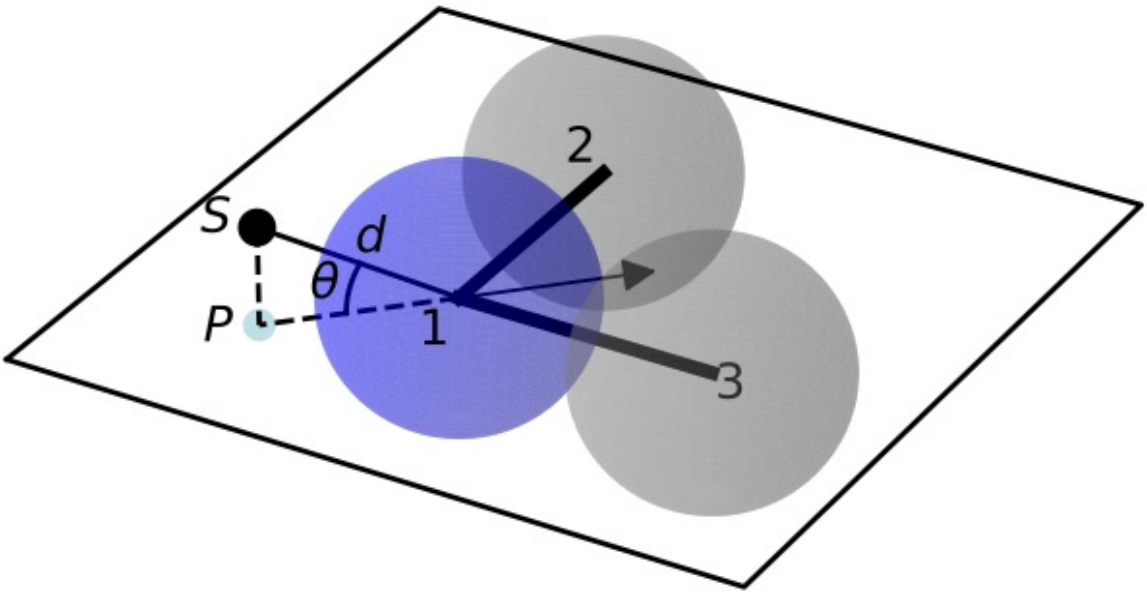
- `BondCharge`: This supports placement of a virtual site `S` along a vector between two specified atoms, e.g. to allow for a sigma hole for halogens or similar contexts. With positive values of the distance, the virtual site lies outside the first indexed atom (green in this image).



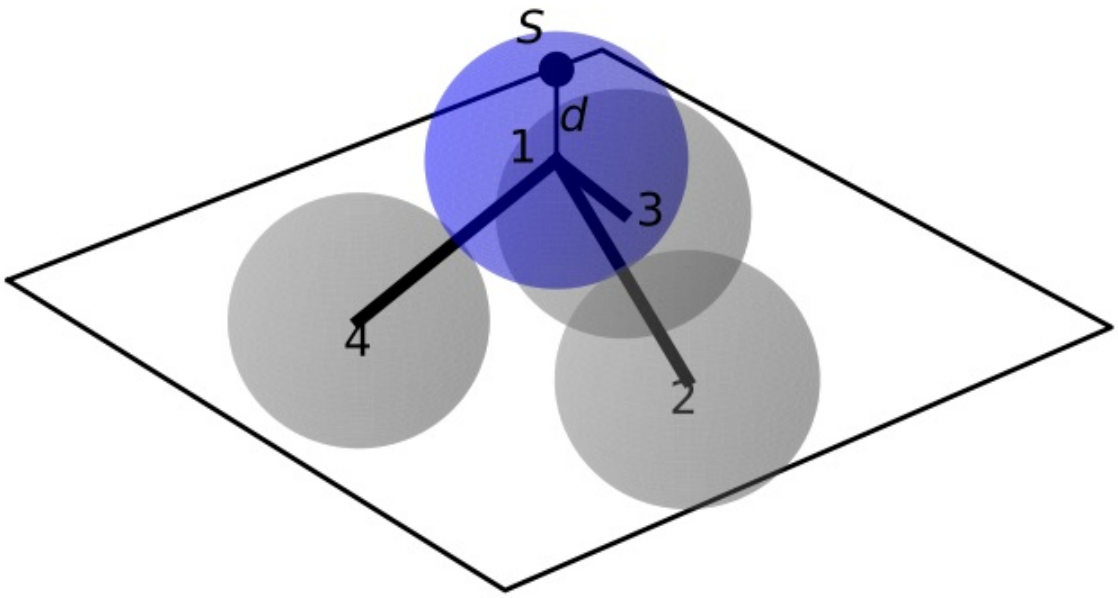
- `MonovalentLonePair`: This is originally intended for situations like a carbonyl, and allows placement of a virtual site `S` at a specified distance `d`, `inPlaneAngle` (theta 1 in the diagram), and `outOfPlaneAngle` (theta 2 in the diagram) relative to a central atom and two connected atoms.



- `DivalentLonePair`: This is suitable for cases like four-point and five-point water models as well as pyrimidine; a charge site `S` lies a specified distance `d` from the central atom among three atoms (blue) along the bisector of the angle between the atoms (if `outOfPlaneAngle` is zero) or out of the plane by the specified angle (if `outOfPlaneAngle` is nonzero) with its projection along the bisector. For positive values fo the distance `d` the virtual site lies outside the 2-1-3 angle and for negative values it lies inside.



- `TrivalentLonePair`: This is suitable for planar or tetrahedral nitrogen lone pairs; a charge site `S` lies above the central atom (e.g. nitrogen, blue) a distance `d` along the vector perpendicular to the plane of the three connected atoms (2,3,4). With positive values of `d` the site lies above the nitrogen and with negative values it lies below the nitrogen.



Each virtual site receives charge which is transferred from the desired atoms specified in the SMIRKS pattern via a `charge_increment#` parameter, e.g., if `charge_increment1=+0.1*elementary_charge` then the virtual site will receive a charge of -0.1 and the atom labeled `1` will have its charge adjusted upwards by +0.1. N may index any indexed atom. Increments which are left unspecified default to zero. Additionally, each virtual site can bear Lennard-Jones parameters, specified by `sigma` and `epsilon` or `rmin_half` and `epsilon`. If unspecified these also default to zero.

In the SMIRNOFF format, these are encoded as:

```
<VirtualSites version="0.3">
    <!-- sigma hole for halogens: "distance" denotes distance along the 2->1 bond vector, measured from␣
↪atom 2 -->
    <!-- Specify that 0.2 charge from atom 1 and 0.1 charge units from atom 2 are to be moved to the␣
↪virtual site, and a small Lennard-Jones site is to be added (sigma = 0.1*angstroms, epsilon=0.05*kcal/
↪mol) -->
    <VirtualSite type="BondCharge" smirks="[Cl:1]-[C:2]" distance="0.30*angstrom" charge_increment1="+0.
↪2*elementary_charge" charge_increment2="+0.1*elementary_charge" sigma="0.1*angstrom" epsilon="0.
↪05*kilocalories_per_mole" />
    <!-- Charge increments can extend out to as many atoms as are labeled, e.g. with a third atom: -->
    <VirtualSite type="BondCharge" smirks="[Cl:1]-[C:2]~[*:3]" distance="0.30*angstrom" charge_
↪increment1="+0.1*elementary_charge" charge_increment2="+0.1*elementary_charge" charge_increment3="+0.
↪05*elementary_charge" sigma="0.1*angstrom" epsilon="0.05*kilocalories_per_mole" />
    <!-- monovalent lone pairs: carbonyl -->
    <!-- X denotes the charge site, and P denotes the projection of the charge site into the plane of 1␣
↪and 2. -->
    <!-- inPlaneAngle is angle point P makes with 1 and 2, i.e. P-1-2 -->
    <!-- outOfPlaneAngle is angle charge site (X) makes out of the plane of 2-1-3 (and P) measured from␣
↪1 -->
    <!-- Since unspecified here, sigma and epsilon for the virtual site default to zero -->
    <VirtualSite type="MonovalentLonePair" smirks="[O:1]=[C:2]-[*:3]" distance="0.30*angstrom"␣
↪outOfPlaneAngle="0*degree" inPlaneAngle="120*degree" charge_increment1="+0.2*elementary_charge" />
    <!-- divalent lone pair: pyrimidine, TIP4P, TIP5P -->
    <!-- The atoms 2-1-3 define the X-Y plane, with Z perpendicular. If outOfPlaneAngle is 0, the␣
↪charge site is a specified distance along the in-plane vector which bisects the angle left by taking␣
↪360 degrees minus angle(2,1,3). If outOfPlaneAngle is nonzero, the charge sites lie out of the plane␣
↪by the specified angle (at the specified distance) and their in-plane projection lines along the angle
↪'s bisector. -->
```

(continues on next page)

(continued from previous page)

```
    <VirtualSite type="DivalentLonePair" smirks="[*:2]~[#7X2:1]~[*:3]" distance="0.30*angstrom"␣
↪outOfPlaneAngle="0.0*degree" charge_increment1="+0.1*elementary_charge" />
    <!-- trivalent nitrogen lone pair -->
    <!-- charge sites lie above and below the nitrogen at specified distances from the nitrogen, along␣
↪the vector perpendicular to the plane of (2,3,4) that passes through the nitrogen. If the nitrogen is␣
↪co-planar with the connected atom, charge sites are simply above and below the plane-->
    <!-- Positive and negative values refer to above or below the nitrogen as measured relative to the␣
↪plane of (2,3,4), i.e. below the nitrogen means nearer the 2,3,4 plane unless they are co-planar -->
    <VirtualSite type="TrivalentLonePair" smirks="[*:2]~[#7X3:1](~[*:4])~[*:3]" distance="0.30*angstrom
↪" charge_increment1="+0.1*elementary_charge"/>
    <VirtualSite type="TrivalentLonePair" smirks="[*:2]~[#7X3:1](~[*:4])~[*:3]" distance="-0.30*angstrom
↪" charge_increment1="+0.1*elementary_charge"/>
</VirtualSites>
```

Aromaticity models

Before conducting SMIRKS substructure searches, molecules are prepared using one of the supported aromaticity models, which must be specified with the `aromaticity_model` attribute. The only aromaticity model currently widely supported (by both the OpenEye toolkit and RDKit) is the `OEAroModel_MDL` model.

Additional plans for future development

See the openforcefield GitHub issue tracker to propose changes to this specification, or read through proposed changes currently being discussed.

1.3.11 The `openforcefield` reference implementation

A Python reference implementation of a parameterization engine implementing the SMIRNOFF force field specification can be found online. This implementation can use either the free-for-academics (but commercially supported) OpenEye toolkit or the free and open source RDKit cheminformatics toolkit. See the installation instructions for information on how to install this implementation and its dependencies.

Examples

A relatively extensive set of examples is made available on the reference implementation repository under examples/.

Parameterizing a system

Consider parameterizing a simple system containing a the drug imatinib.

```
# Create a molecule from a mol2 file
from openforcefield.topology import Molecule
molecule = Molecule.from_file('imatinib.mol2')

# Create a Topology specifying the system to be parameterized containing just the molecule
topology = molecule.to_topology()

# Load the smirnoff99Frosst forcefield
from openforcefield.typing.engines import smirnoff
```

(continues on next page)

(continued from previous page)

```
forcefield = smirnoff.ForceField('test_forcefields/smirnoff99Frosst.offxml')

# Create an OpenMM System from the topology
system = forcefield.create_openmm_system(topology)
```

See `examples/SMIRNOFF_simulation/` for an extension of this example illustrating to simulate this molecule in the gas phase.

The `topology` object provided to `create_openmm_system()` can contain any number of molecules of different types, including biopolymers, ions, buffer molecules, or solvent molecules. The openforcefield toolkit provides a number of convenient methods for importing or constructing topologies given PDB files, Sybyl mol2 files, SDF files, SMILES strings, and IUPAC names; see the toolkit documentation for more information. Notably, this `topology` object differs from those found in OpenMM or MDTraj in that it contains information on the *chemical identity* of the molecules constituting the system, rather than this atomic elements and covalent connectivity; this additional chemical information is required for the direct chemical perception features of SMIRNOFF typing.

Using SMIRNOFF small molecule forcefields with traditional biopolymer force fields

While SMIRNOFF format force fields can cover a wide range of biological systems, our initial focus is on gneral small molecule force fields, meaning that users may have considerable interest in combining SMIRNOFF small molecule parameters to systems in combination with traditional biopolymer parameters from conventional force fields, such as the AMBER family of protein/nucleic acid force fields. Thus, we provide an example of setting up a mixed protein-ligand system in examples/mixedFF_structure, where an AMBER family force field is used for a protein and smirnoff99Frosst for a small molecule.

The optional `id` and `parent_id` attributes and other XML attributes

In general, additional optional XML attributes can be specified and will be ignored by `ForceField` unless they are specifically handled by the parser (and specified in this document).

One attribute we have found helpful in parameter file development is the `id` attribute for a specific parameter line, and we *recommend* that SMIRNOFF force fields utilize this as effectively a parameter serial number, such as in:

```
<Bond smirks="[#6X3:1]-[#6X3:2]" id="b5" k="820.0*kilocalorie_per_mole/angstrom**2" length="1.
↪45*angstrom"/>
```

Some functionality in `ForceField`, such as `ForceField.label_molecules`, looks for the `id` attribute. Without this attribute, there is no way to uniquely identify a specific parameter line in the XML file without referring to it by its smirks string, and since some smirks strings can become long and relatively unwieldy (especially for torsions) this provides a more human- and search-friendly way of referring to specific sets of parameters.

The `parent_id` attribute is also frequently used to denote parameters from which the current parameter is derived in some manner.

A remark about parameter availability

`ForceField` will currently raise an exception if any parameters are missing where expected for your system—i.e. if a bond is assigned no parameters, an exception will be raised. However, use of generic parameters (i.e. `[*:1]~[*:2]` for a bond) in your `.offxml` will result in parameters being assigned everywhere, bypassing this exception. We recommend generics be used sparingly unless it is your intention to provide true universal generic parameters.

1.3.12 Version history

0.3

This is a backwards-incompatible update to the SMIRNOFF 0.2 draft specification. However, the Open Force Field Toolkit version accompanying this update is capable of converting 0.1 spec SMIRNOFF data to 0.2 spec, and subsequently 0.2 spec to 0.3 spec. The 0.1-to-0.2 spec conversion makes a number of assumptions about settings such as long-range nonbonded handling. Warnings are printed about each assumption that is made during this spec conversion. No mechanism to convert backwards in spec is provided.

Key changes in this version of the spec are:

- Section headers now contain individual versions, instead of relying on the `<SMIRNOFF>`-level tag.
- Section headers no longer contain `X_unit` attributes.
- All physical quantities are now written as expressions containing the appropriate units.
- The default potential for `<ProperTorsions>` and `<ImproperTorsions>` was changed from `charmm` to `k*(1+cos(periodicity*theta-phase))`, as CHARMM interprets torsion terms with perioidicity 0 as having a quadratic potential, while the Open Force Field Toolkit would interpret a zero periodicity literally.

0.2

This is a backwards-incompatible overhaul of the SMIRNOFF 0.1 draft specification along with `ForceField` implementation refactor:

- Aromaticity model now defaults to `OEAroModel_MDL`, and aromaticity model names drop OpenEye-specific prefixes
- Top-level tags are now required to specify units for any unit-bearing quantities to avoid the potential for mistakes from implied units.
- Potential energy component definitions were renamed to be more general:
  - `<NonbondedForce>` was renamed to `<vdW>`
  - `<HarmonicBondForce>` was renamed to `<Bonds>`
  - `<HarmonicAngleForce>` was renamed to `<Angles>`
  - `<BondChargeCorrections>` was renamed to `<ChargeIncrementModel>` and generalized to accommodate an arbitrary number of tagged atoms
  - `<GBSAForce>` was renamed to `<GBSA>`
- `<PeriodicTorsionForce>` was split into `<ProperTorsions>` and `<ImproperTorsions>`
- `<vdW>` now specifies 1-2, 1-3, 1-4, and 1-5 scaling factors via `scale12` (default: 0), `scale13` (default: 0), `scale14` (default: 0.5), and `scale15` (default 1.0) attributes. It also specifies the long-range vdW

method to use, currently supporting `cutoff` (default) and `PME`. Coulomb scaling parameters have been removed from `StericsForce`.

- Added the `<Electrostatics>` tag to separately specify 1-2, 1-3, 1-4, and 1-5 scaling factors for electrostatics, as well as the method used to compute electrostatics (`PME`, `reaction-field`, `Coulomb`) since this has a huge effect on the energetics of the system.
- Made it clear that `<Constraint>` entries do not have to be between bonded atoms.
- `<VirtualSites>` has been added, and the specification of charge increments harmonized with `<ChargeIncrementModel>`
- The `potential` attribute was added to most forces to allow flexibility in extending forces to additional functional forms (or algebraic expressions) in the future. `potential` defaults to the current recommended scheme if omitted.
- `<GBSA>` now has defaults specified for `gb_method` and `sa_method`
- Changes to how fractional bond orders are handled:
  - Use of fractional bond order is now are specified at the force tag level, rather than the root level
  - The fractional bond order method is specified via the `fractional_bondorder_method` attribute
  - The fractional bond order interpolation scheme is specified via the `fractional_bondorder_interpolation`
- Section heading names were cleaned up.
- Example was updated to reflect use of the new `openforcefield.topology.Topology` class
- Eliminated "Requirements" section, since it specified requirements for the software, rather than described an aspect of the SMIRNOFF specification
- Fractional bond orders are described in `<Bonds>`, since they currently only apply to this term.

0.1

Initial draft specification.

1.4 Examples using SMIRNOFF with the toolkit

The following examples are available in the openforcefield toolkit repository:

1.4.1 Index of provided examples

- conformer_energies - compute conformer energies of one or more small molecules using a SMIRNOFF force field
- SMIRNOFF_simulation - simulation of a molecule in the gas phase with the SMIRNOFF force field format
- forcefield_modification - modify forcefield parameters and evaluate how system energy changes
- using_smirnoff_in_amber_or_gromacs - convert a System generated with the Open Force Field Toolkit, which can be simulated natively with OpenMM, into AMBER prmtop/inpcrd and GROMACS top/gro input files through the ParmEd library.

- swap_amber_parameters - take a prepared AMBER protein-ligand system (prmtop and crd) along with a structure file of the ligand, and replace ligand parameters with OpenFF parameters.
- inspect_assigned_parameters - check which parameters are used in which molecules and generate parameter usage statistics.
- using_smirnoff_with_amber_protein_forcefield - use SMIRNOFF parameters for small molecules in combination with more conventional force fields for proteins and other components of your system (using ParmEd to combine parameterized structures)
- check_dataset_parameter_coverage - shows how to use the Open Force Field Toolkit to ingest a dataset of molecules, and generate a report summarizing any chemistry that can not be parameterized.
- visualization - shows how rich representation of `Molecule` objects work in the context of Jupyter Notebooks.

1.5 Developing for the toolkit

1.5.1 Overview

Introduction

This guide is written with the understanding that our contributors are NOT professional software developers, but are instead computational chemistry trainees and professionals. With this in mind, we aim to use a minimum of bleeding-edge technology and alphabet soup, and we will define any potentially unfamiliar processes or technologies the first time they are mentioned. We enforce use of certain practices (tests, formatting, coverage analysis, documentation) primarily because they are worthwhile upfront investments in the long-term sustainability of this project. The resources allocated to this project will come and go, but we hope that following these practices will ensure that minimal developer time will maintain this software far into the future.

The process of contributing to the OFF toolkit is more than just writing code. Before contributing, it is a very good idea to start a discussion on the Issue tracker about the functionality you'd like to add. This Issue will help us identify where in the codebase it should go, any overlapping efforts with other developers, and what the user experience should be. Please note that the OFF toolkit is intended to be used primarily as one piece of larger workflows, and that simplicity and reliability are two of our primary goals. Often, the cost/benefit of new features must be discussed, as a complex codebase is harder to maintain. When new functionality is added to the OFF Toolkit, it becomes our responsibility to maintain it, so it's important that we understand contributed code and are in a position to keep it up to date.

Philosophy

- The *core functionality* of the OFF Toolkit is to combine an Open Force Field `ForceField` and `Topology` to create an OpenMM `System`.
- An OpenMM `System` contains *everything* needed to compute the potential energy of a system, except the coordinates.
- The OFF toolkit employs a modular "plugin" architecture wherever possible, providing a standard interface for contributed features.

Terminology

Open Force Field Toolkit Concepts

OFF Molecule A graph representation of a molecule containing enough information to unambiguously parametrize it. Required data fields for an `OFF Molecule` are:

- `atoms`: element (integer), formal_charge (integer), is_aromatic (boolean), stereochemistry (R/S/None)
- `bonds`: order (integer), is_aromatic (boolean), stereochemistry (E/Z/None)

There are several other optional attributes such as `conformers` and `partial_charges` that may be populated in the `Molecule` data structure. These are considered "optional" because they are not required for system creation, however if those fields are populated, the user MAY use them to override values that would otherwise be generated during system creation.

A dictionary, `Molecule.properties` is exposed, which is a Python dict that can be populated with arbitrary data. This data should be considered cosmetic and should not affect system creation. Whenever possible, molecule serialization or format conversion should preserve this data.

OFF System An object that contains everything needed to calculate a molecular system's energy, except the atomic coordinates. Note that this does not exist yet, and that OpenMM `System` objects are being used for this purpose right now.

OFF Topology An object that efficiently holds many OFF `Molecule` objects. The atom indexing in a `Topology` may differ from those of the underlying ``Molecule``s

OFF TopologyMolecule The efficient data structures that make up an OFF Topology. There is one TopologyMolecule for each instance of a chemical species in a Topology. However, each unique chemical species has a single OFF Molecule representing it, which may be shared by multiple TopologyMolecules. TopologyMolecules contain an atom index map, as several copies of the same chemical species in a Topology may be present with different atom orderings. This data structure allows the OFF toolkit to only parametrize each unique Molecule once, and then write a copy of the assigned parameters out for each of the Molecule in the Topology (accounting for atom indexing differences in the process).

OFF ForceField An object generated from an OFFXML file (or other source of SMIRNOFF data). Most information from the SMIRNOFF data source is stored in this object's several `ParameterHandler``s, however some top-level SMIRNOFF data is stored in the ``ForceField` object itself.

SMIRNOFF data A hierarchical data structure that complies with the SMIRNOFF specification. This can be serialized in many formats, including XML (OFFXML). The subsections in a SMIRNOFF data source generally correspond to one energy term in the functional form of a force field.

ParameterHandler An object that has the ability to produce one component of an OpenMM `System`, corresponding to one subsection in a SMIRNOFF data source. Most `ParameterHandler` objects contain a list of `ParameterType` objects.

ParameterType An object corresponding to a single SMARTS-based parameter.

Cosmetic attribute Data in a SMIRNOFF data source that does not correspond to a known attribute. These have no functional effect, but several programs use the extensibility of the OFFXML format to define additional attributes for their own use, and their workflows require the OFF toolkit to process the files while retaining these keywords.

Development Infrastructure

CI "Continuous integration" testing.

Services that run frequently while the code is undergoing changes, ensuring that the codebase still installs and has the intended behavior. Currently, we use a service called Travis CI for this. Every time

we make commits to the `master` branch of the openforcefield Github repository, a set of virtual machines that mimic brand new Linux and Mac OSX computers are created, and follow build instructions specified in the repo's `.travis.yml` file to install the toolkit. After installing the OFF toolkit and its dependencies, these virtual machines run our test suite. If the tests all pass, the build "passes" (returns a green check mark on GitHub). If all the tests for a specific change to the `master` branch return green, then we know that the change has not broken the toolkit's existing functionality. When proposing code changes, we ask that contributors open a Pull Request (PR) on GitHub to merge their changes into the `master` branch. When a pull request is open, CI will run on the latest set of proposed changes and indicate whether they are safe to merge through status checks, summarized as a green check mark or red X.

CodeCov Code coverage.

An extension to our testing framework that reports the fraction of our source code lines that were run during the tests. This functionality is actually the combination of several components – Travis CI runs the tests using the `pytest-cov` package, and then uploads the results to the website codecov.io. This analysis is re-run with each change to the `master` branch, and a badge showing our coverage percentage is in the project README.

LGTM "Looks Good To Me".

A service that analyzes the code in our repository for simple style and formatting issues. This service assigns a letter grade to codebases, and a badge showing our LGTM report is in the project README.

RTD ReadTheDocs.

A service that compiles and renders the packages documentation (from the `docs/` folder). The documentation itself can be accessed from the ReadTheDocs badge in the README.

Modular design features

There are a few areas where we've designed the toolkit with extensibility in mind. Adding functionality at these interfaces should be considerably easier than in other parts of the toolkit, and we encourage experimentation and contribution on these fronts.

ParameterHandler A generic base class for objects that perform parametrization for one section in a SMIRNOFF data source.

Each ParameterHandler-derived class MUST implement:

- `create_force(self, system, topology, **kwargs)`: takes an OpenMM `System` and a OpenFF `Topology` as input, as well as optional keyword arguments, and modifies the `System` to contain the appropriate parameters.
- Class-level `ParameterAttributes` and `IndexedParameterAttributes`: These correspond to the header-level attributes in a SMIRNOFF data source. For example,, the `Bonds` tag in the SMIRNOFF spec has an optional `fractional_bondorder_method` field, which corresponds to the line `fractional_bondorder_method = ParameterAttribute(default=None)` in the `BondHandler` class definition. The `ParameterAttribute` and `IndexedParameterAttribute` classes offer considerable flexibility for validating inputs. Defining these attributes at the class level implements the corresponding behavior in the default `__init__` function.
- Class-level definitions `_MAX_SUPPORTED_SECTION_VERSION` and `_MAX_SUPPORTED_SECTION_VERSION`. ParameterHandler versions allow us to evolve ParameterHandler behavior in a controlled, recorded way. Force field development is experimental by nature, and it is unlikely that the initial choice of header attributes is suitable for all use cases. Recording the "versions" of a SMIRNOFF spec tag allows us to encode the default behavior and API of a specific generation of ParameterHandlers, while allowing the safe addition of new attributes and behaviors.

- **Each ParameterHandler-derived class MAY implement:**
  - `known_kwargs`: Keyword arguments passed to `ForceField.create_openmm_system` are validated against the `known_kwargs` lists of each ParameterHandler that the ForceField owns. If present, these kwargs and their values will be passed on to the ParameterHandler.
  - `to_dict`: converts the ParameterHandler to a hierarchical dict compliant with the SMIRNOFF specification. The default implementation of this function should suffice for most developers.
  - `check_handler_compatibility`: Checks whether this ParameterHandler is "compatible" with another. This function is used when a ForceField is attempted to be constructed from *multiple* SMIRNOFF data sources, and it is necessary to check that two sections with the same tagname can be combined in a sane way. For example, if the user instructed two `vdW` sections to be read, but the sections defined different vdW potentials, then this function should raise an Exception indicating that there is no safe way to combine the parameters. The default implementation of this function should suffice for most developers.
  - `postprocess_system`: operates identically to `create_force`, but is run after each ParameterHandlers' `create_force` has already been called. The default implementation of this method simply does nothing, and should suffice for most developers.

User Experience

One important aspect of how we make design decisions is by asking "who do we envision using this software, and what would they want it to do here?". There is a wide range of possible users, from non-chemists, to students/trainees, to expert computational medicinal chemists. We have decided to build functionality intended for use by *expert medicinal chemists*, and whenever possible, add fatal errors if the toolkit risks doing the wrong thing. So, for example, if a molecule is loaded with an odd ionization state, we assume that the user has input it this way intentionally. This design philosophy invariably has tradeoffs – For example, the OFF Toolkit will give the user a hard time if they try to load a "dirty" molecule dataset, where some molecules have errors or are not described in enough detail for the toolkit to unambiguously parametrize them. If there is risk of misinterpreting the molecule (for example, bond orders being undefined or chiral centers without defined stereochemistry), the toolkit should raise an error that the user can override. In this regard we differ from RDKit, which is more permissive in the level of detail it requires when creating molecules. This makes sense for RDKit's use cases, as several of its analyses can operate with a lower level of detail about the molecules. Often, the same design decision is the best for all types of users, and there is no need for discussion. But when we do need to make tradeoffs, "assume the user is an expert" is our guiding principle.

At the same time, we aim for "automagic" behavior whenever a decision will clearly go one way over another. System parametrization is an inherently complex topic, and the OFF toolkit would be nearly unusable if we required the user to explicitly approve every aspect of the process. For example, if a `Topology` has its `box_vectors` attribute defined, we assume that the resulting `System` should be periodic.

1.5.2 Setting up a development environment

1. Install the `conda` package manager as part of the Anaconda Distribution from here
2. Set up conda environment

```
$ # Create a conda environment with the Open Force Field toolkit and its dependencies
$ conda create --name openff-dev -c conda-forge -c omnia -c openeye openforcefield openeye-toolkits
$ conda activate openff-dev
$ # Remove (only) the toolkit and replace it with a local install
```

(continues on next page)

(continued from previous page)

```
$ conda remove --force openforcefield
$ git clone https://github.com/openforcefield/openforcefield
$ cd openforcefield
$ pip install -e .
```

3. Obtain and store Open Eye license somewhere like `~/.oe_license.txt`. Optionally store the path in environmental variable `OE_LICENSE`, i.e. using a command like `echo "export OE_LICENSE=/Users/yournamehere/.oe_license.txt" >> ~/.bashrc`

1.5.3 Development Process

Development of new toolkit features generally proceeds in the following stages:

- **Begin a discussion on the GitHub issue tracker to determine big-picture "what should this feature do?" and "do**
  - "... typically, for existing water models, we want to assign library charges"
- **Start identifying details of the implementation that will be clear from the outset**
  - "Create a new "special section" in the SMIRNOFF format (kind of analogous to the BondChargeCorrections section) which allows SMIRKS patterns to specify use of library charges for specific groups
  - "Following #86, here's how library charges might work: ..."
- **Create a branch or fork for development**
  - The OFF Toolkit has one unusual aspect of its CI build process, which is that certain functionality requires the OpenEye toolkits, so the builds must contain a valid OpenEye license file. An encrypted OpenEye license is present in the OFF Toolkit GitHub repository, as `oe_license.txt.enc`. Only Travis has the decryption key for this file. However, this setup poses the risk that anyone who can run Travis builds could simply print the contents of the license after decryption, which would put us in violation of our academic contract with OpenEye. For this reason, the OpenEye-dependent tests will be skipped on forks.
  - Note that creating a fork will prevent the OpenEye license from being decrypted on Travis

1.5.4 Contributing

We always welcome GitHub pull requests. For bug fixes, major feature additions, or refactoring, please raise an issue on the GitHub issue tracker first to ensure the design will be amenable to current developer plans.

1.5.5 How can I become a developer?

If you would like to contribute, please post an issue on the GitHub issue tracker describing the contribution you would like to make to start a discussion.

1.5.6 Style guide

Development for the `openforcefield` toolkit conforms to the recommendations given by the Software Development Best Practices for Computational Chemistry guide.

The naming conventions of classes, functions, and variables follows PEP8, consistently with the best practices guide. The naming conventions used in this library not covered by PEP8 are: - Use `file_path`, `file_name`, and `file_stem` to indicate `path/to/stem.extension`, `stem.extension`, and `stem` respectively, consistently with the variables in the standard `pathlib` library. - Use `n_x` to abbreviate "number of X` (e.g. *n_atoms*, *n_molecules*).

We place a high priority on code cleanliness and readability. So, 15-character variable names are fine. Triply nested list comprehensions are not.

Anything not covered above is up to personal preference. To remove the human friction from code formatting, we will likely adopt a standard formatter like black in the near future.

1.6 Frequently asked questions (FAQ)

1.6.1 Input files for applying SMIRNOFF parameters

SMIRNOFF force fields use direct chemical perception meaning that, unlike many molecular mechanics (MM) force fields, they apply parameters based on substructure searches acting directly on molecules. This creates unique opportunities and allows them to encode a great deal of chemistry quite simply, but it also means that the *starting point* for parameter assignment must be well-defined chemically, giving not just the elements and connectivity for all of the atoms of all of the components of your system, but also providing the formal charges and bond orders.

Specifically, to apply SMIRNOFF to a system, you must either:

1. Provide Open Force Field Toolkit `Molecule` objects corresponding to the components of your system, or
2. Provide an OpenMM `Topology` which includes bond orders and thus can be converted to molecules corresponding to the components of your system

Without this information, our direct chemical perception cannot be applied to your molecule, as it requires the chemical identity of the molecules in your system – that is, bond order and formal charge as well as atoms and connectivity. Unless you provide the full chemical identity in this sense, we must attempt to guess or infer the chemical identity of your molecules, which is a recipe for trouble. Different molecules can have the same chemical graph but differ in bond order and formal charge, or different resonance structures may be treated rather differently by some force fields (e.g. `c1cc(ccc1c2cc[nH+]cc2)[O-]` vs `C1=CC(C=CC1=C2C=CNC=C2)=O`, where the central bond is rotatable in one resonance structure but not in the other) even though they have identical formal charge and connectivity (chemical graph). A force field which uses the chemical identity of molecules to assign parameters needs to know the exact chemical identity of the molecule you are intending to parameterize.

1.6.2 Can I use an AMBER (or GROMACS) topology/coordinate file as a starting point for applying a SMIRNOFF force field?

In a word, "no".

Parameter files used by typical molecular dynamics simulation packages do not currently encode enough information to identify the molecules chemically present, or at least not without drawing inferences. For example, one could take a structure file and infer bond orders based on bond lengths, or attempt to infer bond orders from force constants in a parameter file. Such inference work is outside the scope of SMIRNOFF.

If you have such an inference problem, we recommend that you use pre-existing cheminformatics tools available elsewhere (such as via the OpenEye toolkits, such as the `OEPerceiveBondOrders` functionality offered there) to solve this problem and identify your molecules before beginning your work with SMIRNOFF.

1.6.3 What about starting from a PDB file?

PDB files do not in general provide the chemical identity of small molecules contained therein, and thus do not provide suitable starting points for applying SMIROFF to small molecules. This is especially problematic for PDB files from X-ray crystallography which typically do not include proteins, making the problem even worse. For our purposes here, however, we assume you begin with the coordinates of all atoms present and the full topology of your system.

Given a PDB file of a hypothetical biomolecular system of interest containing a small molecule, there are several routes available to you for treating the small molecule present:

- Use a cheminformatics toolkit (see above) to infer bond orders
- Identify your ligand from a database; e.g. if it is in the Protein Data Bank (PDB), it will be present in the Ligand Expo meaning that it has a database entry and code you can use to look up its putative chemical identity
- Identify your ligand by name or SMILES string (or similar) from the literature or your collaborators

1.6.4 What do you recommend as a starting point?

For application of SMIRNOFF force fields, we recommend that you begin your work with formats which provide the chemical identity of your small molecule (including formal charge and bond order). This means we recommend one of the following or equivalent:

- A `.mol2` file or files for the molecules comprising your system, with correct bond orders and formal charges. (Note: Do NOT generate this from a simulation package or tool which does not have access to bond order information; you may end up with a `.mol2` file, but the bond orders will be incorrect)
- Isomeric SMILES strings for the components of your system
- InCHI strings for the components of your system
- Chemical Identity Registry numbers for the components of your system
- IUPAC names for the components of your system

Essentially, anything which provides the full identity of what you want to simulate (including stereochemistry) should work, though it may require more or less work to get it into an acceptable format.

CHAPTER

TWO

API DOCUMENTATION

2.1 Molecular topology representations

This module provides pure-Python classes for representing molecules and molecular systems. These classes offer several advantages over corresponding `Topology` objects in OpenMM and MDTraj, including offering serialization to a variety of standard formats (including XML, JSON, YAML, BSON, TOML, and MessagePack).

2.1.1 Primary objects

`FrozenMolecule`	Immutable chemical representation of a molecule, such as a small molecule or biopolymer.
`Molecule`	Mutable chemical representation of a molecule, such as a small molecule or biopolymer.
`Topology`	A Topology is a chemical representation of a system containing one or more molecules appearing in a specified order.
`TopologyMolecule`	TopologyMolecules are built to be an efficient way to store large numbers of copies of the same molecule for parameterization and system preparation.

openforcefield.topology.FrozenMolecule

class openforcefield.topology.**FrozenMolecule**(*other=None*, *file_format=None*, *toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*, *allow_undefined_stereo=False*)

Immutable chemical representation of a molecule, such as a small molecule or biopolymer.

Examples

Create a molecule from a sdf file

```
>>> from openforcefield.utils import get_data_file_path
>>> sdf_filepath = get_data_file_path('molecules/ethanol.sdf')
>>> molecule = FrozenMolecule.from_file(sdf_filepath)
```

Convert to OpenEye OEMol object

```
>>> oemol = molecule.to_openeye()
```

Create a molecule from an OpenEye molecule

```
>>> molecule = FrozenMolecule.from_openeye(oemol)
```

Convert to RDKit Mol object

```
>>> rdmol = molecule.to_rdkit()
```

Create a molecule from an RDKit molecule

```
>>> molecule = FrozenMolecule.from_rdkit(rdmol)
```

Create a molecule from IUPAC name (requires the OpenEye toolkit)

```
>>> molecule = FrozenMolecule.from_iupac('imatinib')
```

Create a molecule from SMILES

```
>>> molecule = FrozenMolecule.from_smiles('Cc1ccccc1')
```

Warning: This API is experimental and subject to change.

Attributes

angles Get an iterator over all i-j-k angles.

atoms Iterate over all Atom objects.

bonds Iterate over all Bond objects.

conformers Returns the list of conformers for this molecule.

has_unique_atom_names True if the molecule has unique atom names, False otherwise.

hill_formula Get the Hill formula of the molecule

impropers Iterate over all proper torsions in the molecule

n_angles int: number of angles in the Molecule.

n_atoms The number of Atom objects.

n_bonds The number of Bond objects.

n_conformers Returns the number of conformers for this molecule.

n_impropers int: number of improper torsions in the Molecule.

n_particles The number of Particle objects, which corresponds to how many positions must be used.

n_propers int: number of proper torsions in the Molecule.

n_virtual_sites The number of VirtualSite objects.

name The name (or title) of the molecule

partial_charges Returns the partial charges (if present) on the molecule.

particles Iterate over all Particle objects.

propers Iterate over all proper torsions in the molecule

properties The properties dictionary of the molecule

torsions Get an iterator over all i-j-k-l torsions.

total_charge Return the total charge on the molecule

virtual_sites Iterate over all VirtualSite objects.

Methods

`are_isomorphic`(mol1, mol2[, ...])	Determines whether the two molecules are isomorphic by comparing their graph representations and the chosen node/edge attributes.
`assign_fractional_bond_orders`([...])	Update and store list of bond orders this molecule.
`assign_partial_charges`(partial_charge_method)	Calculate partial atomic charges for this molecule using an underlying toolkit, and assign the new values to the partial_charges attribute.
`canonical_order_atoms`([toolkit_registry])	Canonical order the atoms in a copy of the molecule using a toolkit, returns a new copy.
`chemical_environment_matches`(query[, ...])	Retrieve all matches for a given chemical environment query.
`compute_partial_charges_am1bcc`([...])	Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit and assign them to this molecule's partial_charges attribute.
`enumerate_protomers`(**kwargs)	Enumerate the formal charges of a molecule to generate different protomoers.
`enumerate_stereoisomers`([undefined_only, ...])	Enumerate the stereocenters and bonds of the current molecule.
`enumerate_tautomers`([max_states, ...])	Enumerate the possible tautomers of the current molecule
`find_rotatable_bonds`([...])	Find all bonds classed as rotatable ignoring any matched to the `ignore_functional_groups` list.
`from_bson`(serialized)	Instantiate an object from a BSON serialized representation.
`from_dict`(molecule_dict)	Create a new Molecule from a dictionary representation
`from_file`(file_path[, file_format, ...])	Create one or more molecules from a file
`from_inchi`(inchi[, allow_undefined_stereo, ...])	Construct a Molecule from a InChI representation

continues on next page

Table 2 – continued from previous page

from_iupac(*args, **kwargs)	Generate a molecule from IUPAC or common name
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_mapped_smiles(mapped_smiles[, ...])	Create an openforcefield.topology.molecule.Molecule from a mapped SMILES made with cmiles.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_openeye(*args, **kwargs)	Create a Molecule from an OpenEye molecule.
from_pdb_and_smiles(*args, **kwargs)	Create a Molecule from a pdb file and a SMILES string using RDKit.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_qcschema(qca_record[, client, ...])	Create a Molecule from a QCArchive entry based on the cmiles information.
from_rdkit(*args, **kwargs)	Create a Molecule from an RDKit molecule.
from_smiles(smiles[, ...])	Construct a Molecule from a SMILES representation
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_topology(topology)	Return a Molecule representation of an openforcefield Topology containing a single Molecule object.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
generate_conformers([toolkit_registry, ...])	Generate conformers for this molecule using an underlying toolkit.
generate_unique_atom_names()	Generate unique atom names using element name and number of times that element has occurred e.g.
get_bond_between(i, j)	Returns the bond between two atoms
is_isomorphic_with(other, **kwargs)	Check if the molecule is isomorphic with the other molecule which can be an openforcefield.topology.Molecule, or TopologyMolecule or nx.Graph().
remap(mapping_dict[, current_to_new])	Remap all of the indexes in the molecule to match the given mapping dict
to_bson()	Return a BSON serialized representation.
to_dict()	Return a dictionary representation of the molecule.
to_file(file_path, file_format[, ...])	Write the current molecule to a file or file-like object
to_hill_formula(molecule)	Generate the Hill formula from either a FrozenMolecule, TopologyMolecule or nx.Graph() of the molecule
to_inchi([fixed_hydrogens, toolkit_registry])	Create an InChI string for the molecule using the requested toolkit backend.

continues on next page

Table 2 – continued from previous page

to_inchikey([fixed_hydrogens, toolkit_registry])	Create an InChIKey for the molecule using the requested toolkit backend.
to_iupac(**kwargs)	Generate IUPAC name from Molecule
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_networkx()	Generate a NetworkX undirected graph from the Molecule.
to_openeye(**kwargs)	Create an OpenEye molecule
to_pickle()	Return a pickle serialized representation.
to_qcschema([multiplicity, conformer, extras])	Generate the qschema input format used to submit jobs to archive or run qcengine calculations locally, spec can be found here <https://molssi-qc-schema.readthedocs.io/en/latest/index.html>
to_rdkit(**kwargs)	Create an RDKit molecule
to_smiles([isomeric, explicit_hydrogens, ...])	Return a canonical isomeric SMILES representation of the current molecule.
to_toml()	Return a TOML serialized representation.
to_topology()	Return an openforcefield Topology representation containing one copy of this molecule
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

__init__(*other=None*, *file_format=None*, *toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*, *allow_undefined_stereo=False*)

Create a new FrozenMolecule object

Parameters

other [optional, default=None] If specified, attempt to construct a copy of the Molecule from the specified object. This can be any one of the following:

- a `Molecule` object
- a file that can be used to construct a `Molecule` object
- an `openeye.oechem.OEMol`
- an `rdkit.Chem.rdchem.Mol`
- a serialized `Molecule` object

file_format [str, optional, default=None] If providing a file-like object, you must specify the format of the data. If providing a file, the file format will attempt to be guessed from the suffix.

toolkit_registry [a `ToolkitRegistry` or `ToolkitWrapper` object, optional, default=GLOBAL_TOOLKIT_REGISTRY] `ToolkitRegistry` or `ToolkitWrapper` to use for I/O operations

allow_undefined_stereo [bool, default=False] If loaded from a file and `False`, raises an exception if undefined stereochemistry is detected during the molecule's construction.

Examples

Create an empty molecule:

```
>>> empty_molecule = FrozenMolecule()
```

Create a molecule from a file that can be used to construct a molecule, using either a filename or file-like object:

```
>>> from openforcefield.utils import get_data_file_path
>>> sdf_filepath = get_data_file_path('molecules/ethanol.sdf')
>>> molecule = FrozenMolecule(sdf_filepath)
>>> molecule = FrozenMolecule(open(sdf_filepath, 'r'), file_format='sdf')
```

```
>>> import gzip
>>> mol2_gz_filepath = get_data_file_path('molecules/toluene.mol2.gz')
>>> molecule = FrozenMolecule(gzip.GzipFile(mol2_gz_filepath, 'r'), file_format='mol2')
```

Create a molecule from another molecule:

```
>>> molecule_copy = FrozenMolecule(molecule)
```

Convert to OpenEye OEMol object

```
>>> oemol = molecule.to_openeye()
```

Create a molecule from an OpenEye molecule:

```
>>> molecule = FrozenMolecule(oemol)
```

Convert to RDKit Mol object

```
>>> rdmol = molecule.to_rdkit()
```

Create a molecule from an RDKit molecule:

```
>>> molecule = FrozenMolecule(rdmol)
```

Create a molecule from a serialized molecule object:

```
>>> serialized_molecule = molecule.__getstate__()
>>> molecule_copy = Molecule(serialized_molecule)
```

Methods

`__init__`([other, file_format, ...])	Create a new FrozenMolecule object
`are_isomorphic`(mol1, mol2[, ...])	Determines whether the two molecules are isomorphic by comparing their graph representations and the chosen node/edge attributes.
`assign_fractional_bond_orders`([...])	Update and store list of bond orders this molecule.
`assign_partial_charges`(partial_charge_method)	Calculate partial atomic charges for this molecule using an underlying toolkit, and assign the new values to the partial_charges attribute.

continues on next page

Table 3 – continued from previous page

canonical_order_atoms([toolkit_registry])	Canonical order the atoms in a copy of the molecule using a toolkit, returns a new copy.
chemical_environment_matches(query[, ...])	Retrieve all matches for a given chemical environment query.
compute_partial_charges_am1bcc([...])	Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit and assign them to this molecule's partial_charges attribute.
enumerate_protomers(**kwargs)	Enumerate the formal charges of a molecule to generate different protomoers.
enumerate_stereoisomers([undefined_only, ...])	Enumerate the stereocenters and bonds of the current molecule.
enumerate_tautomers([max_states, ...])	Enumerate the possible tautomers of the current molecule
find_rotatable_bonds([...])	Find all bonds classed as rotatable ignoring any matched to the ignore_functional_groups list.
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(molecule_dict)	Create a new Molecule from a dictionary representation
from_file(file_path[, file_format, ...])	Create one or more molecules from a file
from_inchi(inchi[, allow_undefined_stereo, ...])	Construct a Molecule from a InChI representation
from_iupac(*args, **kwargs)	Generate a molecule from IUPAC or common name
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_mapped_smiles(mapped_smiles[, ...])	Create an openforcefield.topology.molecule.Molecule from a mapped SMILES made with cmiles.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_openeye(*args, **kwargs)	Create a Molecule from an OpenEye molecule.
from_pdb_and_smiles(*args, **kwargs)	Create a Molecule from a pdb file and a SMILES string using RDKit.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_qcschema(qca_record[, client, ...])	Create a Molecule from a QCArchive entry based on the cmiles information.
from_rdkit(*args, **kwargs)	Create a Molecule from an RDKit molecule.
from_smiles(smiles[, ...])	Construct a Molecule from a SMILES representation
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_topology(topology)	Return a Molecule representation of an openforcefield Topology containing a single Molecule object.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.

continues on next page

Table 3 – continued from previous page

generate_conformers([toolkit_registry, ...])	Generate conformers for this molecule using an underlying toolkit.
generate_unique_atom_names()	Generate unique atom names using element name and number of times that element has occurred e.g.
get_bond_between(i, j)	Returns the bond between two atoms
is_isomorphic_with(other, **kwargs)	Check if the molecule is isomorphic with the other molecule which can be an openforcefield.topology.Molecule, or TopologyMolecule or nx.Graph().
remap(mapping_dict[, current_to_new])	Remap all of the indexes in the molecule to match the given mapping dict
to_bson()	Return a BSON serialized representation.
to_dict()	Return a dictionary representation of the molecule.
to_file(file_path, file_format[, ...])	Write the current molecule to a file or file-like object
to_hill_formula(molecule)	Generate the Hill formula from either a FrozenMolecule, TopologyMolecule or nx.Graph() of the molecule
to_inchi([fixed_hydrogens, toolkit_registry])	Create an InChI string for the molecule using the requested toolkit backend.
to_inchikey([fixed_hydrogens, toolkit_registry])	Create an InChIKey for the molecule using the requested toolkit backend.
to_iupac(**kwargs)	Generate IUPAC name from Molecule
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_networkx()	Generate a NetworkX undirected graph from the Molecule.
to_openeye(**kwargs)	Create an OpenEye molecule
to_pickle()	Return a pickle serialized representation.
to_qcschema([multiplicity, conformer, extras])	Generate the qschema input format used to submit jobs to archive or run qcengine calculations locally, spec can be found here <https://molssi-qc-schema.readthedocs.io/en/latest/index.html>
to_rdkit(**kwargs)	Create an RDKit molecule
to_smiles([isomeric, explicit_hydrogens, ...])	Return a canonical isomeric SMILES representation of the current molecule.
to_toml()	Return a TOML serialized representation.
to_topology()	Return an openforcefield Topology representation containing one copy of this molecule
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

Attributes

angles	Get an iterator over all i-j-k angles.
atoms	Iterate over all Atom objects.
bonds	Iterate over all Bond objects.
conformers	Returns the list of conformers for this molecule.
has_unique_atom_names	True if the molecule has unique atom names, False otherwise.
hill_formula	Get the Hill formula of the molecule
impropers	Iterate over all proper torsions in the molecule
n_angles	int: number of angles in the Molecule.
n_atoms	The number of Atom objects.
n_bonds	The number of Bond objects.
n_conformers	Returns the number of conformers for this molecule.
n_impropers	int: number of improper torsions in the Molecule.
n_particles	The number of Particle objects, which corresponds to how many positions must be used.
n_propers	int: number of proper torsions in the Molecule.
n_virtual_sites	The number of VirtualSite objects.
name	The name (or title) of the molecule
partial_charges	Returns the partial charges (if present) on the molecule.
particles	Iterate over all Particle objects.
propers	Iterate over all proper torsions in the molecule
properties	The properties dictionary of the molecule
torsions	Get an iterator over all i-j-k-l torsions.
total_charge	Return the total charge on the molecule
virtual_sites	Iterate over all VirtualSite objects.

openforcefield.topology.Molecule

class openforcefield.topology.**Molecule**(**args*, ***kwargs*)

Mutable chemical representation of a molecule, such as a small molecule or biopolymer.

Examples

Create a molecule from an sdf file

```
>>> from openforcefield.utils import get_data_file_path
>>> sdf_filepath = get_data_file_path('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
```

Convert to OpenEye OEMol object

```
>>> oemol = molecule.to_openeye()
```

Create a molecule from an OpenEye molecule

```
>>> molecule = Molecule.from_openeye(oemol)
```

Convert to RDKit Mol object

```
>>> rdmol = molecule.to_rdkit()
```

Create a molecule from an RDKit molecule

```
>>> molecule = Molecule.from_rdkit(rdmol)
```

Create a molecule from IUPAC name (requires the OpenEye toolkit)

```
>>> molecule = Molecule.from_iupac('imatinib')
```

Create a molecule from SMILES

```
>>> molecule = Molecule.from_smiles('Cc1ccccc1')
```

Warning: This API is experimental and subject to change.

Attributes

angles Get an iterator over all i-j-k angles.

atoms Iterate over all Atom objects.

bonds Iterate over all Bond objects.

conformers Returns the list of conformers for this molecule.

has_unique_atom_names True if the molecule has unique atom names, False otherwise.

hill_formula Get the Hill formula of the molecule

impropers Iterate over all proper torsions in the molecule

n_angles int: number of angles in the Molecule.

n_atoms The number of Atom objects.

n_bonds The number of Bond objects.

n_conformers Returns the number of conformers for this molecule.

n_impropers int: number of improper torsions in the Molecule.

n_particles The number of Particle objects, which corresponds to how many positions must be used.

n_propers int: number of proper torsions in the Molecule.

n_virtual_sites The number of VirtualSite objects.

name The name (or title) of the molecule

partial_charges Returns the partial charges (if present) on the molecule.

particles Iterate over all Particle objects.

propers Iterate over all proper torsions in the molecule

properties The properties dictionary of the molecule

torsions Get an iterator over all i-j-k-l torsions.

total_charge Return the total charge on the molecule

virtual_sites Iterate over all VirtualSite objects.

Methods

add_atom(atomic_number, formal_charge, ...)	Add an atom
add_bond(atom1, atom2, bond_order, is_aromatic)	Add a bond between two specified atom indices
add_bond_charge_virtual_site(atoms, distance)	Create a bond charge-type virtual site, in which the location of the charge is specified by the position of two atoms.
add_conformer(coordinates)	Add a conformation of the molecule
add_divalent_lone_pair_virtual_site(atoms, ...)	Create a divalent lone pair-type virtual site, in which the location of the charge is specified by the position of three atoms.
add_monovalent_lone_pair_virtual_site(atoms, ...)	Create a bond charge-type virtual site, in which the location of the charge is specified by the position of three atoms.
add_trivalent_lone_pair_virtual_site(atoms, ...)	Create a trivalent lone pair-type virtual site, in which the location of the charge is specified by the position of four atoms.
are_isomorphic(mol1, mol2[, ...])	Determines whether the two molecules are isomorphic by comparing their graph representations and the chosen node/edge attributes.
assign_fractional_bond_orders([...])	Update and store list of bond orders this molecule.
assign_partial_charges(partial_charge_method)	Calculate partial atomic charges for this molecule using an underlying toolkit, and assign the new values to the partial_charges attribute.
canonical_order_atoms([toolkit_registry])	Canonical order the atoms in a copy of the molecule using a toolkit, returns a new copy.
chemical_environment_matches(query[, ...])	Retrieve all matches for a given chemical environment query.
compute_partial_charges_am1bcc([...])	Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit and assign them to this molecule's partial_charges attribute.
enumerate_protomers(**kwargs)	Enumerate the formal charges of a molecule to generate different protomoers.
enumerate_stereoisomers([undefined_only, ...])	Enumerate the stereocenters and bonds of the current molecule.
enumerate_tautomers([max_states, ...])	Enumerate the possible tautomers of the current molecule
find_rotatable_bonds([...])	Find all bonds classed as rotatable ignoring any matched to the ignore_functional_groups list.
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(molecule_dict)	Create a new Molecule from a dictionary representation
from_file(file_path[, file_format, ...])	Create one or more molecules from a file
from_inchi(inchi[, allow_undefined_stereo, ...])	Construct a Molecule from a InChI representation

continues on next page

Table 5 – continued from previous page

from_iupac(*args, **kwargs)	Generate a molecule from IUPAC or common name
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_mapped_smiles(mapped_smiles[, ...])	Create an openforcefield.topology.molecule.Molecule from a mapped SMILES made with cmiles.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_openeye(*args, **kwargs)	Create a Molecule from an OpenEye molecule.
from_pdb_and_smiles(*args, **kwargs)	Create a Molecule from a pdb file and a SMILES string using RDKit.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_qcschema(qca_record[, client, ...])	Create a Molecule from a QCArchive entry based on the cmiles information.
from_rdkit(*args, **kwargs)	Create a Molecule from an RDKit molecule.
from_smiles(smiles[, ...])	Construct a Molecule from a SMILES representation
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_topology(topology)	Return a Molecule representation of an openforcefield Topology containing a single Molecule object.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
generate_conformers([toolkit_registry, ...])	Generate conformers for this molecule using an underlying toolkit.
generate_unique_atom_names()	Generate unique atom names using element name and number of times that element has occurred e.g.
get_bond_between(i, j)	Returns the bond between two atoms
is_isomorphic_with(other, **kwargs)	Check if the molecule is isomorphic with the other molecule which can be an openforcefield.topology.Molecule, or TopologyMolecule or nx.Graph().
remap(mapping_dict[, current_to_new])	Remap all of the indexes in the molecule to match the given mapping dict
to_bson()	Return a BSON serialized representation.
to_dict()	Return a dictionary representation of the molecule.
to_file(file_path, file_format[, ...])	Write the current molecule to a file or file-like object
to_hill_formula(molecule)	Generate the Hill formula from either a FrozenMolecule, TopologyMolecule or nx.Graph() of the molecule
to_inchi([fixed_hydrogens, toolkit_registry])	Create an InChI string for the molecule using the requested toolkit backend.

continues on next page

Table 5 – continued from previous page

to_inchikey([fixed_hydrogens, toolkit_registry])	Create an InChIKey for the molecule using the requested toolkit backend.
to_iupac(**kwargs)	Generate IUPAC name from Molecule
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_networkx()	Generate a NetworkX undirected graph from the Molecule.
to_openeye(**kwargs)	Create an OpenEye molecule
to_pickle()	Return a pickle serialized representation.
to_qcschema([multiplicity, conformer, extras])	Generate the qschema input format used to submit jobs to archive or run qcengine calculations locally, spec can be found here <https://molssi-qc-schema.readthedocs.io/en/latest/index.html>
to_rdkit(**kwargs)	Create an RDKit molecule
to_smiles([isomeric, explicit_hydrogens, ...])	Return a canonical isomeric SMILES representation of the current molecule.
to_toml()	Return a TOML serialized representation.
to_topology()	Return an openforcefield Topology representation containing one copy of this molecule
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.
visualize([backend, width, height])	Render a visualization of the molecule in Jupyter

__init__(**args*, ***kwargs*)

Create a new Molecule object

Parameters

other [optional, default=None] If specified, attempt to construct a copy of the Molecule from the specified object. This can be any one of the following:

- a `Molecule` object
- a file that can be used to construct a `Molecule` object
- an `openeye.oechem.OEMol`
- an `rdkit.Chem.rdchem.Mol`
- a serialized `Molecule` object

Examples

Create an empty molecule:

```
>>> empty_molecule = Molecule()
```

Create a molecule from a file that can be used to construct a molecule, using either a filename or file-like object:

```
>>> from openforcefield.utils import get_data_file_path
>>> sdf_filepath = get_data_file_path('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
>>> molecule = Molecule(open(sdf_filepath, 'r'), file_format='sdf')
```

```
>>> import gzip
>>> mol2_gz_filepath = get_data_file_path('molecules/toluene.mol2.gz')
>>> molecule = Molecule(gzip.GzipFile(mol2_gz_filepath, 'r'), file_format='mol2')
```

Create a molecule from another molecule:

```
>>> molecule_copy = Molecule(molecule)
```

Convert to OpenEye OEMol object

```
>>> oemol = molecule.to_openeye()
```

Create a molecule from an OpenEye molecule:

```
>>> molecule = Molecule(oemol)
```

Convert to RDKit Mol object

```
>>> rdmol = molecule.to_rdkit()
```

Create a molecule from an RDKit molecule:

```
>>> molecule = Molecule(rdmol)
```

Create a molecule from a serialized molecule object:

```
>>> serialized_molecule = molecule.__getstate__()
>>> molecule_copy = Molecule(serialized_molecule)
```

Methods

`__init__`(*args, **kwargs)	Create a new Molecule object
`add_atom`(atomic_number, formal_charge, ...)	Add an atom
`add_bond`(atom1, atom2, bond_order, is_aromatic)	Add a bond between two specified atom indices
`add_bond_charge_virtual_site`(atoms, distance)	Create a bond charge-type virtual site, in which the location of the charge is specified by the position of two atoms.
`add_conformer`(coordinates)	Add a conformation of the molecule
`add_divalent_lone_pair_virtual_site`(atoms, ...)	Create a divalent lone pair-type virtual site, in which the location of the charge is specified by the position of three atoms.
`add_monovalent_lone_pair_virtual_site`(atoms, ...)	Create a bond charge-type virtual site, in which the location of the charge is specified by the position of three atoms.
`add_trivalent_lone_pair_virtual_site`(atoms, ...)	Create a trivalent lone pair-type virtual site, in which the location of the charge is specified by the position of four atoms.
`are_isomorphic`(mol1, mol2[, ...])	Determines whether the two molecules are isomorphic by comparing their graph representations and the chosen node/edge attributes.

continues on next page

Table 6 – continued from previous page

assign_fractional_bond_orders([...])	Update and store list of bond orders this molecule.
assign_partial_charges(partial_charge_method)	Calculate partial atomic charges for this molecule using an underlying toolkit, and assign the new values to the partial_charges attribute.
canonical_order_atoms([toolkit_registry])	Canonical order the atoms in a copy of the molecule using a toolkit, returns a new copy.
chemical_environment_matches(query[, ...])	Retrieve all matches for a given chemical environment query.
compute_partial_charges_am1bcc([...])	Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit and assign them to this molecule's partial_charges attribute.
enumerate_protomers(**kwargs)	Enumerate the formal charges of a molecule to generate different protomoers.
enumerate_stereoisomers([undefined_only, ...])	Enumerate the stereocenters and bonds of the current molecule.
enumerate_tautomers([max_states, ...])	Enumerate the possible tautomers of the current molecule
find_rotatable_bonds([...])	Find all bonds classed as rotatable ignoring any matched to the ignore_functional_groups list.
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(molecule_dict)	Create a new Molecule from a dictionary representation
from_file(file_path[, file_format, ...])	Create one or more molecules from a file
from_inchi(inchi[, allow_undefined_stereo, ...])	Construct a Molecule from a InChI representation
from_iupac(*args, **kwargs)	Generate a molecule from IUPAC or common name
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_mapped_smiles(mapped_smiles[, ...])	Create an openforcefield.topology.molecule.Molecule from a mapped SMILES made with cmiles.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_openeye(*args, **kwargs)	Create a Molecule from an OpenEye molecule.
from_pdb_and_smiles(*args, **kwargs)	Create a Molecule from a pdb file and a SMILES string using RDKit.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_qcschema(qca_record[, client, ...])	Create a Molecule from a QCArchive entry based on the cmiles information.
from_rdkit(*args, **kwargs)	Create a Molecule from an RDKit molecule.
from_smiles(smiles[, ...])	Construct a Molecule from a SMILES representation
from_toml(serialized)	Instantiate an object from a TOML serialized representation.

continues on next page

Table 6 – continued from previous page

from_topology(topology)	Return a Molecule representation of an openforcefield Topology containing a single Molecule object.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
generate_conformers([toolkit_registry, ...])	Generate conformers for this molecule using an underlying toolkit.
generate_unique_atom_names()	Generate unique atom names using element name and number of times that element has occurred e.g.
get_bond_between(i, j)	Returns the bond between two atoms
is_isomorphic_with(other, **kwargs)	Check if the molecule is isomorphic with the other molecule which can be an openforcefield.topology.Molecule, or TopologyMolecule or nx.Graph().
remap(mapping_dict[, current_to_new])	Remap all of the indexes in the molecule to match the given mapping dict
to_bson()	Return a BSON serialized representation.
to_dict()	Return a dictionary representation of the molecule.
to_file(file_path, file_format[, ...])	Write the current molecule to a file or file-like object
to_hill_formula(molecule)	Generate the Hill formula from either a FrozenMolecule, TopologyMolecule or nx.Graph() of the molecule
to_inchi([fixed_hydrogens, toolkit_registry])	Create an InChI string for the molecule using the requested toolkit backend.
to_inchikey([fixed_hydrogens, toolkit_registry])	Create an InChIKey for the molecule using the requested toolkit backend.
to_iupac(**kwargs)	Generate IUPAC name from Molecule
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_networkx()	Generate a NetworkX undirected graph from the Molecule.
to_openeye(**kwargs)	Create an OpenEye molecule
to_pickle()	Return a pickle serialized representation.
to_qcschema([multiplicity, conformer, extras])	Generate the qschema input format used to submit jobs to archive or run qcengine calculations locally, spec can be found here <https://molssi-qc-schema.readthedocs.io/en/latest/index.html>
to_rdkit(**kwargs)	Create an RDKit molecule
to_smiles([isomeric, explicit_hydrogens, ...])	Return a canonical isomeric SMILES representation of the current molecule.
to_toml()	Return a TOML serialized representation.
to_topology()	Return an openforcefield Topology representation containing one copy of this molecule
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

continues on next page

Table 6 – continued from previous page

visualize([backend, width, height])	Render a visualization of the molecule in Jupyter

Attributes

angles	Get an iterator over all i-j-k angles.
atoms	Iterate over all Atom objects.
bonds	Iterate over all Bond objects.
conformers	Returns the list of conformers for this molecule.
has_unique_atom_names	True if the molecule has unique atom names, False otherwise.
hill_formula	Get the Hill formula of the molecule
impropers	Iterate over all proper torsions in the molecule
n_angles	int: number of angles in the Molecule.
n_atoms	The number of Atom objects.
n_bonds	The number of Bond objects.
n_conformers	Returns the number of conformers for this molecule.
n_impropers	int: number of improper torsions in the Molecule.
n_particles	The number of Particle objects, which corresponds to how many positions must be used.
n_propers	int: number of proper torsions in the Molecule.
n_virtual_sites	The number of VirtualSite objects.
name	The name (or title) of the molecule
partial_charges	Returns the partial charges (if present) on the molecule.
particles	Iterate over all Particle objects.
propers	Iterate over all proper torsions in the molecule
properties	The properties dictionary of the molecule
torsions	Get an iterator over all i-j-k-l torsions.
total_charge	Return the total charge on the molecule
virtual_sites	Iterate over all VirtualSite objects.

openforcefield.topology.Topology

class openforcefield.topology.**Topology**(*other=None*)

A Topology is a chemical representation of a system containing one or more molecules appearing in a specified order.

As of the 0.7.0 release, the Topology particle indexing system puts all atoms before all virtualsites. This ensures that atoms keep the same Topology particle index value, even if the Topology is modified during system creation by the addition of virtual sites.

Warning: This API is experimental and subject to change.

Examples

Import some utilities

```
>>> from simtk.openmm import app
>>> from openforcefield.tests.utils import get_data_file_path, get_packmol_pdb_file_path
>>> pdb_filepath = get_packmol_pdb_file_path('cyclohexane_ethanol_0.4_0.6')
>>> monomer_names = ('cyclohexane', 'ethanol')
```

Create a Topology object from a PDB file and sdf files defining the molecular contents

```
>>> from openforcefield.topology import Molecule, Topology
>>> pdbfile = app.PDBFile(pdb_filepath)
>>> sdf_filepaths = [get_data_file_path(f'systems/monomers/{name}.sdf') for name in monomer_names]
>>> unique_molecules = [Molecule.from_file(sdf_filepath) for sdf_filepath in sdf_filepaths]
>>> topology = Topology.from_openmm(pdbfile.topology, unique_molecules=unique_molecules)
```

Create a Topology object from a PDB file and IUPAC names of the molecular contents

```
>>> pdbfile = app.PDBFile(pdb_filepath)
>>> unique_molecules = [Molecule.from_iupac(name) for name in monomer_names]
>>> topology = Topology.from_openmm(pdbfile.topology, unique_molecules=unique_molecules)
```

Create an empty Topology object and add a few copies of a single benzene molecule

```
>>> topology = Topology()
>>> molecule = Molecule.from_iupac('benzene')
>>> molecule_topology_indices = [topology.add_molecule(molecule) for index in range(10)]
```

Attributes

angles Iterable of Tuple[TopologyAtom]: iterator over the angles in this Topology.

aromaticity_model Get the aromaticity model applied to all molecules in the topology.

box_vectors Return the box vectors of the topology, if specified

charge_model Get the partial charge model applied to all molecules in the topology.

constrained_atom_pairs Returns the constrained atom pairs of the Topology

fractional_bond_order_model Get the fractional bond order model for the Topology.

impropers Iterable of Tuple[TopologyAtom]: iterator over the improper torsions in this Topology.

n_angles int: number of angles in this Topology.

n_impropers int: number of improper torsions in this Topology.

n_propers int: number of proper torsions in this Topology.

n_reference_molecules Returns the number of reference (unique) molecules in in this Topology.

n_topology_atoms Returns the number of topology atoms in in this Topology.

n_topology_bonds Returns the number of TopologyBonds in in this Topology.

n_topology_molecules Returns the number of topology molecules in in this Topology.

n_topology_particles Returns the number of topology particles (TopologyAtoms and TopologyVirtualSites) in in this Topology.

n_topology_virtual_sites Returns the number of TopologyVirtualSites in in this Topology.

propers Iterable of Tuple[TopologyAtom]: iterator over the proper torsions in this Topology.

reference_molecules Get an iterator of reference molecules in this Topology.

topology_atoms Returns an iterator over the atoms in this Topology.

topology_bonds Returns an iterator over the bonds in this Topology

topology_molecules Returns an iterator over all the TopologyMolecules in this Topology

topology_particles Returns an iterator over the particles (TopologyAtoms and TopologyVirtualSites) in this Topology.

topology_virtual_sites Get an iterator over the virtual sites in this Topology

Methods

add_constraint(iatom, jatom[, distance])	Mark a pair of atoms as constrained.
add_molecule(molecule[, ...])	Add a Molecule to the Topology.
add_particle(particle)	Add a Particle to the Topology.
assert_bonded(atom1, atom2)	Raise an exception if the specified atoms are not bonded in the topology.
atom(atom_topology_index)	Get the TopologyAtom at a given Topology atom index.
bond(bond_topology_index)	Get the TopologyBond at a given Topology bond index.
chemical_environment_matches(query[, ...])	Retrieve all matches for a given chemical environment query.
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(d)	Static constructor from dictionary representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_mdtraj(mdtraj_topology[, unique_molecules])	Construct an openforcefield Topology object from an MDTraj Topology object.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_molecules(molecules)	Create a new Topology object containing one copy of each of the specified molecule(s).
from_openmm(openmm_topology[, unique_molecules])	Construct an openforcefield Topology object from an OpenMM Topology object.

continues on next page

Table 8 – continued from previous page

from_parmed(parmed_structure[, unique_molecules])	**Warning:** This functionality will be implemented in a future toolkit release.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
get_bond_between(i, j)	Returns the bond between two atoms
is_bonded(i, j)	Returns True if the two atoms are bonded
is_constrained(iatom, jatom)	Check if a pair of atoms are marked as constrained.
to_bson()	Return a BSON serialized representation.
to_dict()	Convert to dictionary representation.
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_openmm([ensure_unique_atom_names])	Create an OpenMM Topology object.

continues on next page

Table 8 – continued from previous page

`to_parmed()`	**Warning:** This functionality will be implemented in a future toolkit release.
`to_pickle()`	Return a pickle serialized representation.
`to_toml()`	Return a TOML serialized representation.
`to_xml([indent])`	Return an XML representation.
`to_yaml()`	Return a YAML serialized representation.
`virtual_site(vsite_topology_index)`	Get the TopologyAtom at a given Topology atom index.

__init__(*other=None*)

Create a new Topology.

Parameters

other [optional, default=None] If specified, attempt to construct a copy of the Topology from the specified object. This might be a Topology object, or a file that can be used to construct a Topology object or serialized Topology object.

Methods

`__init__([other])`	Create a new Topology.
`add_constraint(iatom, jatom[, distance])`	Mark a pair of atoms as constrained.
`add_molecule(molecule[, ...])`	Add a Molecule to the Topology.
`add_particle(particle)`	Add a Particle to the Topology.
`assert_bonded(atom1, atom2)`	Raise an exception if the specified atoms are not bonded in the topology.
`atom(atom_topology_index)`	Get the TopologyAtom at a given Topology atom index.
`bond(bond_topology_index)`	Get the TopologyBond at a given Topology bond index.

continues on next page

Table 9 – continued from previous page

chemical_environment_matches(query[, ...])	Retrieve all matches for a given chemical environment query.
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(d)	Static constructor from dictionary representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_mdtraj(mdtraj_topology[, unique_molecules])	Construct an openforcefield Topology object from an MDTraj Topology object.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_molecules(molecules)	Create a new Topology object containing one copy of each of the specified molecule(s).
from_openmm(openmm_topology[, unique_molecules])	Construct an openforcefield Topology object from an OpenMM Topology object.
from_parmed(parmed_structure[, unique_molecules])	**Warning:** This functionality will be implemented in a future toolkit release.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
get_bond_between(i, j)	Returns the bond between two atoms
is_bonded(i, j)	Returns True if the two atoms are bonded
is_constrained(iatom, jatom)	Check if a pair of atoms are marked as constrained.
to_bson()	Return a BSON serialized representation.
to_dict()	Convert to dictionary representation.

continues on next page

Table 9 – continued from previous page

to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_openmm([ensure_unique_atom_names])	Create an OpenMM Topology object.
to_parmed()	

Warning: This functionality will be implemented in a future toolkit release.

to_pickle()	Return a pickle serialized representation.
to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.
virtual_site(vsite_topology_index)	Get the TopologyAtom at a given Topology atom index.

Attributes

angles	Iterable of Tuple[TopologyAtom]: iterator over the angles in this Topology.
aromaticity_model	Get the aromaticity model applied to all molecules in the topology.
box_vectors	Return the box vectors of the topology, if specified Returns ——- box_vectors : simtk.unit.Quantity wrapped numpy array The unit-wrapped box vectors of this topology
charge_model	Get the partial charge model applied to all molecules in the topology.
constrained_atom_pairs	Returns the constrained atom pairs of the Topology
fractional_bond_order_model	Get the fractional bond order model for the Topology.
impropers	Iterable of Tuple[TopologyAtom]: iterator over the improper torsions in this Topology.

continues on next page

Table 10 – continued from previous page

n_angles	int: number of angles in this Topology.
n_impropers	int: number of improper torsions in this Topology.
n_propers	int: number of proper torsions in this Topology.
n_reference_molecules	Returns the number of reference (unique) molecules in in this Topology.
n_topology_atoms	Returns the number of topology atoms in in this Topology.
n_topology_bonds	Returns the number of TopologyBonds in in this Topology.
n_topology_molecules	Returns the number of topology molecules in in this Topology.
n_topology_particles	Returns the number of topology particles (TopologyAtoms and TopologyVirtualSites) in in this Topology.
n_topology_virtual_sites	Returns the number of TopologyVirtualSites in in this Topology.
propers	Iterable of Tuple[TopologyAtom]: iterator over the proper torsions in this Topology.
reference_molecules	Get an iterator of reference molecules in this Topology.
topology_atoms	Returns an iterator over the atoms in this Topology.
topology_bonds	Returns an iterator over the bonds in this Topology
topology_molecules	Returns an iterator over all the TopologyMolecules in this Topology
topology_particles	Returns an iterator over the particles (TopologyAtoms and TopologyVirtualSites) in this Topology.
topology_virtual_sites	Get an iterator over the virtual sites in this Topology

openforcefield.topology.TopologyMolecule

class openforcefield.topology.**TopologyMolecule**(*reference_molecule*, *topology*, *local_topology_to_reference_index=None*)

TopologyMolecules are built to be an efficient way to store large numbers of copies of the same molecule for parameterization and system preparation.

> **Warning:** This API is experimental and subject to change.

Attributes

angles Iterable of Tuple[TopologyAtom]: iterator over the angles in this Topology.

atom_start_topology_index Get the topology index of the first atom in this TopologyMolecule

atoms Return an iterator of all the TopologyAtoms in this TopologyMolecule

bond_start_topology_index Get the topology index of the first bond in this TopologyMolecule

bonds Return an iterator of all the TopologyBonds in this TopologyMolecule

impropers Iterable of Tuple[TopologyAtom]: iterator over the improper torsions in this Topology.

n_angles int: number of angles in this Topology.

n_atoms The number of atoms in this topology.

n_bonds Get the number of bonds in this TopologyMolecule

n_impropers int: number of proper torsions in this Topology.

n_particles Get the number of particles in this TopologyMolecule

n_propers int: number of proper torsions in this Topology.

n_virtual_sites Get the number of virtual sites in this TopologyMolecule

particles Return an iterator of all the TopologyParticles in this TopologyMolecules

propers Iterable of Tuple[TopologyAtom]: iterator over the proper torsions in this Topology.

reference_molecule Get the reference molecule for this TopologyMolecule

topology Get the topology that this TopologyMolecule belongs to

virtual_site_start_topology_index Get the topology index of the first virtual site in this TopologyMolecule

virtual_sites Return an iterator of all the TopologyVirtualSites in this TopologyMolecules

Methods

`atom`(index)	Get the TopologyAtom with a given topology atom index in this TopologyMolecule.
`bond`(index)	Get the TopologyBond with a given reference molecule index in this TopologyMolecule
`from_dict`(d)	Static constructor from dictionary representation.
`particle`(index)	Get the TopologyParticle with a given reference molecule index in this TopologyMolecule
`to_dict`()	Convert to dictionary representation.
`virtual_site`(index)	Get the TopologyVirtualSite with a given reference molecule index in this TopologyMolecule

__init__(*reference_molecule*, *topology*, *local_topology_to_reference_index=None*)
Create a new TopologyMolecule.

Parameters

reference_molecule [an openforcefield.topology.molecule.Molecule] The reference molecule, with details like formal charges, partial charges, bond orders, partial bond orders, and atomic symbols.

topology [an openforcefield.topology.Topology] The topology that this Topology-

Molecule belongs to

local_topology_to_reference_index [dict, optional, default=None] Dictionary of {TopologyMolecule_atom_index : Molecule_atom_index} for the TopologyMolecule that will be built

Methods

`__init__`(reference_molecule, topology[, ...])	Create a new TopologyMolecule.
`atom`(index)	Get the TopologyAtom with a given topology atom index in this TopologyMolecule.
`bond`(index)	Get the TopologyBond with a given reference molecule index in this TopologyMolecule
`from_dict`(d)	Static constructor from dictionary representation.
`particle`(index)	Get the TopologyParticle with a given reference molecule index in this TopologyMolecule
`to_dict`()	Convert to dictionary representation.
`virtual_site`(index)	Get the TopologyVirtualSite with a given reference molecule index in this TopologyMolecule

Attributes

`angles`	Iterable of Tuple[TopologyAtom]: iterator over the angles in this Topology.
`atom_start_topology_index`	Get the topology index of the first atom in this TopologyMolecule
`atoms`	Return an iterator of all the TopologyAtoms in this TopologyMolecule
`bond_start_topology_index`	Get the topology index of the first bond in this TopologyMolecule
`bonds`	Return an iterator of all the TopologyBonds in this TopologyMolecule
`impropers`	Iterable of Tuple[TopologyAtom]: iterator over the improper torsions in this Topology.
`n_angles`	int: number of angles in this Topology.
`n_atoms`	The number of atoms in this topology.
`n_bonds`	Get the number of bonds in this TopologyMolecule
`n_impropers`	int: number of proper torsions in this Topology.
`n_particles`	Get the number of particles in this TopologyMolecule
`n_propers`	int: number of proper torsions in this Topology.
`n_virtual_sites`	Get the number of virtual sites in this TopologyMolecule
`particles`	Return an iterator of all the TopologyParticles in this TopologyMolecules
`propers`	Iterable of Tuple[TopologyAtom]: iterator over the proper torsions in this Topology.

continues on next page

Table 13 – continued from previous page

`reference_molecule`	Get the reference molecule for this TopologyMolecule
`topology`	Get the topology that this TopologyMolecule belongs to
`virtual_site_start_topology_index`	Get the topology index of the first virtual site in this TopologyMolecule
`virtual_sites`	Return an iterator of all the TopologyVirtualSites in this TopologyMolecules

2.1.2 Secondary objects

`Particle`	Base class for all particles in a molecule.
`Atom`	A particle representing a chemical atom.
`Bond`	Chemical bond representation.
`VirtualSite`	A particle representing a virtual site whose position is defined in terms of `Atom` positions.
`TopologyAtom`	A TopologyAtom is a lightweight data structure that represents a single openforcefield.topology.molecule.Atom in a Topology.
`TopologyBond`	A TopologyBond is a lightweight data structure that represents a single openforcefield.topology.molecule.Bond in a Topology.
`TopologyVirtualSite`	A TopologyVirtualSite is a lightweight data structure that represents a single openforcefield.topology.molecule.VirtualSite in a Topology.

openforcefield.topology.Particle

class openforcefield.topology.**Particle**

Base class for all particles in a molecule.

A particle object could be an `Atom` or a `VirtualSite`.

> **Warning:** This API is experimental and subject to change.

Attributes

molecule The `Molecule` this atom is part of.

molecule_particle_index Returns the index of this particle in its molecule

name The name of the particle

Methods

from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(d)	Static constructor from dictionary representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
to_bson()	Return a BSON serialized representation.
to_dict()	Convert to dictionary representation.
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_pickle()	Return a pickle serialized representation.
to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(*args, **kwargs)	Initialize self.
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(d)	Static constructor from dictionary representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
to_bson()	Return a BSON serialized representation.

continues on next page

Table 16 – continued from previous page

to_dict()	Convert to dictionary representation.
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_pickle()	Return a pickle serialized representation.
to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

Attributes

molecule	The Molecule this atom is part of.
molecule_particle_index	Returns the index of this particle in its molecule
name	The name of the particle

openforcefield.topology.Atom

class openforcefield.topology.**Atom**(*atomic_number*, *formal_charge*, *is_aromatic*, *name=None*, *molecule=None*, *stereochemistry=None*)

A particle representing a chemical atom.

Note that non-chemical virtual sites are represented by the VirtualSite object.

> **Warning:** This API is experimental and subject to change.

Attributes

atomic_number The integer atomic number of the atom.

bonded_atoms The list of Atom objects this atom is involved in bonds with

bonds The list of Bond objects this atom is involved in.

element The element name

formal_charge The atom's formal charge

is_aromatic The atom's is_aromatic flag

mass The standard atomic weight (abundance-weighted isotopic mass) of the atomic site.

molecule The Molecule this atom is part of.

molecule_atom_index The index of this Atom within the the list of atoms in Molecules.

molecule_particle_index The index of this Atom within the the list of particles in the parent Molecule.

name The name of this atom, if any

partial_charge The partial charge of the atom, if any.

stereochemistry The atom's stereochemistry (if defined, otherwise None)

virtual_sites The list of VirtualSite objects this atom is involved in.

Methods

add_bond(bond)	Adds a bond that this atom is involved in ..
add_virtual_site(vsite)	Adds a bond that this atom is involved in ..
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(atom_dict)	Create an Atom from a dict representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
is_bonded_to(atom2)	Determine whether this atom is bound to another atom
to_bson()	Return a BSON serialized representation.
to_dict()	Return a dict representation of the atom.
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_pickle()	Return a pickle serialized representation.
to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

__init__(*atomic_number*, *formal_charge*, *is_aromatic*, *name=None*, *molecule=None*, *stereochemistry=None*)

Create an immutable Atom object.

Object is serializable and immutable.

Parameters

atomic_number [int] Atomic number of the atom

formal_charge [int or simtk.unit.Quantity-wrapped int with dimension “charge”] Formal charge of the atom

is_aromatic [bool] If True, atom is aromatic; if False, not aromatic

stereochemistry [str, optional, default=None] Either ‘R’ or ‘S’ for specified stereochemistry, or None for ambiguous stereochemistry

name [str, optional, default=None] An optional name to be associated with the atom

Examples

Create a non-aromatic carbon atom

```
>>> atom = Atom(6, 0, False)
```

Create a chiral carbon atom

```
>>> atom = Atom(6, 0, False, stereochemistry='R', name='CT')
```

Methods

`__init__`(atomic_number, formal_charge, ...)	Create an immutable Atom object.
`add_bond`(bond)	Adds a bond that this atom is involved in ..
`add_virtual_site`(vsite)	Adds a bond that this atom is involved in ..
`from_bson`(serialized)	Instantiate an object from a BSON serialized representation.
`from_dict`(atom_dict)	Create an Atom from a dict representation.
`from_json`(serialized)	Instantiate an object from a JSON serialized representation.
`from_messagepack`(serialized)	Instantiate an object from a MessagePack serialized representation.
`from_pickle`(serialized)	Instantiate an object from a pickle serialized representation.
`from_toml`(serialized)	Instantiate an object from a TOML serialized representation.
`from_xml`(serialized)	Instantiate an object from an XML serialized representation.
`from_yaml`(serialized)	Instantiate from a YAML serialized representation.
`is_bonded_to`(atom2)	Determine whether this atom is bound to another atom
`to_bson`()	Return a BSON serialized representation.
`to_dict`()	Return a dict representation of the atom.
`to_json`([indent])	Return a JSON serialized representation.
`to_messagepack`()	Return a MessagePack representation.
`to_pickle`()	Return a pickle serialized representation.
`to_toml`()	Return a TOML serialized representation.
`to_xml`([indent])	Return an XML representation.
`to_yaml`()	Return a YAML serialized representation.

Attributes

`atomic_number`	The integer atomic number of the atom.
`bonded_atoms`	The list of `Atom` objects this atom is involved in bonds with
`bonds`	The list of `Bond` objects this atom is involved in.
`element`	The element name
`formal_charge`	The atom's formal charge

continues on next page

Table 20 – continued from previous page

`is_aromatic`	The atom's is_aromatic flag
`mass`	The standard atomic weight (abundance-weighted isotopic mass) of the atomic site.
`molecule`	The `Molecule` this atom is part of.
`molecule_atom_index`	The index of this Atom within the the list of atoms in `Molecules`.
`molecule_particle_index`	The index of this Atom within the the list of particles in the parent `Molecule`.
`name`	The name of this atom, if any
`partial_charge`	The partial charge of the atom, if any.
`stereochemistry`	The atom's stereochemistry (if defined, otherwise None)
`virtual_sites`	The list of `VirtualSite` objects this atom is involved in.

openforcefield.topology.Bond

class openforcefield.topology.**Bond**(*atom1*, *atom2*, *bond_order*, *is_aromatic*, *fractional_bond_order=None*, *stereochemistry=None*)

Chemical bond representation.

Warning: This API is experimental and subject to change.

Attributes

atom1, atom2 [openforcefield.topology.Atom] Atoms involved in the bond

bondtype [int] Discrete bond type representation for the Open Forcefield aromaticity model TODO: Do we want to pin ourselves to a single standard aromaticity model?

type [str] String based bond type

order [int] Integral bond order

fractional_bond_order [float, optional] Fractional bond order, or None.

.. warning :: This API is experimental and subject to change.

Methods

`from_bson`(serialized)	Instantiate an object from a BSON serialized representation.
`from_dict`(molecule, d)	Create a Bond from a dict representation.
`from_json`(serialized)	Instantiate an object from a JSON serialized representation.
`from_messagepack`(serialized)	Instantiate an object from a MessagePack serialized representation.
`from_pickle`(serialized)	Instantiate an object from a pickle serialized representation.
`from_toml`(serialized)	Instantiate an object from a TOML serialized representation.

continues on next page

Table 21 – continued from previous page

from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
to_bson()	Return a BSON serialized representation.
to_dict()	Return a dict representation of the bond.
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_pickle()	Return a pickle serialized representation.
to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

__init__(*atom1*, *atom2*, *bond_order*, *is_aromatic*, *fractional_bond_order=None*, *stereochemistry=None*)
Create a new chemical bond.

Methods

__init__(atom1, atom2, bond_order, is_aromatic)	Create a new chemical bond.
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(molecule, d)	Create a Bond from a dict representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
to_bson()	Return a BSON serialized representation.
to_dict()	Return a dict representation of the bond.
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_pickle()	Return a pickle serialized representation.
to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

Attributes

atom1	
atom1_index	
atom2	
atom2_index	
atoms	
bond_order	
fractional_bond_order	
is_aromatic	
molecule	
molecule_bond_index	The index of this Bond within the the list of bonds in Molecules.
stereochemistry	

openforcefield.topology.VirtualSite

class openforcefield.topology.**VirtualSite**(*atoms*, *charge_increments=None*, *epsilon=None*, *sigma=None*, *rmin_half=None*, *name=None*)

A particle representing a virtual site whose position is defined in terms of Atom positions.

Note that chemical atoms are represented by the Atom.

Warning: This API is experimental and subject to change.

Attributes

atoms Atoms on whose position this VirtualSite depends.

charge_increments Charges taken from this VirtualSite's atoms and given to the VirtualSite

epsilon The VdW epsilon term of this VirtualSite

molecule The Molecule this atom is part of.

molecule_particle_index The index of this VirtualSite within the the list of particles in the parent Molecule.

molecule_virtual_site_index The index of this VirtualSite within the list of virtual sites within Molecule Note that this can be different from particle_index.

name The name of this VirtualSite

rmin_half The VdW rmin_half term of this VirtualSite

sigma The VdW sigma term of this VirtualSite

type The type of this VirtualSite (returns the class name as string)

Methods

`from_bson`(serialized)	Instantiate an object from a BSON serialized representation.
`from_dict`(vsite_dict)	Create a virtual site from a dict representation.
`from_json`(serialized)	Instantiate an object from a JSON serialized representation.
`from_messagepack`(serialized)	Instantiate an object from a MessagePack serialized representation.
`from_pickle`(serialized)	Instantiate an object from a pickle serialized representation.
`from_toml`(serialized)	Instantiate an object from a TOML serialized representation.
`from_xml`(serialized)	Instantiate an object from an XML serialized representation.
`from_yaml`(serialized)	Instantiate from a YAML serialized representation.
`to_bson`()	Return a BSON serialized representation.
`to_dict`()	Return a dict representation of the virtual site.
`to_json`([indent])	Return a JSON serialized representation.
`to_messagepack`()	Return a MessagePack representation.
`to_pickle`()	Return a pickle serialized representation.
`to_toml`()	Return a TOML serialized representation.
`to_xml`([indent])	Return an XML representation.
`to_yaml`()	Return a YAML serialized representation.

__init__(*atoms*, *charge_increments=None*, *epsilon=None*, *sigma=None*, *rmin_half=None*, *name=None*)

Base class for VirtualSites

Parameters

atoms [list of Atom of shape [N]] atoms[index] is the corresponding Atom for weights[index]

charge_increments [list of floats of shape [N], optional, default=None] The amount of charge to remove from the VirtualSite's atoms and put in the VirtualSite. Indexing in this list should match the ordering in the atoms list. Default is None.

sigma [float, default=None] Sigma term for VdW properties of virtual site. Default is None.

epsilon [float] Epsilon term for VdW properties of virtual site. Default is None.

rmin_half [float] Rmin_half term for VdW properties of virtual site. Default is None.

name [string or None, default=None] The name of this virtual site. Default is None.

virtual_site_type [str] Virtual site type.

name [str or None, default=None] The name of this virtual site. Default is None

Methods

`__init__`(atoms[, charge_increments, ...])	Base class for VirtualSites
`from_bson`(serialized)	Instantiate an object from a BSON serialized representation.
`from_dict`(vsite_dict)	Create a virtual site from a dict representation.
`from_json`(serialized)	Instantiate an object from a JSON serialized representation.
`from_messagepack`(serialized)	Instantiate an object from a MessagePack serialized representation.
`from_pickle`(serialized)	Instantiate an object from a pickle serialized representation.
`from_toml`(serialized)	Instantiate an object from a TOML serialized representation.
`from_xml`(serialized)	Instantiate an object from an XML serialized representation.
`from_yaml`(serialized)	Instantiate from a YAML serialized representation.
`to_bson`()	Return a BSON serialized representation.
`to_dict`()	Return a dict representation of the virtual site.
`to_json`([indent])	Return a JSON serialized representation.
`to_messagepack`()	Return a MessagePack representation.
`to_pickle`()	Return a pickle serialized representation.
`to_toml`()	Return a TOML serialized representation.
`to_xml`([indent])	Return an XML representation.
`to_yaml`()	Return a YAML serialized representation.

Attributes

`atoms`	Atoms on whose position this VirtualSite depends.
`charge_increments`	Charges taken from this VirtualSite's atoms and given to the VirtualSite
`epsilon`	The VdW epsilon term of this VirtualSite
`molecule`	The `Molecule` this atom is part of.
`molecule_particle_index`	The index of this VirtualSite within the the list of particles in the parent `Molecule`.
`molecule_virtual_site_index`	The index of this VirtualSite within the list of virtual sites within `Molecule` Note that this can be different from `particle_index`.
`name`	The name of this VirtualSite
`rmin_half`	The VdW rmin_half term of this VirtualSite
`sigma`	The VdW sigma term of this VirtualSite
`type`	The type of this VirtualSite (returns the class name as string)

openforcefield.topology.TopologyAtom

class openforcefield.topology.**TopologyAtom**(*atom*, *topology_molecule*)

A TopologyAtom is a lightweight data structure that represents a single openforcefield.topology.molecule.Atom in a Topology. A TopologyAtom consists of two references – One to its fully detailed "atom", an openforcefield.topology.molecule.Atom, and another to its parent "topology_molecule", which occupies a spot in the parent Topology's TopologyMolecule list.

As some systems can be very large, there is no always-existing representation of a TopologyAtom. They are created on demand as the user requests them.

> **Warning:** This API is experimental and subject to change.

Attributes

atom Get the reference Atom for this TopologyAtom.

atomic_number Get the atomic number of this atom

molecule Get the reference Molecule that this TopologyAtom belongs to.

topology_atom_index Get the index of this atom in its parent Topology.

topology_bonds Get the TopologyBonds connected to this TopologyAtom.

topology_molecule Get the TopologyMolecule that this TopologyAtom belongs to.

topology_particle_index Get the index of this particle in its parent Topology.

Methods

from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(d)	Static constructor from dictionary representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
to_bson()	Return a BSON serialized representation.
to_dict()	Convert to dictionary representation.
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_pickle()	Return a pickle serialized representation.
to_toml()	Return a TOML serialized representation.

continues on next page

Table 27 – continued from previous page

to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

__init__(*atom*, *topology_molecule*)
Create a new TopologyAtom.

Parameters

atom [An openforcefield.topology.molecule.Atom] The reference atom

topology_molecule [An openforcefield.topology.TopologyMolecule] The TopologyMolecule that this TopologyAtom belongs to

Methods

__init__(atom, topology_molecule)	Create a new TopologyAtom.
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(d)	Static constructor from dictionary representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
to_bson()	Return a BSON serialized representation.
to_dict()	Convert to dictionary representation.
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_pickle()	Return a pickle serialized representation.
to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

Attributes

atom	Get the reference Atom for this TopologyAtom.
atomic_number	Get the atomic number of this atom
molecule	Get the reference Molecule that this TopologyAtom belongs to.
topology_atom_index	Get the index of this atom in its parent Topology.

continues on next page

Table 29 – continued from previous page

`topology_bonds`	Get the TopologyBonds connected to this TopologyAtom.
`topology_molecule`	Get the TopologyMolecule that this TopologyAtom belongs to.
`topology_particle_index`	Get the index of this particle in its parent Topology.

openforcefield.topology.TopologyBond

class openforcefield.topology.**TopologyBond**(*bond*, *topology_molecule*)

A TopologyBond is a lightweight data structure that represents a single openforcefield.topology.molecule.Bond in a Topology. A TopologyBond consists of two references – One to its fully detailed "bond", an openforcefield.topology.molecule.Bond, and another to its parent "topology_molecule", which occupies a spot in the parent Topology's TopologyMolecule list.

As some systems can be very large, there is no always-existing representation of a TopologyBond. They are created on demand as the user requests them.

> **Warning:** This API is experimental and subject to change.

Attributes

atoms Get the TopologyAtoms connected to this TopologyBond.

bond Get the reference Bond for this TopologyBond.

bond_order Get the order of this TopologyBond.

molecule Get the reference Molecule that this TopologyBond belongs to.

topology_bond_index Get the index of this bond in its parent Topology.

topology_molecule Get the TopologyMolecule that this TopologyBond belongs to.

Methods

`from_bson`(serialized)	Instantiate an object from a BSON serialized representation.
`from_dict`(d)	Static constructor from dictionary representation.
`from_json`(serialized)	Instantiate an object from a JSON serialized representation.
`from_messagepack`(serialized)	Instantiate an object from a MessagePack serialized representation.
`from_pickle`(serialized)	Instantiate an object from a pickle serialized representation.
`from_toml`(serialized)	Instantiate an object from a TOML serialized representation.
`from_xml`(serialized)	Instantiate an object from an XML serialized representation.

continues on next page

Table 30 – continued from previous page

from_yaml(serialized)	Instantiate from a YAML serialized representation.
to_bson()	Return a BSON serialized representation.
to_dict()	Convert to dictionary representation.
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_pickle()	Return a pickle serialized representation.
to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

__init__(*bond*, *topology_molecule*)

Parameters

bond [An openforcefield.topology.molecule.Bond] The reference bond.

topology_molecule [An openforcefield.topology.TopologyMolecule] The TopologyMolecule that this TopologyBond belongs to.

Methods

__init__(bond, topology_molecule)	**Parameters**
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(d)	Static constructor from dictionary representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
to_bson()	Return a BSON serialized representation.
to_dict()	Convert to dictionary representation.
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_pickle()	Return a pickle serialized representation.
to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

Attributes

atoms	Get the TopologyAtoms connected to this TopologyBond.
bond	Get the reference Bond for this TopologyBond.
bond_order	Get the order of this TopologyBond.
molecule	Get the reference Molecule that this TopologyBond belongs to.
topology_bond_index	Get the index of this bond in its parent Topology.
topology_molecule	Get the TopologyMolecule that this TopologyBond belongs to.

openforcefield.topology.TopologyVirtualSite

class openforcefield.topology.**TopologyVirtualSite**(*virtual_site*, *topology_molecule*)

A TopologyVirtualSite is a lightweight data structure that represents a single openforcefield.topology.molecule.VirtualSite in a Topology. A TopologyVirtualSite consists of two references – One to its fully detailed "VirtualSite", an openforcefield.topology.molecule.VirtualSite, and another to its parent "topology_molecule", which occupies a spot in the parent Topology's TopologyMolecule list.

As some systems can be very large, there is no always-existing representation of a TopologyVirtualSite. They are created on demand as the user requests them.

> **Warning:** This API is experimental and subject to change.

Attributes

atoms Get the TopologyAtoms involved in this TopologyVirtualSite.

molecule Get the reference Molecule that this TopologyVirtualSite belongs to.

topology_molecule Get the TopologyMolecule that this TopologyVirtualSite belongs to.

topology_particle_index Get the index of this particle in its parent Topology.

topology_virtual_site_index Get the index of this virtual site in its parent Topology.

type Get the type of this virtual site

virtual_site Get the reference VirtualSite for this TopologyVirtualSite.

Methods

atom(index)	Get the atom at a specific index in this TopologyVirtualSite
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(d)	Static constructor from dictionary representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.

continues on next page

Table 33 – continued from previous page

from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
to_bson()	Return a BSON serialized representation.
to_dict()	Convert to dictionary representation.
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_pickle()	Return a pickle serialized representation.
to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

__init__(*virtual_site*, *topology_molecule*)

Parameters

virtual_site [An openforcefield.topology.molecule.VirtualSite] The reference virtual site

topology_molecule [An openforcefield.topology.TopologyMolecule] The TopologyMolecule that this TopologyVirtualSite belongs to

Methods

__init__(virtual_site, topology_molecule)	**Parameters**
atom(index)	Get the atom at a specific index in this TopologyVirtualSite
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(d)	Static constructor from dictionary representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_xml(serialized)	Instantiate an object from an XML serialized representation.

continues on next page

Table 34 – continued from previous page

from_yaml(serialized)	Instantiate from a YAML serialized representation.
to_bson()	Return a BSON serialized representation.
to_dict()	Convert to dictionary representation.
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_pickle()	Return a pickle serialized representation.
to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

Attributes

atoms	Get the TopologyAtoms involved in this TopologyVirtualSite.
molecule	Get the reference Molecule that this TopologyVirtualSite belongs to.
topology_molecule	Get the TopologyMolecule that this TopologyVirtualSite belongs to.
topology_particle_index	Get the index of this particle in its parent Topology.
topology_virtual_site_index	Get the index of this virtual site in its parent Topology.
type	Get the type of this virtual site
virtual_site	Get the reference VirtualSite for this TopologyVirtualSite.

2.2 Forcefield typing tools

2.2.1 Chemical environments

Tools for representing and operating on chemical environments

> **Warning:** This class is largely redundant with the same one in the Chemper package, and will likely be removed.

ChemicalEnvironment	Chemical environment abstract base class used for validating SMIRKS

openforcefield.typing.chemistry.ChemicalEnvironment

class openforcefield.typing.chemistry.**ChemicalEnvironment**(*smirks=None*, *label=None*, *validate_parsable=True*, *validate_valence_type=True*, *toolkit_registry=None*)

Chemical environment abstract base class used for validating SMIRKS

Methods

`get_type`([toolkit_registry])	Return the valence type implied by the connectivity of the bound atoms in this ChemicalEnvironment.
`validate`([validate_valence_type, ...])	Returns True if the underlying smirks is the correct valence type, False otherwise.
`validate_smirks`(smirks[, validate_parsable, ...])	Check the provided SMIRKS string is valid, and if requested, tags atoms appropriate to the specified valence type.

__init__(*smirks=None*, *label=None*, *validate_parsable=True*, *validate_valence_type=True*, *toolkit_registry=None*)

Initialize a chemical environment abstract base class.

smirks = string, optional if smirks is not None, a chemical environment is built from the provided SMIRKS string

label = anything, optional intended to be used to label this chemical environment could be a string, int, or float, or anything

validate_parsable: bool, optional, default=True If specified, ensure the provided smirks is parsable

validate_valence_type [bool, optional, default=True] If specified, ensure the tagged atoms are appropriate to the specified valence type

toolkit_registry = string or ToolkitWrapper or ToolkitRegistry. Default = None Either a ToolkitRegistry, ToolkitWrapper, or the strings 'openeye' or 'rdkit', indicating the backend to use for validating the correct connectivity of the SMIRKS during initialization. If None, this function will use the GLOBAL_TOOLKIT_REGISTRY

Raises

SMIRKSParsingError if smirks was unparsable

SMIRKSMismatchError if smirks did not have expected connectivity between tagged atoms and validate_valence_type=True

Methods

`__init__`([smirks, label, validate_parsable, ...])	Initialize a chemical environment abstract base class.
`get_type`([toolkit_registry])	Return the valence type implied by the connectivity of the bound atoms in this ChemicalEnvironment.
`validate`([validate_valence_type, ...])	Returns True if the underlying smirks is the correct valence type, False otherwise.
`validate_smirks`(smirks[, validate_parsable, ...])	Check the provided SMIRKS string is valid, and if requested, tags atoms appropriate to the specified valence type.

2.2.2 Forcefield typing engines

Engines for applying parameters to chemical systems

The SMIRks-Native Open Force Field (SMIRNOFF)

A reference implementation of the SMIRNOFF specification for parameterizing biomolecular systems

ForceField

The `ForceField` class is a primary part of the top-level toolkit API. `ForceField` objects are initialized from SMIRNOFF data sources (e.g. an `OFFXML` file). For a basic example of system creation using a `ForceField`, see `examples/SMIRNOFF_simulation`.

`ForceField`	A factory that assigns SMIRNOFF parameters to a molecular system

openforcefield.typing.engines.smirnoff.forcefield.ForceField

class openforcefield.typing.engines.smirnoff.forcefield.**ForceField**(**sources*, *parameter_handler_classes=None*, *parameter_io_handler_classes=None*, *disable_version_check=False*, *allow_cosmetic_attributes=False*, *load_plugins=False*)

A factory that assigns SMIRNOFF parameters to a molecular system

`ForceField` is a factory that constructs an OpenMM `simtk.openmm.System` object from a `openforcefield.topology.Topology` object defining a (bio)molecular system containing one or more molecules.

When a `ForceField` object is created from one or more specified SMIRNOFF serialized representations, all `ParameterHandler` subclasses currently imported are identified and registered to handle different sections of the SMIRNOFF force field definition file(s).

All `ParameterIOHandler` subclasses currently imported are identified and registered to handle different serialization formats (such as XML).

The force field definition is processed by these handlers to populate the `ForceField` object model data structures that can easily be manipulated via the API:

Processing a `Topology` object defining a chemical system will then call all :class`ParameterHandler` objects in an order guaranteed to satisfy the declared processing order constraints of each :class`ParameterHandler`.

Examples

Create a new ForceField containing the smirnoff99Frosst parameter set:

```
>>> from openforcefield.typing.engines.smirnoff import ForceField
>>> forcefield = ForceField('test_forcefields/smirnoff99Frosst.offxml')
```

Create an OpenMM system from a `openforcefield.topology.Topology` object:

```
>>> from openforcefield.topology import Molecule, Topology
>>> ethanol = Molecule.from_smiles('CCO')
>>> topology = Topology.from_molecules(molecules=[ethanol])
>>> system = forcefield.create_openmm_system(topology)
```

Modify the long-range electrostatics method:

```
>>> forcefield.get_parameter_handler('Electrostatics').method = 'PME'
```

Inspect the first few vdW parameters:

```
>>> low_precedence_parameters = forcefield.get_parameter_handler('vdW').parameters[0:3]
```

Retrieve the vdW parameters by SMIRKS string and manipulate it:

```
>>> parameter = forcefield.get_parameter_handler('vdW').parameters['[#1:1]-[#7]']
>>> parameter.rmin_half += 0.1 * unit.angstroms
>>> parameter.epsilon *= 1.02
```

Make a child vdW type more specific (checking modified SMIRKS for validity):

```
>>> forcefield.get_parameter_handler('vdW').parameters[-1].smirks += '~[#53]'
```

Warning: While we check whether the modified SMIRKS is still valid and has the appropriate valence type, we currently don't check whether the typing remains hierarchical, which could result in some types no longer being assignable because more general types now come *below* them and preferentially match.

Delete a parameter:

```
>>> del forcefield.get_parameter_handler('vdW').parameters['[#1:1]-[#6X4]']
```

Insert a parameter at a specific point in the parameter tree:

```
>>> from openforcefield.typing.engines.smirnoff import vdWHandler
>>> new_parameter = vdWHandler.vdWType(smirks='[*:1]', epsilon=0.0157*unit.kilocalories_per_mole,␣
↪rmin_half=0.6000*unit.angstroms)
>>> forcefield.get_parameter_handler('vdW').parameters.insert(0, new_parameter)
```

Warning: We currently don't check whether removing a parameter could accidentally remove the root type, so it's possible to no longer type all molecules this way.

Attributes

parameters [dict of str] `parameters[tagname]` is the instantiated `ParameterHandler` class that handles parameters associated with the force `tagname`. This is the primary means of retrieving and modifying parameters, such as `parameters['vdW'][0].sigma *= 1.1`

parameter_object_handlers [dict of str] Registered list of `ParameterHandler` classes that will handle different forcefield tags to create the parameter object model. `parameter_object_handlers[tagname]` is the `ParameterHandler` that will be instantiated to process the force field definition section `tagname`. `ParameterHandler` classes are registered when the ForceField object is created, but can be manipulated afterwards.

parameter_io_handlers [dict of str] Registered list of `ParameterIOHandler` classes that will handle serializing/deserializing the parameter object model to string or file representations, such as XML. `parameter_io_handlers[iotype]` is the `ParameterHandler` that will be instantiated to process the serialization scheme `iotype`. `ParameterIOHandler` classes are registered when the ForceField object is created, but can be manipulated afterwards.

Methods

`create_openmm_system`(topology, **kwargs)	Create an OpenMM System representing the interactions for the specified Topology with the current force field
`create_parmed_structure`(topology, positions, ...)	Create a ParmEd Structure object representing the interactions for the specified Topology with the current force field
`get_parameter_handler`(tagname[, ...])	Retrieve the parameter handlers associated with the provided tagname.
`get_parameter_io_handler`(io_format)	Retrieve the parameter handlers associated with the provided tagname.
`label_molecules`(topology)	Return labels for a list of molecules corresponding to parameters from this force field.
`parse_smirnoff_from_source`(source)	Reads a SMIRNOFF data structure from a source, which can be one of many types.
`parse_sources`(sources[, ...])	Parse a SMIRNOFF force field definition.
`register_parameter_handler`(parameter_handler)	Register a new ParameterHandler for a specific tag, making it available for lookup in the ForceField.

continues on next page

Table 40 – continued from previous page

`register_parameter_io_handler(...)`	Register a new ParameterIOHandler, making it available for lookup in the ForceField.
`to_file`(filename[, io_format, ...])	Write this Forcefield and all its associated parameters to a string in a given format which complies with the SMIRNOFF spec.
`to_string`([io_format, ...])	Write this Forcefield and all its associated parameters to a string in a given format which complies with the SMIRNOFF spec.

__init__(**sources*, *parameter_handler_classes=None*, *parameter_io_handler_classes=None*, *disable_version_check=False*, *allow_cosmetic_attributes=False*, *load_plugins=False*)

Create a new `ForceField` object from one or more SMIRNOFF parameter definition files.

Parameters

sources [string or file-like object or open file handle or URL (or iterable of these)] A list of files defining the SMIRNOFF force field to be loaded. Currently, only the SMIRNOFF XML format is supported. Each entry may be an absolute file path, a path relative to the current working directory, a path relative to this module's data subdirectory (for built in force fields), or an open file-like object with a `read()` method from which the forcefield XML data can be loaded. If multiple files are specified, any top-level tags that are repeated will be merged if they are compatible, with files appearing later in the sequence resulting in parameters that have higher precedence. Support for multiple files is primarily intended to allow solvent parameters to be specified by listing them last in the sequence.

parameter_handler_classes [iterable of ParameterHandler classes, optional, default=None] If not None, the specified set of ParameterHandler classes will be instantiated to create the parameter object model. By default, all imported subclasses of ParameterHandler are automatically registered.

parameter_io_handler_classes [iterable of ParameterIOHandler classes] If not None, the specified set of ParameterIOHandler classes will be used to parse/generate serialized parameter sets. By default, all imported subclasses of ParameterIOHandler are automatically registered.

disable_version_check [bool, optional, default=False] If True, will disable checks against the current highest supported forcefield version. This option is primarily intended for forcefield development.

allow_cosmetic_attributes [bool, optional. Default = False] Whether to retain non-spec kwargs from data sources.

load_plugins: bool, optional. Default = False Whether to load `ParameterHandler` classes which have been registered by installed plugins.

Examples

Load one SMIRNOFF parameter set in XML format (searching the package data directory by default, which includes some standard parameter sets):

```
>>> forcefield = ForceField('test_forcefields/smirnoff99Frosst.offxml')
```

Load multiple SMIRNOFF parameter sets:

forcefield = ForceField('test_forcefields/smirnoff99Frosst.offxml', 'test_forcefields/tip3p.offxml')

Load a parameter set from a string:

```
>>> offxml = '<SMIRNOFF version="0.2" aromaticity_model="OEAroModel_MDL"/>'
>>> forcefield = ForceField(offxml)
```

Methods

`__init__`(*sources[, ...])	Create a new `ForceField` object from one or more SMIRNOFF parameter definition files.
`create_openmm_system`(topology, **kwargs)	Create an OpenMM System representing the interactions for the specified Topology with the current force field
`create_parmed_structure`(topology, positions, ...)	Create a ParmEd Structure object representing the interactions for the specified Topology with the current force field
`get_parameter_handler`(tagname[, ...])	Retrieve the parameter handlers associated with the provided tagname.
`get_parameter_io_handler`(io_format)	Retrieve the parameter handlers associated with the provided tagname.
`label_molecules`(topology)	Return labels for a list of molecules corresponding to parameters from this force field.
`parse_smirnoff_from_source`(source)	Reads a SMIRNOFF data structure from a source, which can be one of many types.
`parse_sources`(sources[, ...])	Parse a SMIRNOFF force field definition.
`register_parameter_handler`(parameter_handler)	Register a new ParameterHandler for a specific tag, making it available for lookup in the ForceField.
`register_parameter_io_handler`(...)	Register a new ParameterIOHandler, making it available for lookup in the ForceField.
`to_file`(filename[, io_format, ...])	Write this Forcefield and all its associated parameters to a string in a given format which complies with the SMIRNOFF spec.
`to_string`([io_format, ...])	Write this Forcefield and all its associated parameters to a string in a given format which complies with the SMIRNOFF spec.

Attributes

author	Returns the author data for this ForceField object.
date	Returns the date data for this ForceField object.

Parameter Type

ParameterType objects are representations of individual SMIRKS-based SMIRNOFF parameters. These are usually initialized during ForceField creation, and can be inspected and modified by users via the Python API. For more information, see examples/forcefield_modification.

ParameterType	Base class for SMIRNOFF parameter types.
BondHandler.BondType	A SMIRNOFF bond type
AngleHandler.AngleType	A SMIRNOFF angle type.
ProperTorsionHandler.ProperTorsionType	A SMIRNOFF torsion type for proper torsions.
ImproperTorsionHandler.ImproperTorsionType	A SMIRNOFF torsion type for improper torsions.
vdWHandler.vdWType	A SMIRNOFF vdWForce type.
LibraryChargeHandler.LibraryChargeType	A SMIRNOFF Library Charge type.
GBSAHandler.GBSAType	A SMIRNOFF GBSA type.

openforcefield.typing.engines.smirnoff.parameters.ParameterType

class openforcefield.typing.engines.smirnoff.parameters.**ParameterType**(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

Base class for SMIRNOFF parameter types.

This base class provides utilities to create new parameter types. See the below for examples of how to do this.

Warning: This API is experimental and subject to change.

See also:

ParameterAttribute

IndexedParameterAttribute

Examples

This class allows to define new parameter types by just listing its attributes. In the example below, _VALENCE_TYPE AND _ELEMENT_NAME are used for the validation of the SMIRKS pattern associated to the parameter and the automatic serialization/deserialization into a dict.

```
>>> class MyBondParameter(ParameterType):
...     _VALENCE_TYPE = 'Bond'
...     _ELEMENT_NAME = 'Bond'
...     length = ParameterAttribute(unit=unit.angstrom)
```

(continues on next page)

(continued from previous page)

```
...     k = ParameterAttribute(unit=unit.kilocalorie_per_mole / unit.angstrom**2)
...
```

The parameter automatically inherits the required smirks attribute from `ParameterType`. Associating a `unit` to a `ParameterAttribute` cause the attribute to accept only values in compatible units and to parse string expressions.

```
>>> my_par = MyBondParameter(
...     smirks='[*:1]-[*:2]',
...     length='1.01 * angstrom',
...     k=5 * unit.kilocalorie_per_mole / unit.angstrom**2
... )
>>> my_par.length
Quantity(value=1.01, unit=angstrom)
>>> my_par.k = 3.0 * unit.gram
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: k=3.0 g should have units of kilocalorie/
↪(angstrom**2*mole)
```

Each attribute can be made optional by specifying a default value, and you can attach a converter function by passing a callable as an argument or through the decorator syntax.

```
>>> class MyParameterType(ParameterType):
...     _VALENCE_TYPE = 'Atom'
...     _ELEMENT_NAME = 'Atom'
...
...     attr_optional = ParameterAttribute(default=2)
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to floats
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameterType(smirks='[*:1]', attr_all_to_float='3.0', attr_int_to_float=1)
>>> my_par.attr_optional
2
>>> my_par.attr_all_to_float
3.0
>>> my_par.attr_int_to_float
1.0
```

The float() function can convert strings to integers, but our custom converter forbids it

```
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_int_to_float = '4.0'
Traceback (most recent call last):
...
TypeError: Cannot convert '4.0' to float
```

Parameter attributes that can be indexed can be handled with the `IndexedParameterAttribute`. These support unit validation and converters exactly as ``ParameterAttribute``s, but the validation/conversion is performed for each indexed attribute.

```
>>> class MyTorsionType(ParameterType):
...     _VALENCE_TYPE = 'ProperTorsion'
...     _ELEMENT_NAME = 'Proper'
...     periodicity = IndexedParameterAttribute(converter=int)
...     k = IndexedParameterAttribute(unit=unit.kilocalorie_per_mole)
...
>>> my_par = MyTorsionType(
...     smirks='[*:1]-[*:2]-[*:3]-[*:4]',
...     periodicity1=2,
...     k1=5 * unit.kilocalorie_per_mole,
...     periodicity2='3',
...     k2=6 * unit.kilocalorie_per_mole,
... )
>>> my_par.periodicity
[2, 3]
```

Indexed attributes, can be accessed both as a list or as their indexed parameter name.

```
>>> my_par.periodicity2 = 6
>>> my_par.periodicity[0] = 1
>>> my_par.periodicity
[1, 6]
```

Attributes

smirks [str] The SMIRKS pattern that this parameter matches.

id [str or None] An optional identifier for the parameter.

parent_id [str or None] Optionally, the identifier of the parameter of which this parameter is a specialization.

Methods

`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`to_dict`([discard_cosmetic_attributes])	Convert this object to dict format.

__init__(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)
Create a ParameterType.

Parameters

smirks [str] The SMIRKS match for the provided parameter type.

allow_cosmetic_attributes [bool optional. Default = False] Whether to permit non-spec kwargs ("cosmetic attributes"). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

Methods

`__init__`(smirks[, allow_cosmetic_attributes])	Create a ParameterType.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`to_dict`([discard_cosmetic_attributes])	Convert this object to dict format.

Attributes

`id`	A descriptor for `ParameterType` attributes.
`parent_id`	A descriptor for `ParameterType` attributes.
`smirks`	A descriptor for `ParameterType` attributes.

openforcefield.typing.engines.smirnoff.parameters.BondHandler.BondType

openforcefield.typing.engines.smirnoff.parameters.AngleHandler.AngleType

openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler.ProperTorsionType

openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler.ImproperTorsionType

openforcefield.typing.engines.smirnoff.parameters.vdWHandler.vdWType

openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandler.LibraryChargeType

openforcefield.typing.engines.smirnoff.parameters.GBSAHandler.GBSAType

Parameter Handlers

Each `ForceField` primarily consists of several `ParameterHandler` objects, which each contain the machinery to add one energy component to a system. During system creation, each `ParameterHandler` registered to a `ForceField` has its `assign_parameters()` function called..

`ParameterList`	Parameter list that also supports accessing items by SMARTS string.
`ParameterHandler`	Base class for parameter handlers.
`BondHandler`	Handle SMIRNOFF `<Bonds>` tags
`AngleHandler`	Handle SMIRNOFF `<AngleForce>` tags
`ProperTorsionHandler`	Handle SMIRNOFF `<ProperTorsionForce>` tags
`ImproperTorsionHandler`	Handle SMIRNOFF `<ImproperTorsionForce>` tags
`vdWHandler`	Handle SMIRNOFF `<vdW>` tags
`ElectrostaticsHandler`	Handles SMIRNOFF `<Electrostatics>` tags.
`LibraryChargeHandler`	Handle SMIRNOFF `<LibraryCharges>` tags

continues on next page

Table 47 – continued from previous page

ToolkitAM1BCCHandler	Handle SMIRNOFF <ToolkitAM1BCC> tags
GBSAHandler	Handle SMIRNOFF <GBSA> tags

openforcefield.typing.engines.smirnoff.parameters.ParameterList

class openforcefield.typing.engines.smirnoff.parameters.**ParameterList**(*input_parameter_list=None*)

Parameter list that also supports accessing items by SMARTS string.

> **Warning:** This API is experimental and subject to change.

Methods

append(parameter)	Add a ParameterType object to the end of the ParameterList
clear(/)	Remove all items from list.
copy(/)	Return a shallow copy of the list.
count(value, /)	Return number of occurrences of value.
extend(other)	Add a ParameterList object to the end of the ParameterList
index(item)	Get the numerical index of a ParameterType object or SMIRKS in this ParameterList.
insert(index, parameter)	Add a ParameterType object as if this were a list
pop([index])	Remove and return item at index (default last).
remove(value, /)	Remove first occurrence of value.
reverse(/)	Reverse *IN PLACE*.
sort(*[, key, reverse])	Stable sort *IN PLACE*.
to_list([discard_cosmetic_attributes])	Render this ParameterList to a normal list, serializing each ParameterType object in it to dict.

__init__(*input_parameter_list=None*)

Initialize a new ParameterList, optionally providing a list of ParameterType objects to initially populate it.

Parameters

input_parameter_list: list[ParameterType], default=None A pre-existing list of ParameterType-based objects. If None, this ParameterList will be initialized empty.

Methods

__init__([input_parameter_list])	Initialize a new ParameterList, optionally providing a list of ParameterType objects to initially populate it.
append(parameter)	Add a ParameterType object to the end of the ParameterList
clear(/)	Remove all items from list.
copy(/)	Return a shallow copy of the list.

continues on next page

Table 49 – continued from previous page

count(value, /)	Return number of occurrences of value.
extend(other)	Add a ParameterList object to the end of the ParameterList
index(item)	Get the numerical index of a ParameterType object or SMIRKS in this ParameterList.
insert(index, parameter)	Add a ParameterType object as if this were a list
pop([index])	Remove and return item at index (default last).
remove(value, /)	Remove first occurrence of value.
reverse(/)	Reverse *IN PLACE*.
sort(*[, key, reverse])	Stable sort *IN PLACE*.
to_list([discard_cosmetic_attributes])	Render this ParameterList to a normal list, serializing each ParameterType object in it to dict.

openforcefield.typing.engines.smirnoff.parameters.ParameterHandler

class openforcefield.typing.engines.smirnoff.parameters.**ParameterHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Base class for parameter handlers.

Parameter handlers are configured with some global parameters for a given section. They may also contain a ParameterList populated with ParameterType objects if they are responsible for assigning SMIRKS-based parameters.

Warning: This API is experimental and subject to change.

Attributes

known_kwargs List of kwargs that can be parsed by the function.

parameters The ParameterList that holds this ParameterHandler's parameter objects

version A descriptor for ParameterType attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have None as a default value, required attributes have the default set to the special type UNDEFINED.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

Methods

add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity)	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
postprocess_system(topology, system, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)
Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

allow_cosmetic_attributes [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

skip_version_check: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.

****kwargs** [dict] The dict representation of the SMIRNOFF data source

Methods

`__init__`([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`add_parameter`(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
`assign_parameters`(topology, system)	Assign parameters for the given Topology to the specified System object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`check_handler_compatibility`(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`find_matches`(entity)	Find the elements of the topology/molecule matched by a parameter type.
`get_parameter`(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
`postprocess_system`(topology, system, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
`to_dict`([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

`known_kwargs`	List of kwargs that can be parsed by the function.
`parameters`	The ParameterList that holds this ParameterHandler's parameter objects
`version`	A descriptor for `ParameterType` attributes.

openforcefield.typing.engines.smirnoff.parameters.BondHandler

class openforcefield.typing.engines.smirnoff.parameters.**BondHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF `<Bonds>` tags

> **Warning:** This API is experimental and subject to change.

Attributes

fractional_bondorder_interpolation A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

fractional_bondorder_method A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

known_kwargs List of kwargs that can be parsed by the function.

parameters The ParameterList that holds this ParameterHandler's parameter objects

potential A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

version A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
```

(continues on next page)

(continued from previous page)

```
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

Methods

BondType(smirks[, allow_cosmetic_attributes])	A SMIRNOFF bond type
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity)	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
postprocess_system(topology, system, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

create_force	

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)
Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

allow_cosmetic_attributes [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

skip_version_check: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.

****kwargs** [dict] The dict representation of the SMIRNOFF data source

Methods

`__init__`([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`add_parameter`(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
`assign_parameters`(topology, system)	Assign parameters for the given Topology to the specified System object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`check_handler_compatibility`(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
`create_force`(system, topology, **kwargs)	
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`find_matches`(entity)	Find the elements of the topology/molecule matched by a parameter type.
`get_parameter`(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
`postprocess_system`(topology, system, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
`to_dict`([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

fractional_bondorder_interpolation	A descriptor for ParameterType attributes.
fractional_bondorder_method	A descriptor for ParameterType attributes.
known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects
potential	A descriptor for ParameterType attributes.
version	A descriptor for ParameterType attributes.

openforcefield.typing.engines.smirnoff.parameters.AngleHandler

class openforcefield.typing.engines.smirnoff.parameters.**AngleHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF <AngleForce> tags

> **Warning:** This API is experimental and subject to change.

Attributes

known_kwargs List of kwargs that can be parsed by the function.

parameters The ParameterList that holds this ParameterHandler's parameter objects

potential A descriptor for ParameterType attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have None as a default value, required attributes have the default set to the special type UNDEFINED.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

version A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

Methods

AngleType(smirks[, allow_cosmetic_attributes])	A SMIRNOFF angle type.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.

continues on next page

Table 56 – continued from previous page

assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity)	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
postprocess_system(topology, system, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

create_force

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)
Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

allow_cosmetic_attributes [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

skip_version_check: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.

****kwargs** [dict] The dict representation of the SMIRNOFF data source

Methods

__init__([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.

continues on next page

Table 57 – continued from previous page

create_force(system, topology, **kwargs)	
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity)	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
postprocess_system(topology, system, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects
potential	A descriptor for ParameterType attributes.
version	A descriptor for ParameterType attributes.

openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler

class openforcefield.typing.engines.smirnoff.parameters.**ProperTorsionHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF <ProperTorsionForce> tags

> **Warning:** This API is experimental and subject to change.

Attributes

default_idivf A descriptor for ParameterType attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have None as a default value, required attributes have the default set to the special type UNDEFINED.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed

into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
```

(continues on next page)

(continued from previous page)

```
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

fractional_bondorder_interpolation A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

fractional_bondorder_method A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have None as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

known_kwargs List of kwargs that can be parsed by the function.

parameters The ParameterList that holds this ParameterHandler's parameter objects

potential A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
```

(continues on next page)

(continued from previous page)

```
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

version A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

Methods

ProperTorsionType(smirks[, ...])	A SMIRNOFF torsion type for proper torsions.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.

continues on next page

Table 59 – continued from previous page

assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity)	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
postprocess_system(topology, system, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

assign_partial_bond_orders_from_molecules	
check_partial_bond_orders_from_molecules_duplicates	
create_force	

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)
Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

allow_cosmetic_attributes [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

skip_version_check: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.

****kwargs** [dict] The dict representation of the SMIRNOFF data source

Methods

__init__([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
assign_partial_bond_orders_from_molecules(...)	
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.

continues on next page

Table 60 – continued from previous page

check_handler_compatibility(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
check_partial_bond_orders_from_molecules_duplicates(pb_mols)	
create_force(system, topology, **kwargs)	
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity)	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
postprocess_system(topology, system, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

default_idivf	A descriptor for ParameterType attributes.
fractional_bondorder_interpolation	A descriptor for ParameterType attributes.
fractional_bondorder_method	A descriptor for ParameterType attributes.
known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects
potential	A descriptor for ParameterType attributes.
version	A descriptor for ParameterType attributes.

openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler

class openforcefield.typing.engines.smirnoff.parameters.**ImproperTorsionHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF <ImproperTorsionForce> tags

> **Warning:** This API is experimental and subject to change.

Attributes

default_idivf A descriptor for ParameterType attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have None as a default value, required attributes have the default set to the special type UNDEFINED.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
```

(continues on next page)

(continued from previous page)

```
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

known_kwargs List of kwargs that can be parsed by the function.

parameters The ParameterList that holds this ParameterHandler's parameter objects

potential A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

version A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

Methods

`ImproperTorsionType(smirks[, ...])`	A SMIRNOFF torsion type for improper torsions.
`add_cosmetic_attribute(attr_name, attr_value)`	Add a cosmetic attribute to this object.
`add_parameter(parameter_kwargs)`	Add a parameter to the forcefield, ensuring all parameters are valid.

continues on next page

Table 62 – continued from previous page

assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity)	Find the improper torsions in the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
postprocess_system(topology, system, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

create_force

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)
Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

allow_cosmetic_attributes [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

skip_version_check: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.

****kwargs** [dict] The dict representation of the SMIRNOFF data source

Methods

__init__([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.

continues on next page

Table 63 – continued from previous page

create_force(system, topology, **kwargs)	
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity)	Find the improper torsions in the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
postprocess_system(topology, system, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

default_idivf	A descriptor for ParameterType attributes.
known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects
potential	A descriptor for ParameterType attributes.
version	A descriptor for ParameterType attributes.

openforcefield.typing.engines.smirnoff.parameters.vdWHandler

class openforcefield.typing.engines.smirnoff.parameters.**vdWHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF <vdW> tags

Warning: This API is experimental and subject to change.

Attributes

combining_rules A descriptor for ParameterType attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have None as a default value, required attributes have the default set to the special type UNDEFINED.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
```

(continues on next page)

(continued from previous page)

```
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

cutoff A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

known_kwargs List of kwargs that can be parsed by the function.

method A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have None as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

parameters The ParameterList that holds this ParameterHandler's parameter objects

potential A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

scale12 A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

scale13 A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

scale14 A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

scale15 A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

switch_width A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
```

(continues on next page)

(continued from previous page)

```
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

version A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

Methods

add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_charges_assigned(ref_mol, topology)	Check whether charges have been assigned for a reference molecule.
check_handler_compatibility(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity)	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
mark_charges_assigned(ref_mol, topology)	Record that charges have been assigned for a reference molecule.
postprocess_system(system, topology, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.
vdWType(**kwargs)	A SMIRNOFF vdWForce type.

create_force

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

allow_cosmetic_attributes [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

skip_version_check: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.

****kwargs** [dict] The dict representation of the SMIRNOFF data source

Methods

`__init__`([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`add_parameter`(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
`assign_parameters`(topology, system)	Assign parameters for the given Topology to the specified System object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`check_charges_assigned`(ref_mol, topology)	Check whether charges have been assigned for a reference molecule.
`check_handler_compatibility`(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
`create_force`(system, topology, **kwargs)	
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`find_matches`(entity)	Find the elements of the topology/molecule matched by a parameter type.
`get_parameter`(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
`mark_charges_assigned`(ref_mol, topology)	Record that charges have been assigned for a reference molecule.
`postprocess_system`(system, topology, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
`to_dict`([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

`combining_rules`	A descriptor for `ParameterType` attributes.
`cutoff`	A descriptor for `ParameterType` attributes.
`known_kwargs`	List of kwargs that can be parsed by the function.
`method`	A descriptor for `ParameterType` attributes.
`parameters`	The ParameterList that holds this ParameterHandler's parameter objects
`potential`	A descriptor for `ParameterType` attributes.
`scale12`	A descriptor for `ParameterType` attributes.
`scale13`	A descriptor for `ParameterType` attributes.
`scale14`	A descriptor for `ParameterType` attributes.
`scale15`	A descriptor for `ParameterType` attributes.
`switch_width`	A descriptor for `ParameterType` attributes.
`version`	A descriptor for `ParameterType` attributes.

openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler

class openforcefield.typing.engines.smirnoff.parameters.**ElectrostaticsHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handles SMIRNOFF <Electrostatics> tags.

Warning: This API is experimental and subject to change.

Attributes

cutoff A descriptor for ParameterType attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have None as a default value, required attributes have the default set to the special type UNDEFINED.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

known_kwargs List of kwargs that can be parsed by the function.

method A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
```

(continues on next page)

(continued from previous page)

```
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

parameters The ParameterList that holds this ParameterHandler's parameter objects

scale12 A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

scale13 A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

scale14 A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
```

(continues on next page)

(continued from previous page)

```
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

scale15 A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

switch_width A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

version A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

Methods

add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_charge_from_molecules(molecule, ...)	Given an input molecule, checks against a list of molecules for an isomorphic match.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_charges_assigned(ref_mol, topology)	Check whether charges have been assigned for a reference molecule.
check_handler_compatibility(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity)	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
mark_charges_assigned(ref_mol, topology)	Record that charges have been assigned for a reference molecule.
postprocess_system(system, topology, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

create_force

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)
Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

allow_cosmetic_attributes [bool, optional. Default = False] Whether to permit

non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

skip_version_check: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.

****kwargs** [dict] The dict representation of the SMIRNOFF data source

Methods

`__init__`([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`add_parameter`(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
`assign_charge_from_molecules`(molecule, ...)	Given an input molecule, checks against a list of molecules for an isomorphic match.
`assign_parameters`(topology, system)	Assign parameters for the given Topology to the specified System object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`check_charges_assigned`(ref_mol, topology)	Check whether charges have been assigned for a reference molecule.
`check_handler_compatibility`(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
`create_force`(system, topology, **kwargs)	
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`find_matches`(entity)	Find the elements of the topology/molecule matched by a parameter type.
`get_parameter`(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
`mark_charges_assigned`(ref_mol, topology)	Record that charges have been assigned for a reference molecule.
`postprocess_system`(system, topology, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
`to_dict`([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

cutoff	A descriptor for ParameterType attributes.
known_kwargs	List of kwargs that can be parsed by the function.
method	A descriptor for ParameterType attributes.
parameters	The ParameterList that holds this ParameterHandler's parameter objects
scale12	A descriptor for ParameterType attributes.
scale13	A descriptor for ParameterType attributes.
scale14	A descriptor for ParameterType attributes.
scale15	A descriptor for ParameterType attributes.
switch_width	A descriptor for ParameterType attributes.
version	A descriptor for ParameterType attributes.

openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandler

class openforcefield.typing.engines.smirnoff.parameters.**LibraryChargeHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF <LibraryCharges> tags

> **Warning:** This API is experimental and subject to change.

Attributes

known_kwargs List of kwargs that can be parsed by the function.

parameters The ParameterList that holds this ParameterHandler's parameter objects

version A descriptor for ParameterType attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have None as a default value, required attributes have the default set to the special type UNDEFINED.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

Methods

`LibraryChargeType`(**kwargs)	A SMIRNOFF Library Charge type.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`add_parameter`(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
`assign_parameters`(topology, system)	Assign parameters for the given Topology to the specified System object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`check_charges_assigned`(ref_mol, topology)	Check whether charges have been assigned for a reference molecule.
`check_handler_compatibility`(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`find_matches`(entity)	Find the elements of the topology/molecule matched by a parameter type.
`get_parameter`(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
`mark_charges_assigned`(ref_mol, topology)	Record that charges have been assigned for a reference molecule.
`postprocess_system`(topology, system, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
`to_dict`([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

create_force

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)
Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

allow_cosmetic_attributes [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

skip_version_check: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.

****kwargs** [dict] The dict representation of the SMIRNOFF data source

Methods

`__init__`([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`add_parameter`(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
`assign_parameters`(topology, system)	Assign parameters for the given Topology to the specified System object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`check_charges_assigned`(ref_mol, topology)	Check whether charges have been assigned for a reference molecule.
`check_handler_compatibility`(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
`create_force`(system, topology, **kwargs)	
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`find_matches`(entity)	Find the elements of the topology/molecule matched by a parameter type.
`get_parameter`(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
`mark_charges_assigned`(ref_mol, topology)	Record that charges have been assigned for a reference molecule.
`postprocess_system`(topology, system, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
`to_dict`([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

`known_kwargs`	List of kwargs that can be parsed by the function.
`parameters`	The ParameterList that holds this ParameterHandler's parameter objects
`version`	A descriptor for `ParameterType` attributes.

openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler

class openforcefield.typing.engines.smirnoff.parameters.**ToolkitAM1BCCHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF <ToolkitAM1BCC> tags

Warning: This API is experimental and subject to change.

Attributes

known_kwargs List of kwargs that can be parsed by the function.

parameters The ParameterList that holds this ParameterHandler's parameter objects

version A descriptor for ParameterType attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have None as a default value, required attributes have the default set to the special type UNDEFINED.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a Quantity.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

Methods

add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_charges_assigned(ref_mol, topology)	Check whether charges have been assigned for a reference molecule.
check_handler_compatibility(other_handler[, ...])	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity)	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
mark_charges_assigned(ref_mol, topology)	Record that charges have been assigned for a reference molecule.
postprocess_system(system, topology, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

create_force

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)
Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

allow_cosmetic_attributes [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

skip_version_check: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.

****kwargs** [dict] The dict representation of the SMIRNOFF data source

Methods

`__init__`([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`add_parameter`(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
`assign_parameters`(topology, system)	Assign parameters for the given Topology to the specified System object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`check_charges_assigned`(ref_mol, topology)	Check whether charges have been assigned for a reference molecule.
`check_handler_compatibility`(other_handler[, ...])	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
`create_force`(system, topology, **kwargs)	
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`find_matches`(entity)	Find the elements of the topology/molecule matched by a parameter type.
`get_parameter`(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
`mark_charges_assigned`(ref_mol, topology)	Record that charges have been assigned for a reference molecule.
`postprocess_system`(system, topology, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
`to_dict`([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

`known_kwargs`	List of kwargs that can be parsed by the function.
`parameters`	The ParameterList that holds this ParameterHandler's parameter objects
`version`	A descriptor for `ParameterType` attributes.

openforcefield.typing.engines.smirnoff.parameters.GBSAHandler

class openforcefield.typing.engines.smirnoff.parameters.**GBSAHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF `<GBSA>` tags

> **Warning:** This API is experimental and subject to change.

Attributes

gb_model A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

known_kwargs List of kwargs that can be parsed by the function.

parameters The ParameterList that holds this ParameterHandler's parameter objects

sa_model A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
```

(continues on next page)

(continued from previous page)

```
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

solute_dielectric A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

solvent_dielectric A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have None as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

solvent_radius A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
```

(continues on next page)

(continued from previous page)

```
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

surface_area_penalty A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

version A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

IndexedParameterAttribute A parameter attribute with multiple terms.

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0␣
↪dimensionless should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
```

(continues on next page)

(continued from previous page)

```
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

Methods

GBSAType(smirks[, allow_cosmetic_attributes])	A SMIRNOFF GBSA type.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity)	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
postprocess_system(topology, system, **kwargs)	Allow the force to perform a final post-processing pass on the System following parameter assignment, if needed.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

create_force	

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)
Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

allow_cosmetic_attributes [bool, optional. Default = False] Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

skip_version_check: bool, optional. Default = False If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.

****kwargs** [dict] The dict representation of the SMIRNOFF data source

Methods

`__init__`([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`add_parameter`(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
`assign_parameters`(topology, system)	Assign parameters for the given Topology to the specified System object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`check_handler_compatibility`(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
`create_force`(system, topology, **kwargs)	
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`find_matches`(entity)	Find the elements of the topology/molecule matched by a parameter type.
`get_parameter`(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
`postprocess_system`(topology, system, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
`to_dict`([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

`gb_model`	A descriptor for `ParameterType` attributes.
`known_kwargs`	List of kwargs that can be parsed by the function.
`parameters`	The ParameterList that holds this ParameterHandler's parameter objects
`sa_model`	A descriptor for `ParameterType` attributes.
`solute_dielectric`	A descriptor for `ParameterType` attributes.

continues on next page

Table 79 – continued from previous page

solvent_dielectric	A descriptor for ParameterType attributes.
solvent_radius	A descriptor for ParameterType attributes.
surface_area_penalty	A descriptor for ParameterType attributes.
version	A descriptor for ParameterType attributes.

Parameter I/O Handlers

ParameterIOHandler objects handle reading and writing of serialzied SMIRNOFF data sources.

ParameterIOHandler	Base class for handling serialization/deserialization of SMIRNOFF ForceField objects
XMLParameterIOHandler	Handles serialization/deserialization of SMIRNOFF ForceField objects from OFFXML format.

openforcefield.typing.engines.smirnoff.io.ParameterIOHandler

class openforcefield.typing.engines.smirnoff.io.**ParameterIOHandler**

Base class for handling serialization/deserialization of SMIRNOFF ForceField objects

Methods

parse_file(file_path)	**Parameters**
parse_string(data)	Parse a SMIRNOFF force field definition in a seriaized format
to_file(file_path, smirnoff_data)	Write the current forcefield parameter set to a file.
to_string(smirnoff_data)	Render the forcefield parameter set to a string

__init__()

Create a new ParameterIOHandler.

Methods

__init__()	Create a new ParameterIOHandler.
parse_file(file_path)	**Parameters**
parse_string(data)	Parse a SMIRNOFF force field definition in a seriaized format
to_file(file_path, smirnoff_data)	Write the current forcefield parameter set to a file.
to_string(smirnoff_data)	Render the forcefield parameter set to a string

openforcefield.typing.engines.smirnoff.io.XMLParameterIOHandler

class openforcefield.typing.engines.smirnoff.io.**XMLParameterIOHandler**

Handles serialization/deserialization of SMIRNOFF ForceField objects from OFFXML format.

Methods

parse_file(source)	Parse a SMIRNOFF force field definition in XML format, read from a file.
parse_string(data)	Parse a SMIRNOFF force field definition in XML format.
to_file(file_path, smirnoff_data)	Write the current forcefield parameter set to a file.
to_string(smirnoff_data)	Write the current forcefield parameter set to an XML string.

__init__()

Create a new ParameterIOHandler.

Methods

__init__()	Create a new ParameterIOHandler.
parse_file(source)	Parse a SMIRNOFF force field definition in XML format, read from a file.
parse_string(data)	Parse a SMIRNOFF force field definition in XML format.
to_file(file_path, smirnoff_data)	Write the current forcefield parameter set to a file.
to_string(smirnoff_data)	Write the current forcefield parameter set to an XML string.

Parameter Attributes

ParameterAttribute and IndexedParameterAttribute provide a standard backend for ParameterHandler and Parameter attributes, while also enforcing validation of types and units.

ParameterAttribute	A descriptor for ParameterType attributes.
IndexedParameterAttribute	The attribute of a parameter with an unspecified number of terms.

openforcefield.typing.engines.smirnoff.parameters.ParameterAttribute

class openforcefield.typing.engines.smirnoff.parameters.**ParameterAttribute**(*default=<class 'openforcefield.typing.engines.smirnoff.parameters.UNDEFINED'>*, *unit=None*, *converter=None*)

A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures

converter(value): -> converted_value converter(instance, parameter_attribute, value): -> converted_value

A decorator syntax is available (see example below).

Parameters

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.

converter [callable, optional] An optional function that can be used to convert values before setting the attribute.

See also:

`IndexedParameterAttribute` A parameter attribute with multiple terms.

Examples

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from simtk import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openforcefield.utils.utils.IncompatibleUnitError: attr_quantity=3.0 dimensionless should have␣
↪units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead. >>> my_par.attr_int_to_float = 3 >>> my_par.attr_int_to_float 3.0 >>> my_par.attr_int_to_float = '4.0' Traceback (most recent call last): ... TypeError: Cannot convert '4.0' to float

Attributes

name

Methods

UNDEFINED()	Custom type used by ParameterAttribute to differentiate between None and undeclared default.
converter(converter)	Create a new ParameterAttribute with an associated converter.

__init__(*default=<class 'openforcefield.typing.engines.smirnoff.parameters.ParameterAttribute.UNDEFINED'>*, *unit=None*, *converter=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([default, unit, converter])	Initialize self.
converter(converter)	Create a new ParameterAttribute with an associated converter.

Attributes

name

openforcefield.typing.engines.smirnoff.parameters.IndexedParameterAttribute

class openforcefield.typing.engines.smirnoff.parameters.**IndexedParameterAttribute**(*default=<class 'openforcefield.typing.engines.smirnoff.pa unit=None*, *converter=None*)

The attribute of a parameter with an unspecified number of terms.

Some parameters can be associated to multiple terms, For example, torsions have parameters such as k1, k2, ..., and IndexedParameterAttribute can be used to encapsulate the sequence of terms.

The only substantial difference with ParameterAttribute is that only sequences are supported as values and converters and units are checked on each element of the sequence.

Currently, the descriptor makes the sequence immutable. This is to avoid that an element of the sequence could be set without being properly validated. In the future, the data could be wrapped in a safe list that would safely allow mutability.

Parameters

default [object, optional] When specified, the descriptor makes this attribute optional by attaching a default value to it.

unit [simtk.unit.Quantity, optional] When specified, only sequences of quantities with compatible units are allowed to be set.

converter [callable, optional] An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

See also:

`ParameterAttribute` A simple parameter attribute.

Examples

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into Quantity objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with `ParameterAttribute`, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

Attributes

name

Methods

`UNDEFINED`()	Custom type used by `ParameterAttribute` to differentiate between `None` and undeclared default.
`converter`(converter)	Create a new ParameterAttribute with an associated converter.

__init__(*default=<class 'openforcefield.typing.engines.smirnoff.parameters.ParameterAttribute.UNDEFINED'>*, *unit=None*, *converter=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([default, unit, converter])	Initialize self.
`converter`(converter)	Create a new ParameterAttribute with an associated converter.

Attributes

`name`

2.3 Utilities

2.3.1 Toolkit wrappers

The toolkit wrappers provide a simple uniform API for accessing minimal functionality of cheminformatics toolkits.

These toolkit wrappers are generally used through a `ToolkitRegistry`, which can be constructed with a desired precedence of toolkits:

```
from openforcefield.utils.toolkits import ToolkitRegistry
toolkit_registry = ToolkitRegistry()
toolkit_precedence = [OpenEyeToolkitWrapper, RDKitToolkitWrapper, AmberToolsToolkitWrapper]
[ toolkit_registry.register(toolkit) for toolkit in toolkit_precedence if toolkit.is_available() ]
```

Alternatively, the global toolkit registry (which will attempt to register any available toolkits) can be used:

```
from openforcefield.utils.toolkits import DEFAULT_TOOLKIT_REGISTRY as toolkit_registry
```

The toolkit wrappers can then be accessed through the registry:

```
molecule = Molecule.from_smiles('Cc1ccccc1')
smiles = toolkit_registry.call('to_smiles', molecule)
```

`ToolkitRegistry`	Registry for ToolkitWrapper objects
`ToolkitWrapper`	Toolkit wrapper base class.
`OpenEyeToolkitWrapper`	OpenEye toolkit wrapper
`RDKitToolkitWrapper`	RDKit toolkit wrapper
`AmberToolsToolkitWrapper`	AmberTools toolkit wrapper

openforcefield.utils.toolkits.ToolkitRegistry

class openforcefield.utils.toolkits.**ToolkitRegistry**(*register_imported_toolkit_wrappers=False*, *toolkit_precedence=None*, *exception_if_unavailable=True*)

Registry for ToolkitWrapper objects

Examples

Register toolkits in a specified order, skipping if unavailable

```
>>> from openforcefield.utils.toolkits import ToolkitRegistry
>>> toolkit_registry = ToolkitRegistry()
>>> toolkit_precedence = [OpenEyeToolkitWrapper, RDKitToolkitWrapper, AmberToolsToolkitWrapper]
>>> for toolkit in toolkit_precedence:
...     if toolkit.is_available():
...         toolkit_registry.register_toolkit(toolkit)
```

Register specified toolkits, raising an exception if one is unavailable

```
>>> toolkit_registry = ToolkitRegistry()
>>> toolkits = [OpenEyeToolkitWrapper, AmberToolsToolkitWrapper]
>>> for toolkit in toolkits:
...     toolkit_registry.register_toolkit(toolkit)
```

Register all available toolkits in arbitrary order

```
>>> from openforcefield.utils import all_subclasses
>>> toolkits = all_subclasses(ToolkitWrapper)
>>> for toolkit in toolkit_precedence:
...     if toolkit.is_available():
...         toolkit_registry.register_toolkit(toolkit)
```

Retrieve the global singleton toolkit registry, which is created when this module is imported from all available toolkits:

```
>>> from openforcefield.utils.toolkits import GLOBAL_TOOLKIT_REGISTRY as toolkit_registry
>>> available_toolkits = toolkit_registry.registered_toolkits
```

Warning: This API is experimental and subject to change.

Attributes

registered_toolkits List registered toolkits.

Methods

add_toolkit(toolkit_wrapper)	Append a ToolkitWrapper onto the list of toolkits in this ToolkitRegistry
call(method_name, *args[, raise_exception_types])	Execute the requested method by attempting to use all registered toolkits in order of precedence.
register_toolkit(toolkit_wrapper[, ...])	Register the provided toolkit wrapper class, instantiating an object of it.
resolve(method_name)	Resolve the requested method name by checking all registered toolkits in order of precedence for one that provides the requested method.

__init__(*register_imported_toolkit_wrappers=False*, *toolkit_precedence=None*, *exception_if_unavailable=True*)
Create an empty toolkit registry.

Parameters

register_imported_toolkit_wrappers [bool, optional, default=False]

If True, will attempt to register all imported ToolkitWrapper subclasses that can be found, in no p
order.

toolkit_precedence [list, optional, default=None] List of toolkit wrapper classes, in order of desired precedence when performing molecule operations. If None, defaults to [OpenEyeToolkitWrapper, RDKitToolkitWrapper, AmberToolsToolkitWrapper].

exception_if_unavailable [bool, optional, default=True] If True, an exception will be raised if the toolkit is unavailable

Methods

__init__([...])	Create an empty toolkit registry.
add_toolkit(toolkit_wrapper)	Append a ToolkitWrapper onto the list of toolkits in this ToolkitRegistry
call(method_name, *args[, raise_exception_types])	Execute the requested method by attempting to use all registered toolkits in order of precedence.
register_toolkit(toolkit_wrapper[, ...])	Register the provided toolkit wrapper class, instantiating an object of it.
resolve(method_name)	Resolve the requested method name by checking all registered toolkits in order of precedence for one that provides the requested method.

Attributes

registered_toolkits	List registered toolkits.

openforcefield.utils.toolkits.ToolkitWrapper

class openforcefield.utils.toolkits.**ToolkitWrapper**

Toolkit wrapper base class.

> **Warning:** This API is experimental and subject to change.

Attributes

toolkit_file_read_formats List of file formats that this toolkit can read.

toolkit_file_write_formats List of file formats that this toolkit can write.

toolkit_installation_instructions classmethod(function) -> method

toolkit_name The name of the toolkit wrapped by this class.

Methods

from_file(file_path, file_format[, ...])	Return an openforcefield.topology.Molecule from a file using this toolkit.
from_file_obj(file_obj, file_format[, ...])	Return an openforcefield.topology.Molecule from a file-like object (an object with a ".read()" method using this
is_available()	Check whether the corresponding toolkit can be imported
requires_toolkit()	

__init__(**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(*args, **kwargs)	Initialize self.
from_file(file_path, file_format[, ...])	Return an openforcefield.topology.Molecule from a file using this toolkit.
from_file_obj(file_obj, file_format[, ...])	Return an openforcefield.topology.Molecule from a file-like object (an object with a ".read()" method using this
is_available()	Check whether the corresponding toolkit can be imported
requires_toolkit()	

Attributes

toolkit_file_read_formats	List of file formats that this toolkit can read.
toolkit_file_write_formats	List of file formats that this toolkit can write.
toolkit_installation_instructions	classmethod(function) -> method
toolkit_name	The name of the toolkit wrapped by this class.

openforcefield.utils.toolkits.OpenEyeToolkitWrapper

class openforcefield.utils.toolkits.**OpenEyeToolkitWrapper**

OpenEye toolkit wrapper

> **Warning:** This API is experimental and subject to change.

Attributes

toolkit_file_read_formats List of file formats that this toolkit can read.

toolkit_file_write_formats List of file formats that this toolkit can write.

toolkit_installation_instructions classmethod(function) -> method

toolkit_name The name of the toolkit wrapped by this class.

Methods

assign_fractional_bond_orders(molecule[, ...])	Update and store list of bond orders this molecule.
assign_partial_charges(molecule[, ...])	Compute partial charges with OpenEye quacpac, and assign the new values to the partial_charges attribute.
canonical_order_atoms(molecule)	Canonical order the atoms in the molecule using the OpenEye toolkit.
compute_partial_charges_am1bcc(molecule[, ...])	Compute AM1BCC partial charges with OpenEye quacpac.
enumerate_protomers(molecule[, max_states])	Enumerate the formal charges of a molecule to generate different protomoers.
enumerate_stereoisomers(molecule[, ...])	Enumerate the stereocenters and bonds of the current molecule.
enumerate_tautomers(molecule[, max_states])	Enumerate the possible tautomers of the current molecule
find_smarts_matches(molecule, smarts[, ...])	Find all SMARTS matches for the specified molecule, using the specified aromaticity model.
from_file(file_path, file_format[, ...])	Return an openforcefield.topology.Molecule from a file using this toolkit.
from_file_obj(file_obj, file_format[, ...])	Return an openforcefield.topology.Molecule from a file-like object (an object with a ".read()" method using this toolkit.
from_inchi(inchi[, allow_undefined_stereo])	Construct a Molecule from a InChI representation

continues on next page

Table 99 – continued from previous page

from_object(object[, allow_undefined_stereo])	If given an OEMol (or OEMol-derived object), this function will load it into an openforcefield.topology.molecule
from_openeye(oemol[, allow_undefined_stereo])	Create a Molecule from an OpenEye molecule.
from_smiles(smiles[, ...])	Create a Molecule from a SMILES string using the OpenEye toolkit.
generate_conformers(molecule[, ...])	Generate molecule conformers using OpenEye Omega.
get_tagged_smarts_connectivity(smarts)	Returns a tuple of tuples indicating connectivity between tagged atoms in a SMARTS string.
is_available([oetools])	Check if the given OpenEye toolkit components are available.
requires_toolkit()	
to_file(molecule, file_path, file_format)	Writes an OpenFF Molecule to a file-like object
to_file_obj(molecule, file_obj, file_format)	Writes an OpenFF Molecule to a file-like object
to_inchi(molecule[, fixed_hydrogens])	Create an InChI string for the molecule using the RDKit Toolkit.
to_inchikey(molecule[, fixed_hydrogens])	Create an InChIKey for the molecule using the RDKit Toolkit.
to_openeye(molecule[, aromaticity_model])	Create an OpenEye molecule using the specified aromaticity model
to_smiles(molecule[, isomeric, ...])	Uses the OpenEye toolkit to convert a Molecule into a SMILES string.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
assign_fractional_bond_orders(molecule[, ...])	Update and store list of bond orders this molecule.
assign_partial_charges(molecule[, ...])	Compute partial charges with OpenEye quacpac, and assign the new values to the partial_charges attribute.
canonical_order_atoms(molecule)	Canonical order the atoms in the molecule using the OpenEye toolkit.
compute_partial_charges_am1bcc(molecule[, ...])	Compute AM1BCC partial charges with OpenEye quacpac.
enumerate_protomers(molecule[, max_states])	Enumerate the formal charges of a molecule to generate different protomoers.
enumerate_stereoisomers(molecule[, ...])	Enumerate the stereocenters and bonds of the current molecule.
enumerate_tautomers(molecule[, max_states])	Enumerate the possible tautomers of the current molecule
find_smarts_matches(molecule, smarts[, ...])	Find all SMARTS matches for the specified molecule, using the specified aromaticity model.
from_file(file_path, file_format[, ...])	Return an openforcefield.topology.Molecule from a file using this toolkit.

continues on next page

Table 100 – continued from previous page

from_file_obj(file_obj, file_format[, ...])	Return an openforcefield.topology.Molecule from a file-like object (an object with a ".read()" method using this toolkit.
from_inchi(inchi[, allow_undefined_stereo])	Construct a Molecule from a InChI representation
from_object(object[, allow_undefined_stereo])	If given an OEMol (or OEMol-derived object), this function will load it into an openforcefield.topology.molecule
from_openeye(oemol[, allow_undefined_stereo])	Create a Molecule from an OpenEye molecule.
from_smiles(smiles[, ...])	Create a Molecule from a SMILES string using the OpenEye toolkit.
generate_conformers(molecule[, ...])	Generate molecule conformers using OpenEye Omega.
get_tagged_smarts_connectivity(smarts)	Returns a tuple of tuples indicating connectivity between tagged atoms in a SMARTS string.
is_available([oetools])	Check if the given OpenEye toolkit components are available.
requires_toolkit()	
to_file(molecule, file_path, file_format)	Writes an OpenFF Molecule to a file-like object
to_file_obj(molecule, file_obj, file_format)	Writes an OpenFF Molecule to a file-like object
to_inchi(molecule[, fixed_hydrogens])	Create an InChI string for the molecule using the RDKit Toolkit.
to_inchikey(molecule[, fixed_hydrogens])	Create an InChIKey for the molecule using the RDKit Toolkit.
to_openeye(molecule[, aromaticity_model])	Create an OpenEye molecule using the specified aromaticity model
to_smiles(molecule[, isomeric, ...])	Uses the OpenEye toolkit to convert a Molecule into a SMILES string.

Attributes

toolkit_file_read_formats	List of file formats that this toolkit can read.
toolkit_file_write_formats	List of file formats that this toolkit can write.
toolkit_installation_instructions	classmethod(function) -> method
toolkit_name	The name of the toolkit wrapped by this class.

openforcefield.utils.toolkits.RDKitToolkitWrapper

class openforcefield.utils.toolkits.**RDKitToolkitWrapper**

RDKit toolkit wrapper

> **Warning:** This API is experimental and subject to change.

Attributes

toolkit_file_read_formats List of file formats that this toolkit can read.

toolkit_file_write_formats List of file formats that this toolkit can write.

toolkit_installation_instructions classmethod(function) -> method

toolkit_name The name of the toolkit wrapped by this class.

Methods

canonical_order_atoms(molecule)	Canonical order the atoms in the molecule using the RDKit.
enumerate_stereoisomers(molecule[, ...])	Enumerate the stereocenters and bonds of the current molecule.
enumerate_tautomers(molecule[, max_states])	Enumerate the possible tautomers of the current molecule.
find_smarts_matches(molecule, smarts[, ...])	Find all SMARTS matches for the specified molecule, using the specified aromaticity model.
from_file(file_path, file_format[, ...])	Create an openforcefield.topology.Molecule from a file using this toolkit.
from_file_obj(file_obj, file_format[, ...])	Return an openforcefield.topology.Molecule from a file-like object (an object with a ".read()" method using this toolkit.
from_inchi(inchi[, allow_undefined_stereo])	Construct a Molecule from a InChI representation
from_object(object[, allow_undefined_stereo])	If given an rdchem.Mol (or rdchem.Mol-derived object), this function will load it into an openforcefield.topology.molecule.
from_pdb_and_smiles(file_path, smiles[, ...])	Create a Molecule from a pdb file and a SMILES string using RDKit.
from_rdkit(rdmol[, allow_undefined_stereo])	Create a Molecule from an RDKit molecule.
from_smiles(smiles[, ...])	Create a Molecule from a SMILES string using the RDKit toolkit.
generate_conformers(molecule[, ...])	Generate molecule conformers using RDKit.
get_tagged_smarts_connectivity(smarts)	Returns a tuple of tuples indicating connectivity between tagged atoms in a SMARTS string.
is_available()	Check whether the RDKit toolkit can be imported
requires_toolkit()	
to_file(molecule, file_path, file_format)	Writes an OpenFF Molecule to a file-like object
to_file_obj(molecule, file_obj, file_format)	Writes an OpenFF Molecule to a file-like object
to_inchi(molecule[, fixed_hydrogens])	Create an InChI string for the molecule using the RDKit Toolkit.
to_inchikey(molecule[, fixed_hydrogens])	Create an InChIKey for the molecule using the RDKit Toolkit.
to_rdkit(molecule[, aromaticity_model])	Create an RDKit molecule
to_smiles(molecule[, isomeric, ...])	Uses the RDKit toolkit to convert a Molecule into a SMILES string.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`()	Initialize self.
`canonical_order_atoms`(molecule)	Canonical order the atoms in the molecule using the RDKit.
`enumerate_stereoisomers`(molecule[, ...])	Enumerate the stereocenters and bonds of the current molecule.
`enumerate_tautomers`(molecule[, max_states])	Enumerate the possible tautomers of the current molecule.
`find_smarts_matches`(molecule, smarts[, ...])	Find all SMARTS matches for the specified molecule, using the specified aromaticity model.
`from_file`(file_path, file_format[, ...])	Create an openforcefield.topology.Molecule from a file using this toolkit.
`from_file_obj`(file_obj, file_format[, ...])	Return an openforcefield.topology.Molecule from a file-like object (an object with a ".read()" method using this toolkit.
`from_inchi`(inchi[, allow_undefined_stereo])	Construct a Molecule from a InChI representation
`from_object`(object[, allow_undefined_stereo])	If given an rdchem.Mol (or rdchem.Mol-derived object), this function will load it into an openforcefield.topology.molecule.
`from_pdb_and_smiles`(file_path, smiles[, ...])	Create a Molecule from a pdb file and a SMILES string using RDKit.
`from_rdkit`(rdmol[, allow_undefined_stereo])	Create a Molecule from an RDKit molecule.
`from_smiles`(smiles[, ...])	Create a Molecule from a SMILES string using the RDKit toolkit.
`generate_conformers`(molecule[, ...])	Generate molecule conformers using RDKit.
`get_tagged_smarts_connectivity`(smarts)	Returns a tuple of tuples indicating connectivity between tagged atoms in a SMARTS string.
`is_available`()	Check whether the RDKit toolkit can be imported
`requires_toolkit`()	
`to_file`(molecule, file_path, file_format)	Writes an OpenFF Molecule to a file-like object
`to_file_obj`(molecule, file_obj, file_format)	Writes an OpenFF Molecule to a file-like object
`to_inchi`(molecule[, fixed_hydrogens])	Create an InChI string for the molecule using the RDKit Toolkit.
`to_inchikey`(molecule[, fixed_hydrogens])	Create an InChIKey for the molecule using the RDKit Toolkit.
`to_rdkit`(molecule[, aromaticity_model])	Create an RDKit molecule
`to_smiles`(molecule[, isomeric, ...])	Uses the RDKit toolkit to convert a Molecule into a SMILES string.

Attributes

toolkit_file_read_formats	List of file formats that this toolkit can read.
toolkit_file_write_formats	List of file formats that this toolkit can write.
toolkit_installation_instructions	classmethod(function) -> method
toolkit_name	The name of the toolkit wrapped by this class.

openforcefield.utils.toolkits.AmberToolsToolkitWrapper

class openforcefield.utils.toolkits.**AmberToolsToolkitWrapper**

AmberTools toolkit wrapper

> **Warning:** This API is experimental and subject to change.

Attributes

toolkit_file_read_formats List of file formats that this toolkit can read.

toolkit_file_write_formats List of file formats that this toolkit can write.

toolkit_installation_instructions classmethod(function) -> method

toolkit_name The name of the toolkit wrapped by this class.

Methods

assign_fractional_bond_orders(molecule[, ...])	Update and store list of bond orders this molecule.
assign_partial_charges(molecule[, ...])	Compute partial charges with AmberTools using antechamber/sqm, and assign the new values to the partial_charges attribute.
compute_partial_charges_am1bcc(molecule[, ...])	Compute partial charges with AmberTools using antechamber/sqm.
from_file(file_path, file_format[, ...])	Return an openforcefield.topology.Molecule from a file using this toolkit.
from_file_obj(file_obj, file_format[, ...])	Return an openforcefield.topology.Molecule from a file-like object (an object with a ".read()" method using this
is_available()	Check whether the AmberTools toolkit is installed
requires_toolkit()	

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`()	Initialize self.
`assign_fractional_bond_orders`(molecule[, ...])	Update and store list of bond orders this molecule.
`assign_partial_charges`(molecule[, ...])	Compute partial charges with AmberTools using antechamber/sqm, and assign the new values to the partial_charges attribute.
`compute_partial_charges_am1bcc`(molecule[, ...])	Compute partial charges with AmberTools using antechamber/sqm.
`from_file`(file_path, file_format[, ...])	Return an openforcefield.topology.Molecule from a file using this toolkit.
`from_file_obj`(file_obj, file_format[, ...])	Return an openforcefield.topology.Molecule from a file-like object (an object with a ".read()" method using this
`is_available`()	Check whether the AmberTools toolkit is installed
`requires_toolkit`()	

Attributes

`toolkit_file_read_formats`	List of file formats that this toolkit can read.
`toolkit_file_write_formats`	List of file formats that this toolkit can write.
`toolkit_installation_instructions`	classmethod(function) -> method
`toolkit_name`	The name of the toolkit wrapped by this class.

2.3.2 Serialization support

`Serializable`	Mix-in to add serialization and deserialization support via JSON, YAML, BSON, TOML, MessagePack, and XML.

openforcefield.utils.serialization.Serializable

class openforcefield.utils.serialization.**Serializable**

Mix-in to add serialization and deserialization support via JSON, YAML, BSON, TOML, MessagePack, and XML.

For more information on these formats, see: JSON, BSON, YAML, TOML, MessagePack, and XML.

To use this mix-in, the class inheriting from this class must have implemented `to_dict()` and `from_dict()` methods that utilize dictionaries containing only serialiable Python objects.

> **Warning:** The serialization/deserialiation schemes used here place some strict constraints on what kinds of `dict` objects can be serialized. No effort is made to add further protection to ensure serialization is possible. Use with caution.

Examples

Example class using `Serializable` mix-in:

```
>>> from openforcefield.utils.serialization import Serializable
>>> class Thing(Serializable):
...     def __init__(self, description):
...         self.description = description
...
...     def to_dict(self):
...         return { 'description' : self.description }
...
...     @classmethod
...     def from_dict(cls, d):
...         return cls(d['description'])
...
>>> # Create an example object
>>> thing = Thing('blorb')
```

Get JSON representation:

```
>>> json_thing = thing.to_json()
```

Reconstruct an object from its JSON representation:

```
>>> thing_from_json = Thing.from_json(json_thing)
```

Get BSON representation:

```
>>> bson_thing = thing.to_bson()
```

Reconstruct an object from its BSON representation:

```
>>> thing_from_bson = Thing.from_bson(bson_thing)
```

Get YAML representation:

```
>>> yaml_thing = thing.to_yaml()
```

Reconstruct an object from its YAML representation:

```
>>> thing_from_yaml = Thing.from_yaml(yaml_thing)
```

Get MessagePack representation:

```
>>> messagepack_thing = thing.to_messagepack()
```

Reconstruct an object from its MessagePack representation:

```
>>> thing_from_messagepack = Thing.from_messagepack(messagepack_thing)
```

Get XML representation:

```
>>> xml_thing = thing.to_xml()
```

Methods

from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
to_bson()	Return a BSON serialized representation.
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_pickle()	Return a pickle serialized representation.
to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

from_dict	
to_dict	

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(*args, **kwargs)	Initialize self.
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(d)	
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
to_bson()	Return a BSON serialized representation.

continues on next page

Table 110 – continued from previous page

to_dict()	
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_pickle()	Return a pickle serialized representation.
to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

2.3.3 Miscellaneous utilities

Miscellaneous utility functions.

inherit_docstrings	Inherit docstrings from parent class
all_subclasses	Recursively retrieve all subclasses of the specified class
temporary_cd	Context to temporary change the working directory.
get_data_file_path	Get the full path to one of the reference files in test-systems.
convert_0_1_smirnoff_to_0_2	Convert an 0.1-compliant SMIRNOFF dict to an 0.2-compliant one.
convert_0_2_smirnoff_to_0_3	Convert an 0.2-compliant SMIRNOFF dict to an 0.3-compliant one.

openforcefield.utils.utils.inherit_docstrings

openforcefield.utils.utils.**inherit_docstrings**(*cls*)

Inherit docstrings from parent class

openforcefield.utils.utils.all_subclasses

openforcefield.utils.utils.**all_subclasses**(*cls*)

Recursively retrieve all subclasses of the specified class

openforcefield.utils.utils.temporary_cd

openforcefield.utils.utils.**temporary_cd**(*dir_path*)

Context to temporary change the working directory. Parameters ———- dir_path : str

The directory path to enter within the context

```
>>> dir_path = '/tmp'
>>> with temporary_cd(dir_path):
...     pass  # do something in dir_path
```

openforcefield.utils.utils.get_data_file_path

openforcefield.utils.utils.**get_data_file_path**(*relative_path*)

Get the full path to one of the reference files in testsystems. In the source distribution, these files are in openforcefield/data/, but on installation, they're moved to somewhere in the user's python site-packages directory. Parameters ———- name : str

Name of the file to load (with respect to the repex folder).

openforcefield.utils.utils.convert_0_1_smirnoff_to_0_2

openforcefield.utils.utils.**convert_0_1_smirnoff_to_0_2**(*smirnoff_data_0_1*)

Convert an 0.1-compliant SMIRNOFF dict to an 0.2-compliant one. This involves renaming several tags, adding Electrostatics and ToolkitAM1BCC tags, and separating improper torsions into their own section.

Parameters

smirnoff_data_0_1 [dict] Hierarchical dict representing a SMIRNOFF data structure according the the 0.1 spec

Returns

smirnoff_data_0_2 Hierarchical dict representing a SMIRNOFF data structure according the the 0.2 spec

openforcefield.utils.utils.convert_0_2_smirnoff_to_0_3

openforcefield.utils.utils.**convert_0_2_smirnoff_to_0_3**(*smirnoff_data_0_2*)

Convert an 0.2-compliant SMIRNOFF dict to an 0.3-compliant one. This involves removing units from header tags and adding them to attributes of child elements. It also requires converting ProperTorsions and ImproperTorsions potentials from "charmm" to "fourier".

Parameters

smirnoff_data_0_2 [dict] Hierarchical dict representing a SMIRNOFF data structure according the the 0.2 spec

Returns

smirnoff_data_0_3 Hierarchical dict representing a SMIRNOFF data structure according the the 0.3 spec

2.3.4 Structure tools

Tools for manipulating molecules and structures

Warning: These methods are deprecated and will be removed and replaced with integrated API methods. We recommend that no new code makes use of these functions.

get_molecule_parameterIDs	Process a list of molecules with a specified SMIRNOFF ffxml file and determine which parameters are used by which molecules, returning collated results.

openforcefield.utils.structure.get_molecule_parameterIDs

openforcefield.utils.structure.**get_molecule_parameterIDs**(*molecules*, *forcefield*)

Process a list of molecules with a specified SMIRNOFF ffxml file and determine which parameters are used by which molecules, returning collated results.

Parameters

molecules [list of openforcefield.topology.Molecule] List of molecules (with explicit hydrogens) to parse

forcefield [openforcefield.typing.engines.smirnoff.ForceField] The ForceField to apply

Returns

parameters_by_molecule [dict] Parameter IDs used in each molecule, keyed by isomeric SMILES generated from provided OEMols. Each entry in the dict is a list which does not necessarily have unique entries; i.e. parameter IDs which are used more than once will occur multiple times.

parameters_by_ID [dict] Molecules in which each parameter ID occur, keyed by parameter ID. Each entry in the dict is a set of isomeric SMILES for molecules in which that parameter occurs. No frequency information is stored.

INDEX

Symbols

T

V

X