openforcefield Documentation

Release 0.8.4

Open Force Field Consortium

Mar 09, 2021

CONTENTS

1 User Guide 3

2 API documentation 65

Index 513

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 0.8.4 - Minor feature and bugfix release

This release is intended to be functionally identical to 0.9.1. The only difference is that it uses the "openforcefield" namespace.

This release is a final patch for the `0.8.X` series of releases of the toolkit, and also marks the last version of the toolkit which will be imported as `import openforcefield.XXX` / `from openforcefield import XXX`. From version `0.9.0` onwards the toolkit will be importable only as `import openff.toolkit.XXX` / `from openff.toolkit import XXX`.

Note This change will also be accompanied by a renaming of the package from `openforcefield` to `openff-toolkit`, so users need not worry about accidentally pulling in a version with changed imports. Users will have to explicitly choose to install the `openff-toolkit` package once released which will contain the breaking import changes.

New features

- PR #839: Add support for computing WBOs from multiple conformers using the AmberTools and OpenEye toolkits, and from ELF10 conformers using the OpenEye toolkit wrapper.
- PR #832: Expose ELF conformer selection through the `Molecule` API via a new `apply_elf_conformer_selection` function.
- PR #831: Expose ELF conformer selection through the OpenEye wrapper.
- PR #790: Fixes Issue #720 where qcschema roundtrip to/from results in an error due to missing cmiles entry in attributes.
- PR #793: Add an initial ELF conformer selection implementation which uses RDKit.
- PR #799: Closes Issue #746 by adding `Molecule.smirnoff_impropers`, `Molecule.amber_impropers`, `TopologyMolecule.smirnoff_impropers`, `TopologyMolecule.amber_impropers`, `Topology.smirnoff_impropers`, and `Topology.amber_impropers`.
- PR #847: Instances of `ParameterAttribute` documentation can now specify their docstrings with the optional `docstring` argument to the `__init__()` method.
- PR #827: The setter for `Topology.box_vectors` now infers box vectors when box lengths are pass as a list of length 3.

Behavior changed

- PR #802: Fixes Issue #408. The 1-4 scaling factor for electrostatic interactions is now properly set by the value specified in the force field. Previously it fell back to a default value of 0.83333. The toolkit may now produce slightly different energies as a result of this change.
- PR #839: The average WBO will now be returned when multiple conformers are provided to `assign_fractional_bond_orders` using `use_conformers`.
- PR #816: Force field file paths are now loaded in a case-insensitive manner.

Bugfixes

- PR #849: Changes `create_openmm_system` so that it no longer uses the conformers on existing reference molecules (if present) to calculate Wiberg bond orders. Instead, new conformers are always generated during parameterization.
- PR #838: Corrects spacing of "forcefield" to "force field" throughout documentation. Fixes Issue #112.
- PR #846: Corrects dead links throughout release history. Fixes Issue #835.
- PR #847: Documentation now compiles with far fewer warnings, and in many cases more correctly. Additionally, `ParameterAttribute` documentation no longer appears incorrectly in classes where it is used. Fixes Issue #397.
- PR #802: Fixes Issue #408. The 1-4 scaling factor for electrostatic interactions is now properly set by the value specified in the force field. Previously it fell back to a default value of 0.83333. The toolkit may now produce slightly different energies as a result of this change.

1.2.2 0.8.3 - Major bugfix release

This release fixes a critical bug in van der Waals parameter assignment.

This release is also a final patch for the `0.8.X` series of releases of the toolkit, and also marks the last version of the toolkit which will be imported as `import openforcefield.XXX` / `from openforcefield import XXX`. From version `0.9.0` onwards the toolkit will be importable only as `import openff.toolkit.XXX` / `from openff.toolkit import XXX`.

Note This change will also be accompanied by a renaming of the package from `openforcefield` to `openff-toolkit`, so users need not worry about accidentally pulling in a version with changed imports. Users will have to explicitly choose to install the `openff-toolkit` package once released which will contain the breaking import changes.

Bugfixes

- PR #808: Fixes Issue #807, which tracks a major bug in the interconversion between a vdW `sigma` and `rmin_half` parameter.

New features

- PR #794: Adds a decorator `@requires_package` that denotes a function requires an optional dependency.
- PR #790: Fixes Issue #720 where qcschema roundtrip to/from results in an error due to missing cmiles entry in attributes.
- PR #805: Adds a deprecation warning for the up-coming release of the `openff-toolkit` package and its import breaking changes.

1.2.3 0.8.2 - Bugfix release

WARNING: This release was later found to contain a major bug, Issue #807, and produces incorrect energies.

Bugfixes

- PR #786: Fixes Issue #785 where RDKitToolkitWrapper would sometimes expect stereochemistry to be defined for non-stereogenic bonds when loading from SDF.
- PR #786: Fixes an issue where using the `Molecule` copy constructor (`newmol = Molecule(oldmol)`) would result in the copy sharing the same `.properties` dict as the original (as in, changes to the `.properties` dict of the copy would be reflected in the original).
- PR #789: Fixes a regression noted in Issue #788 where creating `vdWHandler.vdWType` or setting `sigma` or `rmin_half` using Quantities represented as strings resulted in an error.

1.2.4 0.8.1 - Bugfix and minor feature release

WARNING: This release was later found to contain a major bug, Issue #807, and produces incorrect energies.

API-breaking changes

- PR #757: Renames `test_forcefields/smirnoff99Frosst.offxml` to `test_forcefields/test_forcefield.offxml` to avoid confusion with any of the ACTUAL released FFs in the smirnoff99Frosst line
- PR #751: Removes the optional `oetools=("oechem", "oequacpac", "oeiupac", "oeomega")` keyword argument from `OpenEyeToolkitWrapper.is_available`, as there are no special behaviors that are accessed in the case of partially-licensed OpenEye backends. The new behavior of this method is the same as if the default value above is always provided.

Behavior Changed

- PR #583: Methods such as `Molecule.from_rdkit` and `Molecule.from_openeye`, which delegate their internal logic to `ToolkitRegistry` functions, now guarantee that they will return an object of the correct type when being called on `Molecule`-derived classes. Previously, running these constructors using subclasses of `FrozenMolecule` would not return an instance of that subclass, but rather just an instance of a `Molecule`.
- PR #753: `ParameterLookupError` is now raised when passing to `ParameterList.index` a SMIRKS pattern not found in the parameter list.

New features

- PR #751: Adds `LicenseError`, a subclass of `ToolkitUnavailableException` which is raised when attempting to add a cheminformatics `ToolkitWrapper` for a toolkit that is installed but unlicensed.
- PR #678: Adds `ForceField.deregister_parameter_handler`.
- PR #730: Adds `Topology.is_periodic`.
- PR #753: Adds `ParameterHandler.__getitem__` to look up individual `ParameterType` objects.

Bugfixes

- PR #745: Fixes bug when serializing molecule with conformers to JSON.
- PR #750: Fixes a bug causing either `sigma` or `rmin_half` to sometimes be missing on `vdWHandler.vdWType` objects.
- PR #756: Fixes bug when running `vdWHandler.create_force` using a `vdWHandler` that was initialized using the API.
- PR #776: Fixes a bug in which the `Topology.from_openmm` and `Topology.from_mdtraj` methods would dangerously allow `unique_molecules=None`.
- PR #777: `RDKitToolkitWrapper` now outputs the full warning message when `allow_undefined_stereo=True` (previously the description of which stereo was undefined was squelched)

1.2.5 0.8.0 - Virtual Sites

This release implements the SMIRNOFF virtual site specification. The implementation enables support for models using off-site charges, including 4- and 5-point water models, in addition to lone pair modeling on various functional groups. The primary focus was on the ability to parameterize a system using virtual sites, and generating an OpenMM system with all virtual sites present and ready for evaluation. Support for formats other than OpenMM has not be implemented in this release, but may come with the appearance of the OpenFF system object. In addition to implementing the specification, the toolkit `Molecule` objects now allow the creation and manipulation of virtual sites.

Major Feature: Support for the SMIRNOFF VirtualSite tag

Virtual sites can be added to a System in two ways:

- SMIRNOFF Force Fields can contain a VirtualSites tag , specifying the addition of virtual sites according to SMARTS-based rules.
- Virtual sites can be added to a `Molecule`, and these will appear in the final OpenMM system if a virtual site handler is present in the `ForceField`.

Virtual sites are the first parameters which directly depend on 3D conformation, where the position of the virtual sites are based on vectors defined on the atoms that were matched during parameterization. Because of this, a virtual site matching the triplet of atoms 1-2-3 will define a point that is different from a triplet matching 3-2-1. This is similar to defining “right-handed” and “left-handed” coordinate systems. This subtlety interplays with two major concepts in force field development:

1) we sometimes want to define a single virtual site describing two points with the same parameters (distance, angle, etc.), such as 5-point water models
2) we have a match that produces multiple orderings of the atoms (e.g. if wildcards are present in the SMARTS pattern), and we only want one to be applied.

Case 1) is very useful for parameter optimization, where a single SMARTS-based parameter can be used to optimize both points, such as the angle defining the virtual points for a 5-point water model. Case 2) is the typical scenario for the nitrogen lone pair in ammonia, where only one point needs to be specified. We discuss a few more illustrative examples below. Beyond these attributes, the virtual site specification allows a policy for specifying how to handle exclusions in the OpenMM force evaluator. The current default is to add pairwise energy exclusions in the OpenMM system between a virtual site and all tagged atoms matched in its SMARTS (`exclusion_policy="parents"`,). Currently defined are `"none"`, `"minimal"`, and `"parents"`, where `"minimal"` specifies the single atom that the virtual site defines as the "origin". For water, for example, `"minimal"` would mean just the oxygen, whereas `"parents"` would mean all three atoms.

In order to give consistent and intended behavior, the specification was modified from its draft form in following manner: The `"name"` and `"match"` attributes have been added to each virtual site parameter type. These changes allow for

- specifying different virtual site types using the same atoms
- allowing two virtual sites with the same type and same atoms but different physical parameters to be added simultaneously
- allowing the ability to control whether the virtual site encodes one or multiple particles, based on the number of ways the matching atoms can be ordered.

The `"name"` attribute encodes whether the virtual site to be added should override an existing virtual site of the same type (e.g. hierarchy preference), or if this virtual site should be added in addition to the other existing virtual sites on the given atoms. This means that different virtual site types can share the same group of parent atoms and use the same name without overwriting each other (the default `name` is `EP` for all sites, which gives the expected hierarchical behavior used in other SMIRNOFF tags).

The `"match"` attribute accepts either `"once"` or `"all_permutations"`, offering control for situations where a SMARTS pattern can possibly match the same group of atoms in different orders (either due to wildcards or local symmetry) and it is desired to either add just one or all of the possible virtual particles. The default value is `"all_permutations"`, but for TrivalentLonePair it is always set to `"once"`, regardless of what the file contains, since all orderings always place the particle in the exact same position.

The following cases exemplify our reasoning in implementing this behavior, and should draw caution to complex issues that may arise when designing virtual site parameters. Let us consider 4-, 5-, and 6-point water models:

- A 4-point water model with a `DivalentLonePair`: This can be implemented by specifying `match="once"`, `outOfPlaneAngle="0*degree"`, and `distance=-.15*angstrom"`. Since the SMIRKS pattern `"[#1:1]-[#8X2:2]-[#2:3]"` would match water twice and would create two particles in the exact same position if `all_permutations` was specified, we specify `"once"` to have only one particle generated. Although having two particles in the same position should not affect the physics if the proper exclusion policy is applied, it would effectively make the 4-point model just as expensive as 5-point models.
- A 5-point water model with a `DivalentLonePair`: This can be implemented by using `match="all_permutations"` (unlike the 4-point model), `outOfPlaneAngle="56.26*degree`, and `distance=0.7*angstrom`, for example. Here the permutations will cause particles to be placed at $\pm$56.26 degrees, and changing any of the physical quantities will affect *both* particles.
- A 6-point water model with both `DivalentLonePair` sites above. Since these two parameters look identical, it is unclear whether they should both be applied or if one should override the other. The toolkit never compares the physical numbers to determine equality as this can lead to instability during e.g. parameter fitting. To get this to work, we specify `name="EP1"` for the first parameter, and `name="EP2"` for the second parameter. This instructs the parameter handler keep them separate, and therefore both are applied. (If both had the same name, then the typical SMIRNOFF hierarchy rules are used, and only the last matched parameter would be applied.)

- Dinitrogen, N#N with a BondCharge virtual site. Since we want a BondCharge on both ends, we specify match="all_permutations".
- Formaldehyde, H2C=O, with MonovalentLonePair virtual site(s) on the oxygen, with the aim of modeling both lone pairs. This one is subtle, since [#1:3]-[#6X3:2]=[#8X1:1] matches two unique groups of atoms (1-3-4 and 2-3-4). It is important to note in this situation that match="all_permutations" behaves exactly the same as match="once". Due to the anchoring hydrogens (1 and 2) being symmetric but opposite about the bond between 3 and 4, a single parameter does correctly place both lone pairs. A standing issue here is that the default exclusion policy (parents) will allow these two virtual sites to interact since they have different indexed atoms (parents), causing the energy to be different than the non-virtual site parameterization. In the future, the exclusion_policy="local" will account for this, and make virtual sites that share at least one "parent" atom not interact with each other. As a special note: when applying a MonovalentLonePair to a completely symmetric molecule, e.g. water, all_permutations can come into play, but this will apply two particles (one for each hydrogen).

Finally, the toolkit handles the organization of atoms and virtual sites in a specific manner. Virtual sites are expected to be added *after all molecules in the topology are present*. This is because the Open Force Field Toolkit organizes a topology by placing all atoms first, then all virtual sites last. This differs from the OpenMM Modeller object, for example, which interleaves the order of atoms and virtual sites in such a way that all particles of a molecule are contiguous. In addition, due to the fact that a virtual site may contain multiple particles coupled to single parameters, the toolkit makes a distinction between a virtual *site*, and a virtual *particle*. A virtual site may represent multiple virtual particles, so the total number of particles cannot be directly determined by simply summing the number of atoms and virtual sites in a molecule. This is taken into account, however, and the Molecule and Topology classes now implement particle iterators.

Minor Feature: Support for the 0.4 ChargeIncrementModel tag

To allow for more convenient fitting of ChargeIncrement parameters, it is now possible to specify one less charge_increment value than there are tagged atoms in a ChargeIncrement's smirks. The missing charge_increment value will be calculated at parameterization-time to make the sum of the charge contributions from a ChargeIncrement parameter equal to zero. Since this change allows for force fields that are incompatible with the previous specification, this new style of ChargeIncrement must specify a ChargeIncrementModel section version of 0.4. All 0.3-compatible ChargeIncrement parameters are compatible with the 0.4 ChargeIncrementModel specification.

More details and examples of this change are available in The ChargeIncrementModel tag in the SMIRNOFF specification

New features

- PR #726: Adds support for the 0.4 ChargeIncrementModel spec, allowing for the specification of one fewer charge_increment values than there are tagged atoms in the smirks, and automatically assigning the final atom an offsetting charge.
- PR #548: Adds support for the VirtualSites tag in the SMIRNOFF specification
- PR #548: Adds replace and all_permutations kwarg to
  - Molecule.add_bond_charge_virtual_site
  - Molecule.add_monovalent_lone_pair_virtual_site
  - Molecule.add_divalent_lone_pair_virtual_site
  - Molecule.add_trivalent_lone_pair_virtual_site
- PR #548: Adds orientations to
  - BondChargeVirtualSite

  - MonovalentLonePairVirtualSite
  - DivalentLonePairVirtualSite
  - TrivalentLonePairVirtualSite
- PR #548: Adds
  - VirtualParticle
  - TopologyVirtualParticle
  - BondChargeVirtualSite.get_openmm_virtual_site
  - MonovalentLonePairVirtualSite.get_openmm_virtual_site
  - DivalentLonePairVirtualSite.get_openmm_virtual_site
  - TrivalentLonePairVirtualSite.get_openmm_virtual_site
  - ValenceDict.key_transform
  - ValenceDict.index_of
  - ImproperDict.key_transform
  - ImproperDict.index_of
- PR #705: Adds interpolation based on fractional bond orders for harmonic bonds. This includes interpolation for both the force constant `k` and/or equilibrium bond distance `length`. This is accompanied by a bump in the <Bonds> section of the SMIRNOFF spec (but not the entire spec).
- PR #718: Adds `.rings` and `.n_rings` to `Molecule` and `.is_in_ring` to `Atom` and `Bond`

Bugfixes

- PR #682: Catches failures in `Molecule.from_iupac` instead of silently failing.
- PR #743: Prevents the non-bonded (vdW) cutoff from silently falling back to the OpenMM default of 1 nm in `Forcefield.create_openmm_system` and instead sets its to the value specified by the force field.
- PR #737: Prevents OpenEye from incidentally being used in the conformer generation step of `AmberToolsToolkitWrapper.assign_fractional_bond_orders`.

Behavior changed

- PR #705: Changes the default values in the <Bonds> section of the SMIRNOFF spec to `fractional_bondorder_method="AM1-Wiberg"` and `potential="(k/2)*(r-length)^2"`, which is backwards-compatible with and equivalent to `potential="harmonic"`.

Examples added

- PR #548: Adds a virtual site example notebook to run an OpenMM simulation with virtual sites, and compares positions and potential energy of TIP5P water between OpenFF and OpenMM forcefields.

API-breaking changes

- PR #548: Methods
  - `Molecule.add_bond_charge_virtual_site`
  - `Molecule.add_monovalent_lone_pair_virtual_site`
  - `Molecule.add_divalent_lone_pair_virtual_site`
  - `Molecule.add_trivalent_lone_pair_virtual_site`

  now only accept a list of atoms, not a list of integers, to define to parent atoms
- PR #548: Removes `VirtualParticle.molecule_particle_index`
- PR #548: Removes `outOfPlaneAngle` from
  - `DivalentLonePairVirtualSite`
  - `TrivalentLonePairVirtualSite`
- PR #548: Removes `inPlaneAngle` from `TrivalentLonePairVirtualSite`
- PR #548: Removes `weights` from
  - `BondChargeVirtualSite`
  - `MonovalentLonePairVirtualSite`
  - `DivalentLonePairVirtualSite`
  - `TrivalentLonePairVirtualSite`

Tests added

- PR #548: Adds test for
  - The virtual site parameter handler
  - TIP5P water dimer energy and positions
  - Adds tests to for virtual site/particle indexing/counting

1.2.6 0.7.2 - Bugfix and minor feature release

New features

- PR #662: Adds `.aromaticity_model` of `ForceField` and `.TAGNAME` of `ParameterHandler` as public attributes.
- PR #667 and PR #681 linted the codebase with `black` and `isort`, respectively.
- PR #675 adds `.toolkit_version` to `ToolkitWrapper` and `.registered_toolkit_versions` to `ToolkitRegistry`.
- PR #696 Exposes a setter for `ForceField.aromaticity_model`
- PR #685 Adds a custom `__hash__` function to `ForceField`

Behavior changed

- PR #684: Changes `ToolkitRegistry` to return an empty registry when initialized with no arguments, i.e. `ToolkitRegistry()` and makes the `register_imported_toolkit_wrappers` argument private.
- PR #711: The setter for `Topology.box_vectors` now infers box vectors (a 3x3 matrix) when box lengths (a 3x1 array) are passed, assuming an orthogonal box.
- PR #649: Makes SMARTS searches stereochemistry-specific (if stereo is specified in the SMARTS) for both OpenEye and RDKit backends. Also ensures molecule aromaticity is re-perceived according to the ForceField's specified aromaticity model, which may overwrite user-specified aromaticity on the `Molecule`
- PR #648: Removes the `utils.structure` module, which was deprecated in 0.2.0.
- PR #670: Makes the `Topology` returned by `create_openmm_system` contain the partial charges and partial bond orders (if any) assigned during parameterization.
- PR #675 changes the exception raised when no `antechamber` executable is found from `IOError` to `AntechamberNotFoundError`
- PR #696 Adds an `aromaticity_model` keyword argument to the `ForceField` constructor, which defaults to `DEFAULT_AROMATICITY_MODEL`.

Bugfixes

- PR #715: Closes issue Issue #475 writing a "PDB" file using OE backend rearranges the order of the atoms by pushing the hydrogens to the bottom.
- PR #649: Prevents 2020 OE toolkit from issuing a warning caused by doing stereo-specific smarts searches on certain structures.
- PR #724: Closes issue Issue #502 Adding a utility function Topology.to_file() to write topology and positions to a "PDB" file using openmm backend for pdb file write.

Tests added

- PR #694: Adds automated testing to code snippets in docs.
- PR #715: Adds tests for pdb file writes using OE backend.
- PR #724: Adds tests for the utility function Topology.to_file().

1.2.7 0.7.1 - OETK2020 Compatibility and Minor Update

This is the first of our patch releases on our new planned monthly release schedule.

Detailed release notes are below, but the major new features of this release are updates for compatibility with the new 2020 OpenEye Toolkits release, the `get_available_force_fields` function, and the disregarding of pyrimidal nitrogen stereochemistry in molecule isomorphism checks.

Behavior changed

- PR #646: Checking for `Molecule` equality using the == operator now disregards all pyrimidal nitrogen stereochemistry by default. To re-enable, use `Molecule.{is|are}_isomorphic` with the `strip_pyrimidal_n_atom_stereo=False` keyword argument.
- PR #646: Adds an optional `toolkit_registry` keyword argument to `Molecule.are_isomorphic`, which identifies the toolkit that should be used to search for pyrimidal nitrogens.

Bugfixes

- PR #647: Updates `OpenEyeToolkitWrapper` for 2020.0.4 OpenEye Toolkit behavior/API changes.
- PR #646: Fixes a bug where `Molecule.chemical_environment_matches` was not able to accept a `ChemicalEnvironment` object as a query.
- PR #634: Fixes a bug in which calling `RDKitToolkitWrapper.from_file` directly would not load files correctly if passed lowercase `file_format`. Note that this bug did not occur when calling `Molecule.from_file`.
- PR #631: Fixes a bug in which calling `unit_to_string` returned `None` when the unit is dimensionless. Now `"dimensionless"` is returned.
- PR #630: Closes issue Issue #629 in which the wrong exception is raised when attempting to instantiate a `ForceField` from an unparsable string.

New features

- PR #632: Adds `ForceField.registered_parameter_handlers`
- PR #614: Adds `ToolkitRegistry.deregister_toolkit` to de-register registered toolkits, which can include toolkit wrappers loaded into `GLOBAL_TOOLKIT_REGISTRY` by default.
- PR #656: Adds a new allowed `am1elf10` option to the OpenEye implementation of `assign_partial_charges` which calculates the average partial charges at the AM1 level of theory using conformers selected using the ELF10 method.
- PR #643: Adds `openforcefield.typing.engines.smirnoff.forcefield.get_available_force_fields`, which returns paths to the files of force fields available through entry point plugins.

1.2.8 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.

1.2.9 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.

1.2.10 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.

1.2.11 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.

1.2.12 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.

1.2.13 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`.

1.2.14 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

1.2.15 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

1.2.16 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

1.2.17 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

1.2.18 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 the example on using SMIRNOFF in AMBER or GROMACS 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="0.4" 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"/>
  <!--- 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"/>
  <!-- As of version 0.4 of the ChargeIncrementModel tag, it is possible to define one less charge_
↪increment attribute than there are tagged atoms -->
  <!-- The final, undefined charge_increment will be calculated as to make the sum of the charge_
↪increments equal 0 -->
  <ChargeIncrement smirks="[#6X4:1]-[#8:2]" charge_increment1="-0.0718*elementary_charge"/>
  <ChargeIncrement smirks="[N]-[C:1]-[C:2]-[Cl:3]" charge_increment1="-0.123" charge_increment2="0.456"␣
↪/>
</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.

Starting in the 0.4 version of this section, a `ChargeIncrement` may be specified with one less `charge_increment` value than it has tagged atoms. The missing `charge_increment` value must be that of the highest tagged atom index. This missing `charge_increment` will be calculated to offset the sum of the other `charge_increments` in the same `ChargeIncrement` parameter to achieve a net value of 0. This allows `ChargeIncrement` parameters to be defined similar to bond charge corrections.

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.

ChargeIncrementModel section tag version	Tag attributes and default values	Required parameter attributes	Optional parameter attributes
0.3	`number_of_conformers="1", partial_charge_method='AM1-Mulliken'`	`smirks`, `charge_increment` (indexed, must be equal to number of tagged atoms in `smirks`)	`name`, `id`, `parent_id`
0.4	`number_of_conformers="1", partial_charge_method='AM1-Mulliken'`	`smirks`, `charge_increment` (indexed, must be equal to- or one less than-number of tagged atoms in `smirks`)	`name`, `id`, `parent_id`

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

Warning: 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 a recent force field in the "Parsley" line (openff-1.2.0.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>
```

(continues on next page)

(continued from previous page)

```
   <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

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="AM1-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 `AM1-Wiberg`.
- `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`
0.4	`potential="(k/2)*(r-length)^2"`, `fractional_bondorder_method="AM1-Wiberg"`, `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 piecewiese 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.

ProperTorsions 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`
0.4	`potential="k*(1+cos(periodicity*theta-phase))"`, `default_idivf="auto"`, `fractional_bondorder_method="AM1-Wiberg"`, `fractional_bondorder_interpolation="linear"`	`smirks`, (`k` OR `k_bondorder`), `phase`, `periodicity`	`idivf`, `id`, `parent_id`

<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"/>
```

(continues on next page)

(continued from previous page)

```
  <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

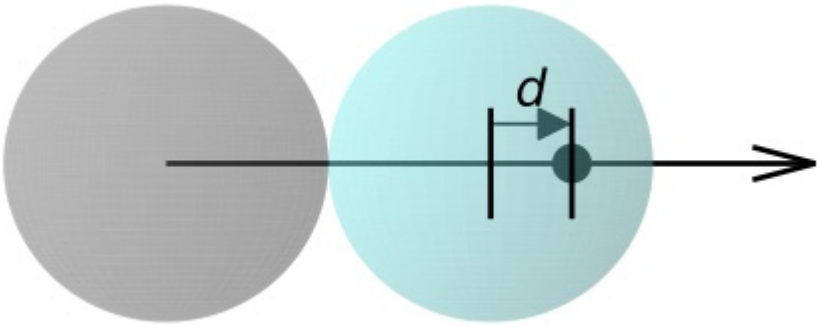
1.3.10 Advanced features

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

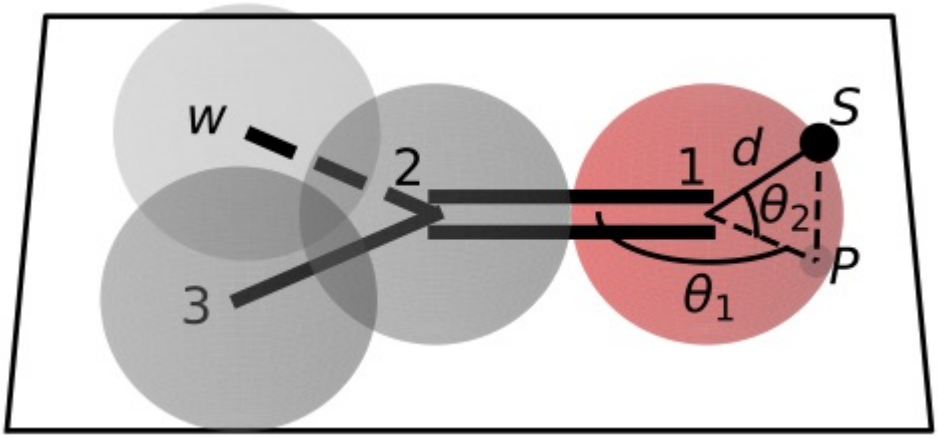
<VirtualSites>: Virtual sites for off-atom charges

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):

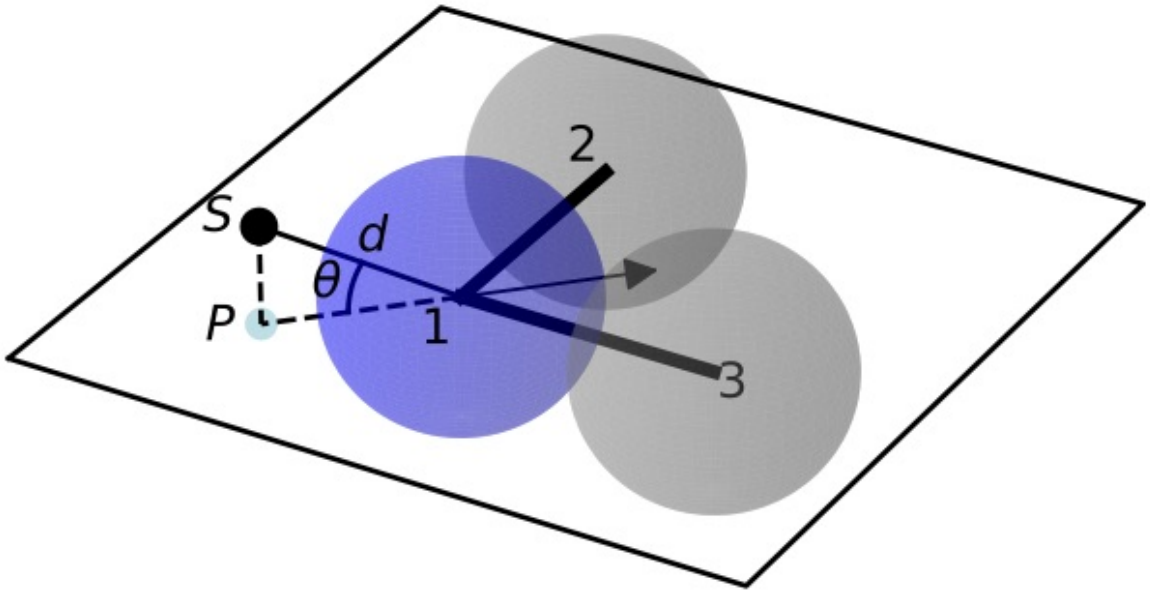
- `BondCharge`: This supports placement of a virtual site `S` along a vector between two specified atoms, e.g. to allow for a sigma hole for halogens or similar contexts. With positive values of the distance, the virtual site lies outside the first indexed atom (green in this image).



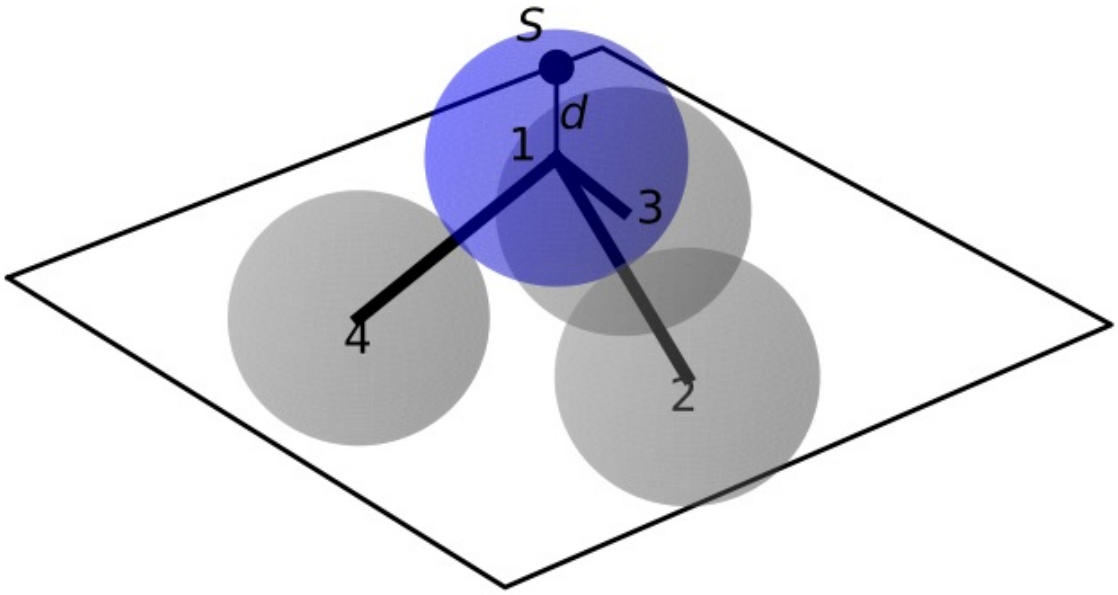
- `MonovalentLonePair`: This is originally intended for situations like a carbonyl, and allows placement of a virtual site `S` at a specified distance `d`, `inPlaneAngle` (theta 1 in the diagram), and `outOfPlaneAngle` (theta 2 in the diagram) relative to a central atom and two connected atoms.



- `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. Additionally, each virtual site can bear Lennard-Jones parameters, specified by `sigma` and `epsilon` or `rmin_half` and `epsilon`. If unspecified these default to zero.

In the SMIRNOFF format, these are encoded as:

```
<VirtualSites version="0.3" exclusion_policy="parents">
  <!-- 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" charge_
↪increment2="0.2*elementary_charge" charge_increment3="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. -->
  <VirtualSite type="DivalentLonePair" smirks="[*:2]~[#7X2:1]~[*:3]" distance="0.30*angstrom"␣
↪outOfPlaneAngle="0.0*degree" charge_increment1="0.1*elementary_charge" charge_increment2="0.
↪2*elementary_charge" charge_increment3="0.2*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 -->
  <!-- To ensure that the second site does not overwrite the first, specify a unique name for each. -->
  <VirtualSite type="TrivalentLonePair" smirks="[*:2]~[#7X3:1](~[*:4])~[*:3]" name="A" distance="0.
↪30*angstrom" charge_increment1="0.1*elementary_charge" charge_increment2="0.2*elementary_charge"␣
↪charge_increment3="0.2*elementary_charge" charge_increment4="0.2*elementary_charge"/>
  <VirtualSite type="TrivalentLonePair" smirks="[*:2]~[#7X3:1](~[*:4])~[*:3]" name="B" distance="-0.
↪30*angstrom" charge_increment1="0.1*elementary_charge" charge_increment2="0.2*elementary_charge"␣
↪charge_increment3="0.2*elementary_charge" charge_increment4="0.2*elementary_charge"/>
</VirtualSites>
```

VirtualSites section tag version	Tag attributes and default values	Required parameter attributes and default values	Optional parameter attributes
0.3	exclusion_policy="parents"	smirks, type, distance, charge_increment (indexed), inPlaneAngle IF type="MonovalentLonePair", outOfPlaneAngle IF type="MonovalentLonePair" OR type="DivalentLonePair", sigma=0.*angstrom, epsilon=0.*kilocalories_per_mole, name="EP", match="all_permutations" IF type=BondCharge OR type="MonovalentLonePair OR type="DivalentLonePair", match="once" IF type="TrivalentLonePair	N/A

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()
```

(continues on next page)

(continued from previous page)

```
# Load the first release of the "Parsley" forcefield
from openforcefield.typing.engines.smirnoff import ForceField
forcefield = ForceField('openff-1.0.0.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 how 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/using_smirnoff_with_amber_protein_forcefield, where an AMBER family force field is used for a protein and the original "Parsley" force field (`openff-1.0.0`) 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. Each can be run interactively in the browser with binder, without installing anyting on your computer.

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.

ParameterType

ToolkitRegistry

`ToolkitRegistry.from_object` / `ToolkitRegistry.from_smiles` / `OpenEyeToolkitWrapper.from_openeye` / `RDKit`

- These methods are a bit strange because they are effectively classmethods for `FrozenMolecule` and `Molecule` subclasses. In PR #583, jaimergp raised a concern that effectively boils down to "if I subclass `Molecule` into a new class, `MyMol`, then I expect `MyMol.from_rdkit` to return an instance of `MyMol`, not `Molecule`. However, before this PR, methods like `ToolkitRegistry.from_smiles` didn't have any way to know what type of object they should return, and instead always returned `Molecule` objects. So as of PR #583, ToolkitRegistry methods that produce a Molecule must take a private parameter, `_cls`, indicating the type of object to return. This parameter should be of type `type` and should subclass `FrozenMolecule`, or otherwise expose `Molecule._add_atom`, `._add_bond`, `.add_conformer`, and `.partial_charges`.

Molecule.to_X

Molecule.from_X

Force field directories

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
$ 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, even if code could be written more compactly. For example, 15-character variable names are fine. Triply nested list comprehensions are not.

The `openforcefield` toolkit is in the process of adopting code formatting tools ("linters") to maintain consistent style and remove the burden of adhering to these standards by hand. Currently, two are employed: 1. Black, the uncompromising code formatter, automatically formats code with a consistent style. 1. isort, sorts imports

There is a step in CI that uses these tools to check for a consistent style. These checks will use the most recent versions of each linter. To ensure that changes follow these standards, you can install and run these tools locally:

```
$ conda install black isort -c conda-forge
$ black openforcefield
$ isort openforcefield
```

Anything not covered above is currently up to personal preference, but may change as new linters are added.

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.

1.6.5 My conda installation of the toolkit doesn't appear to work. What should I try next?

We recommend that you install the toolkit in a fresh conda environment, explicitly passing the channels to be used, in-order:

```
conda create -n <my_new_env> -c conda-forge -c omnia openforcefield
conda activate <my_new_env>
```

Installing into a new environment avoids forcing conda to satisfy the dependencies of both the toolkit and all existing packages in that environment. Taking the approach that conda environments are generally disposable, even ephemeral, minimizes the chances for hard-to-diagnose dependency issues.

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=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*, *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

amber_impropers Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom first in each improper.

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 improper 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 possible 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_rings Return the number of rings found in the Molecule

n_virtual_particles The number of VirtualParticle objects.

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

rings Return the number of rings in this molecule.

smirnoff_impropers Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom second in each improper.

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

apply_elf_conformer_selection([percentage, ...])	Applies the ELF method to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from a molecules conformers.
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([max_states])	Enumerate the formal charges of a molecule to generate different protomoers.

continues on next page

Table 2 – continued from previous page

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(iupac_name[, toolkit_registry, ...])	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(oemol[, allow_undefined_stereo])	Create a Molecule from an OpenEye molecule.
from_pdb_and_smiles(file_path, smiles[, ...])	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 molecule record or dataset entry based on attached cmiles information.
from_rdkit(rdmol[, allow_undefined_stereo])	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 OpenFF 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

continues on next page

Table 2 – continued from previous page

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
strip_atom_stereochemistry(smarts[, ...])	Delete stereochemistry information for certain atoms, if it is present.
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([toolkit_registry])	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([aromaticity_model])	Create an OpenEye molecule
to_pickle()	Return a pickle serialized representation.
to_qcschema([multiplicity, conformer, extras])	Create a QCElemental Molecule.
to_rdkit([aromaticity_model])	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 OpenFF 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=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*, *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
`apply_elf_conformer_selection`([percentage, ...])	Applies the ELF method to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from a molecules conformers.
`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`([max_states])	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`(iupac_name[, toolkit_registry, ...])	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`(oemol[, allow_undefined_stereo])	Create a Molecule from an OpenEye molecule.

continues on next page

Table 3 – continued from previous page

from_pdb_and_smiles(file_path, smiles[, ...])	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 molecule record or dataset entry based on attached cmiles information.
from_rdkit(rdmol[, allow_undefined_stereo])	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 OpenFF 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
strip_atom_stereochemistry(smarts[, ...])	Delete stereochemistry information for certain atoms, if it is present.
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([toolkit_registry])	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([aromaticity_model])	Create an OpenEye molecule
to_pickle()	Return a pickle serialized representation.

continues on next page

Table 3 – continued from previous page

to_qcschema([multiplicity, conformer, extras])	Create a QCElemental Molecule.
to_rdkit([aromaticity_model])	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 OpenFF Topology representation containing one copy of this molecule
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

Attributes

amber_impropers	Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom first in each improper.
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 improper 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 possible 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_rings	Return the number of rings found in the Molecule
n_virtual_particles	The number of VirtualParticle objects.
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
rings	Return the number of rings in this molecule.
smirnoff_impropers	Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom second in each improper.
torsions	Get an iterator over all i-j-k-l torsions.

continues on next page

Table 4 – continued from previous page

total_charge	Return the total charge on the molecule
virtual_sites	Iterate over all VirtualSite objects.

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

generate_unique_atom_names()
Generate unique atom names using element name and number of times that element has occurred e.g. 'C1', 'H1', 'O1', 'C2', ...

strip_atom_stereochemistry(*smarts*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)
Delete stereochemistry information for certain atoms, if it is present. This method can be used to "normalize" molecules imported from different cheminformatics toolkits, which differ in which atom centers are considered stereogenic.

Parameters

smarts: str or ChemicalEnvironment Tagged SMARTS with a single atom with index 1. Any matches for this atom will have any assigned stereocheistry information removed.

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

to_dict()
Return a dictionary representation of the molecule.

Returns

molecule_dict [OrderedDict] A dictionary representation of the molecule.

classmethod from_dict(*molecule_dict*)
Create a new Molecule from a dictionary representation

Parameters

molecule_dict [OrderedDict] A dictionary representation of the molecule.

Returns

molecule [Molecule] A Molecule created from the dictionary representation

to_smiles(*isomeric=True*, *explicit_hydrogens=True*, *mapped=False*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)
Return a canonical isomeric SMILES representation of the current molecule. A partially mapped smiles can also be generated for atoms of interest by supplying an *atom_map* to the properties dictionary.

Note: RDKit and OpenEye versions will not necessarily return the same representation.

Parameters

isomeric: bool optional, default= True return an isomeric smiles

explicit_hydrogens: bool optional, default=True return a smiles string containing all hydrogens explicitly

mapped: bool optional, default=False return a explicit hydrogen mapped smiles, the atoms to be mapped can be controlled by supplying an atom map into the properties dictionary. If no mapping is passed all atoms will be mapped in order, else an atom map dictionary from the current atom index to the map id should be supplied with no duplicates. The map ids (values) should start from 0 or 1.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES conversion

Returns

smiles [str] Canonical isomeric explicit-hydrogen SMILES

Examples

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

classmethod from_inchi(*inchi*, *allow_undefined_stereo=False*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Construct a Molecule from a InChI representation

Parameters

inchi [str] The InChI representation of the molecule.

allow_undefined_stereo [bool, default=False] Whether to accept InChI with undefined stereochemistry. If False, an exception will be raised if a InChI with undefined stereochemistry is passed into this function.

toolkit_registry [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for InChI-to-molecule conversion

Returns

molecule [openforcefield.topology.Molecule]

Examples

make cis-1,2-Dichloroethene >>> molecule = Molecule.from_inchi('InChI=1S/C2H2Cl2/c3-1-2-4/h1-2H/b2-1-')

to_inchi(*fixed_hydrogens=False*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Create an InChI string for the molecule using the requested toolkit backend. InChI is a standardised representation that does not capture tautomers unless specified using the fixed hydrogen layer.

For information on InChi see here https://iupac.org/who-we-are/divisions/division-details/inchi/

Parameters

fixed_hydrogens: bool, default=False If a fixed hydrogen layer should be added to the InChI, if *True* this will produce a non standard specific InChI string of the molecule.

toolkit_registry [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for molecule-to-InChI conversion

Returns

inchi: str The InChI string of the molecule.

Raises

InvalidToolkitRegistryError If an invalid object is passed as the toolkit_registry parameter

to_inchikey(*fixed_hydrogens=False*, *toolkit_registry=ToolkitRegistry containing The RDKit*, *AmberTools*, *Built-in Toolkit*)

Create an InChIKey for the molecule using the requested toolkit backend. InChIKey is a standardised representation that does not capture tautomers unless specified using the fixed hydrogen layer.

For information on InChi see here https://iupac.org/who-we-are/divisions/division-details/inchi/

Parameters

fixed_hydrogens: bool, default=False If a fixed hydrogen layer should be added to the InChI, if *True* this will produce a non standard specific InChI string of the molecule.

toolkit_registry [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for molecule-to-InChIKey conversion

Returns

inchi_key: str The InChIKey representation of the molecule.

Raises

InvalidToolkitRegistryError If an invalid object is passed as the toolkit_registry parameter

classmethod from_smiles(*smiles*, *hydrogens_are_explicit=False*, *toolkit_registry=ToolkitRegistry containing The RDKit*, *AmberTools*, *Built-in Toolkit*, *allow_undefined_stereo=False*)

Construct a Molecule from a SMILES representation

Parameters

smiles [str] The SMILES representation of the molecule.

hydrogens_are_explicit [bool, default = False] If False, the cheminformatics toolkit will perform hydrogen addition

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion

allow_undefined_stereo [bool, default=False] Whether to accept SMILES with undefined stereochemistry. If False, an exception will be raised if a SMILES with undefined stereochemistry is passed into this function.

Returns

molecule [openforcefield.topology.Molecule]

Examples

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

static are_isomorphic(*mol1*, *mol2*, *return_atom_map=False*, *aromatic_matching=True*, *formal_charge_matching=True*, *bond_order_matching=True*, *atom_stereochemistry_matching=True*, *bond_stereochemistry_matching=True*, *strip_pyrimidal_n_atom_stereo=True*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Determines whether the two molecules are isomorphic by comparing their graph representations and the chosen node/edge attributes. Minimally connections and atomic_number are checked.

If nx.Graphs() are given they must at least have atomic_number attributes on nodes. other optional attributes for nodes are: is_aromatic, formal_charge and stereochemistry. optional attributes for edges are: is_aromatic, bond_order and stereochemistry.

Warning: This API is experimental and subject to change.

Parameters

mol1 [an openforcefield.topology.molecule.FrozenMolecule or TopologyMolecule or nx.Graph()]

mol2 [an openforcefield.topology.molecule.FrozenMolecule or TopologyMolecule or nx.Graph()] The molecule to test for isomorphism.

return_atom_map: bool, default=False, optional will return an optional dict containing the atomic mapping.

aromatic_matching: bool, default=True, optional compare the aromatic attributes of bonds and atoms.

formal_charge_matching: bool, default=True, optional compare the formal charges attributes of the atoms.

bond_order_matching: bool, deafult=True, optional compare the bond order on attributes of the bonds.

atom_stereochemistry_matching [bool, default=True, optional] If `False`, atoms' stereochemistry is ignored for the purpose of determining equality.

bond_stereochemistry_matching [bool, default=True, optional] If `False`, bonds' stereochemistry is ignored for the purpose of determining equality.

strip_pyrimidal_n_atom_stereo: bool, default=True, optional If `True`, any stereochemistry defined around pyrimidal nitrogen stereocenters will be disregarded in the isomorphism check.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for removing stereochemistry from pyrimidal nitrogens.

Returns

molecules_are_isomorphic [bool]

atom_map [default=None, Optional,] [Dict[int,int]] ordered by mol1 indexing {mol1_index: mol2_index} If molecules are not isomorphic given input arguments, will return None instead of dict.

`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(). Full matching is done using the options described bellow.

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

Parameters

other: openforcefield.topology.Molecule or TopologyMolecule or nx.Graph()

return_atom_map: bool, default=False, optional will return an optional dict containing the atomic mapping.

aromatic_matching: bool, default=True, optional

compare the aromatic attributes of bonds and atoms.

formal_charge_matching: bool, default=True, optional

compare the formal charges attributes of the atoms.

bond_order_matching: bool, deafult=True, optional

compare the bond order on attributes of the bonds.

atom_stereochemistry_matching [bool, default=True, optional] If `False`, atoms' stereochemistry is ignored for the purpose of determining equality.

bond_stereochemistry_matching [bool, default=True, optional] If `False`, bonds' stereochemistry is ignored for the purpose of determining equality.

strip_pyrimidal_n_atom_stereo: bool, default=True, optional If `True`, any stereochemistry defined around pyrimidal nitrogen stereocenters will be disregarded in the isomorphism check.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for removing stereochemistry from pyrimidal nitrogens.

Returns

isomorphic [bool]

`generate_conformers`(*toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*, *n_conformers=10*, *rms_cutoff=None*, *clear_existing=True*)

Generate conformers for this molecule using an underlying toolkit. If n_conformers=0, no toolkit wrapper will be called. If n_conformers=0 and clear_existing=True, molecule.conformers will be set to None.

Parameters

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion

n_conformers [int, default=1] The maximum number of conformers to produce

rms_cutoff [simtk.Quantity-wrapped float, in units of distance, optional, default=None] The minimum RMS value at which two conformers are considered redundant and one is deleted. Precise implementation of this cutoff may be toolkit-dependent. If None, the cutoff is set to be the default value for each ToolkitWrapper (generally 1 Angstrom).

clear_existing [bool, default=True] Whether to overwrite existing conformers for the molecule

Raises

InvalidToolkitRegistryError If an invalid object is passed as the toolkit_registry parameter

Examples

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.generate_conformers()
```

apply_elf_conformer_selection(*percentage: float = 2.0*, *limit: int = 10*, *toolkit_registry: Optional[Union[openforcefield.utils.toolkits.ToolkitRegistry, openforcefield.utils.toolkits.ToolkitWrapper]] = ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*, ***kwargs*)

Applies the ELF method to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from a molecules conformers.

Parameters

toolkit_registry The underlying toolkit to use to select the ELF conformers.

percentage The percentage of conformers with the lowest electrostatic interaction energies to greedily select from.

limit The maximum number of conformers to select.

See also:

OpenEyeToolkitWrapper.apply_elf_conformer_selection

RDKitToolkitWrapper.apply_elf_conformer_selection

Notes

- The input molecule should have a large set of conformers already generated to select the ELF conformers from.
- The selected conformers will be retained in the *conformers* list while unselected conformers will be discarded.

compute_partial_charges_am1bcc(*use_conformers=None*, *strict_n_conformers=False*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

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.

Parameters

strict_n_conformers [bool, default=False] Whether to raise an exception if an invalid number of conformers is provided for the given charge method. If this is False and an invalid number of conformers is found, a warning will be raised.

use_conformers [iterable of simtk.unit.Quantity-wrapped numpy arrays, each with shape (n_atoms, 3) and dimension of distance. Optional, default=None] Coordinates to use for partial charge calculation. If None, an appropriate number of conformers for the given charge method will be generated.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for the calculation

Raises

InvalidToolkitRegistryError If an invalid object is passed as the toolkit_registry parameter

Examples

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.generate_conformers()
>>> molecule.compute_partial_charges_am1bcc()
```

assign_partial_charges(*partial_charge_method*, *strict_n_conformers=False*, *use_conformers=None*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Calculate partial atomic charges for this molecule using an underlying toolkit, and assign the new values to the partial_charges attribute.

Parameters

partial_charge_method [string] The partial charge calculation method to use for partial charge calculation.

strict_n_conformers [bool, default=False] Whether to raise an exception if an invalid number of conformers is provided for the given charge method. If this is False and an invalid number of conformers is found, a warning will be raised.

use_conformers [iterable of simtk.unit.Quantity-wrapped numpy arrays, each with shape (n_atoms, 3) and dimension of distance. Optional, default=None] Coordinates to use for partial charge calculation. If None, an appropriate number of conformers will be generated.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for the calculation.

Raises

InvalidToolkitRegistryError If an invalid object is passed as the toolkit_registry parameter

Examples

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.assign_partial_charges('am1-mulliken')
```

assign_fractional_bond_orders(*bond_order_model=None*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*, *use_conformers=None*)

Update and store list of bond orders this molecule. Bond orders are stored on each bond, in the *bond.fractional_bond_order* attribute.

Warning: This API is experimental and subject to change.

Parameters

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion

bond_order_model [string, optional. Default=None] The bond order model to use for fractional bond order calculation. If None, "am1-wiberg" will be used.

use_conformers [iterable of simtk.unit.Quantity(np.array) with shape (n_atoms, 3) and dimension of distance, optional, default=None] The conformers to use for fractional bond order calculation. If None, an appropriate number of conformers will be generated by an available ToolkitWrapper.

Raises

InvalidToolkitRegistryError If an invalid object is passed as the toolkit_registry parameter

Examples

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.assign_fractional_bond_orders()
```

to_networkx()

Generate a NetworkX undirected graph from the Molecule.

Nodes are Atoms labeled with particle indices and atomic elements (via the `element` node attribute). Edges denote chemical bonds between Atoms. Virtual sites are not included, since they lack a concept of chemical connectivity.

Returns

graph [networkx.Graph] The resulting graph, with nodes (atoms) labeled with atom indices, elements, stereochemistry and aromaticity flags and bonds with two atom indices, bond order, stereochemistry, and aromaticity flags

Examples

Retrieve the bond graph for imatinib (OpenEye toolkit required)

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

find_rotatable_bonds(*ignore_functional_groups=None*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Find all bonds classed as rotatable ignoring any matched to the `ignore_functional_groups` list.

Parameters

ignore_functional_groups: optional, List[str], default=None, A list of bond SMARTS patterns to be ignored when finding rotatable bonds.

toolkit_registry: openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitW
`ToolkitRegistry` or `ToolkitWrapper` to use for SMARTS matching

Returns

bonds: List[openforcefield.topology.molecule.Bond] The list of openforcefield.topology.molecule.Bond instances which are rotatable.

property partial_charges

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

Returns

partial_charges [a simtk.unit.Quantity - wrapped numpy array [1 x n_atoms] or None] The partial charges on this Molecule's atoms. Returns None if no charges have been specified.

property n_particles

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

property n_atoms

The number of Atom objects.

property n_virtual_sites

The number of VirtualSite objects.

property n_virtual_particles

The number of VirtualParticle objects.

property n_bonds

The number of Bond objects.

property n_angles

int: number of angles in the Molecule.

property n_propers

int: number of proper torsions in the Molecule.

property n_impropers

int: number of possible improper torsions in the Molecule.

property n_rings

Return the number of rings found in the Molecule

Requires the RDKit to be installed.

Note: For systems containing some special cases of connected rings, this function may not be well-behaved and may report a different number rings than expected. Some problematic cases include networks of many (5+) rings or bicyclic moieties (i.e. norbornane).

property particles
Iterate over all Particle objects.

property atoms
Iterate over all Atom objects.

property conformers
Returns the list of conformers for this molecule. This returns a list of simtk.unit.Quantity-wrapped numpy arrays, of shape (3 x n_atoms) and with dimensions of distance. The return value is the actual list of conformers, and changes to the contents affect the original FrozenMolecule.

property n_conformers
Returns the number of conformers for this molecule.

property virtual_sites
Iterate over all VirtualSite objects.

property bonds
Iterate over all Bond objects.

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

property torsions
Get an iterator over all i-j-k-l torsions. Note that i-j-k-i torsions (cycles) are excluded.

Returns

torsions [iterable of 4-Atom tuples]

property propers
Iterate over all proper torsions in the molecule

property impropers
Iterate over all improper torsions in the molecule.

Returns

impropers [set of tuple] An iterator of tuples, each containing the indices of atoms making up a possible improper torsion.

See also:

`smirnoff_impropers`, `amber_impropers`

property smirnoff_impropers
Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom second in each improper.

Note that it's possible that a trivalent center will not have an improper assigned. This will depend on the force field that is used.

Also note that this will return 6 possible atom orderings around each improper center. In current SMIRNOFF parameterization, three of these six orderings will be used for the actual assignment of the improper term and measurement of the angles. These three orderings capture the three

unique angles that could be calculated around the improper center, therefore the sum of these three terms will always return a consistent energy.

The exact three orderings that will be applied during parameterization can not be determined in this method, since it requires sorting the particle indices, and those indices may change when this molecule is added to a Topology.

For more details on the use of three-fold ('trefoil') impropers, see https://open-forcefield-toolkit.readthedocs.io/en/latest/smirnoff.html#impropertorsions

Returns

impropers [set of tuple] An iterator of tuples, each containing the indices of atoms making up a possible improper torsion. The central atom is listed second in each tuple.

See also:

`impropers`, `amber_impropers`

property amber_impropers

Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom first in each improper.

Note that it's possible that a trivalent center will not have an improper assigned. This will depend on the force field that is used.

Also note that this will return 6 possible atom orderings around each improper center. In current AMBER parameterization, one of these six orderings will be used for the actual assignment of the improper term and measurement of the angle. This method does not encode the logic to determine which of the six orderings AMBER would use.

Returns

impropers [set of tuple] An iterator of tuples, each containing the indices of atoms making up a possible improper torsion. The central atom is listed first in each tuple.

See also:

`impropers`, `smirnoff_impropers`

property total_charge

Return the total charge on the molecule

property name

The name (or title) of the molecule

property properties

The properties dictionary of the molecule

property hill_formula

Get the Hill formula of the molecule

static to_hill_formula(*molecule*)

Generate the Hill formula from either a FrozenMolecule, TopologyMolecule or nx.Graph() of the molecule

Parameters

molecule [FrozenMolecule, TopologyMolecule or nx.Graph()]

Returns

formula [the Hill formula of the molecule]

Raises

NotImplementedError [if the molecule is not of one of the specified types.]

chemical_environment_matches(*query*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Retrieve all matches for a given chemical environment query.

Parameters

query [str or ChemicalEnvironment] SMARTS string (with one or more tagged atoms) or `ChemicalEnvironment` query Query will internally be resolved to SMIRKS using `query.asSMIRKS()` if it has an `.asSMIRKS` method.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=GLOBAL_TOOLKIT_REGISTRY] `ToolkitRegistry` or `ToolkitWrapper` to use for chemical environment matches

Returns

matches [list of atom index tuples] A list of tuples, containing the indices of the matching atoms.

Examples

Retrieve all the carbon-carbon bond matches in a molecule

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> matches = molecule.chemical_environment_matches('[#6X3:1]~[#6X3:2]')
```

classmethod from_iupac(*iupac_name*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*, *allow_undefined_stereo=False*, ***kwargs*)

Generate a molecule from IUPAC or common name

Parameters

iupac_name [str] IUPAC name of molecule to be generated

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=GLOBAL_TOOLKIT_REGISTRY] `ToolkitRegistry` or `ToolkitWrapper` to use for chemical environment matches

allow_undefined_stereo [bool, default=False] If false, raises an exception if molecule contains undefined stereochemistry.

Returns

molecule [Molecule] The resulting molecule with position

Note: This method requires the OpenEye toolkit to be installed. ..

Examples

Create a molecule from an IUPAC name

```
>>> molecule = Molecule.from_iupac('4-[(4-methylpiperazin-1-yl)methyl]-N-(4-methyl-3-{[4-
↪(pyridin-3-yl)pyrimidin-2-yl]amino}phenyl)benzamide')
```

Create a molecule from a common name

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

to_iupac(*toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)
Generate IUPAC name from Molecule

Returns

iupac_name [str] IUPAC name of the molecule

Note: This method requires the OpenEye toolkit to be installed. ..

Examples

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

classmethod from_topology(*topology*)
Return a Molecule representation of an OpenFF Topology containing a single Molecule object.

Parameters

topology [openforcefield.topology.Topology] The `Topology` object containing a single `Molecule` object. Note that OpenMM and MDTraj `Topology` objects are not supported.

Returns

molecule [openforcefield.topology.Molecule] The Molecule object in the topology

Raises

ValueError If the topology does not contain exactly one molecule.

Examples

Create a molecule from a Topology object that contains exactly one molecule

```
>>> molecule = Molecule.from_topology(topology)
```

to_topology()
Return an OpenFF Topology representation containing one copy of this molecule

Returns

topology [openforcefield.topology.Topology] A Topology representation of this molecule

Examples

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

classmethod from_file(*file_path*, *file_format=None*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*, *allow_undefined_stereo=False*)

Create one or more molecules from a file

Parameters

file_path [str or file-like object] The path to the file or file-like object to stream one or more molecules from.

file_format [str, optional, default=None] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for your loaded toolkits for details.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper,]

optional, default=GLOBAL_TOOLKIT_REGISTRY `ToolkitRegistry` or `ToolkitWrapper` to use for file loading. If a Toolkit is passed, only the highest-precedence toolkit is used

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

Returns

molecules [Molecule or list of Molecules] If there is a single molecule in the file, a Molecule is returned; otherwise, a list of Molecule objects is returned.

Examples

```
>>> from openforcefield.tests.utils import get_monomer_mol2_file_path
>>> mol2_file_path = get_monomer_mol2_file_path('cyclohexane')
>>> molecule = Molecule.from_file(mol2_file_path)
```

to_file(*file_path*, *file_format*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Write the current molecule to a file or file-like object

Parameters

file_path [str or file-like object] A file-like object or the path to the file to be written.

file_format [str] Format specifier, one of ['MOL2', 'MOL2H', 'SDF', 'PDB', 'SMI', 'CAN', 'TDT'] Note that not all toolkits support all formats

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper,]

optional, default=GLOBAL_TOOLKIT_REGISTRY `ToolkitRegistry` or `ToolkitWrapper` to use for file writing. If a Toolkit is passed, only the highest-precedence toolkit is used

Raises

ValueError If the requested file_format is not supported by one of the installed chemInformatics toolkits

Examples

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> molecule.to_file('imatinib.mol2', file_format='mol2')
>>> molecule.to_file('imatinib.sdf', file_format='sdf')
>>> molecule.to_file('imatinib.pdb', file_format='pdb')
```

enumerate_tautomers(*max_states=20*, *toolkit_registry=ToolkitRegistry containing The RDKit*, *AmberTools*, *Built-in Toolkit*)

Enumerate the possible tautomers of the current molecule

Parameters

max_states: int optional, default=20 The maximum amount of molecules that should be returned

toolkit_registry: openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitW

optional, default=GLOBAL_TOOLKIT_REGISTRY *ToolkitRegistry* or `ToolkitWrapper` to use to enumerate the tautomers.

Returns

molecules: List[openforcefield.topology.Molecule] A list of openforcefield.topology.Molecule instances not including the input molecule.

enumerate_stereoisomers(*undefined_only=False*, *max_isomers=20*, *rationalise=True*, *toolkit_registry=ToolkitRegistry containing The RDKit*, *AmberTools*, *Built-in Toolkit*)

Enumerate the stereocenters and bonds of the current molecule.

Parameters

undefined_only: bool optional, default=False If we should enumerate all stereocenters and bonds or only those with undefined stereochemistry

max_isomers: int optional, default=20 The maximum amount of molecules that should be returned

rationalise: bool optional, default=True If we should try to build and rationalise the molecule to ensure it can exist

toolkit_registry: openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitW

optional, default=GLOBAL_TOOLKIT_REGISTRY *ToolkitRegistry* or `ToolkitWrapper` to use to enumerate the stereoisomers.

Returns

molecules: List[openforcefield.topology.Molecule] A list of openforcefield.topology.Molecule instances not including the input molecule.

enumerate_protomers(*max_states=10*)

Enumerate the formal charges of a molecule to generate different protomoers.

Parameters

max_states: int optional, default=10, The maximum number of protomer states to be returned.

Returns

molecules: List[openforcefield.topology.Molecule], A list of the protomers of the input molecules not including the input.

classmethod from_rdkit(*rdmol*, *allow_undefined_stereo=False*)

Create a Molecule from an RDKit molecule.

Requires the RDKit to be installed.

Parameters

rdmol [rkit.RDMol] An RDKit molecule

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

Returns

molecule [openforcefield.topology.Molecule] An OpenFF molecule

Examples

Create a molecule from an RDKit molecule

```
>>> from rdkit import Chem
>>> from openforcefield.tests.utils import get_data_file_path
>>> rdmol = Chem.MolFromMolFile(get_data_file_path('systems/monomers/ethanol.sdf'))
>>> molecule = Molecule.from_rdkit(rdmol)
```

to_rdkit(*aromaticity_model='OEAroModel_MDL'*)

Create an RDKit molecule

Requires the RDKit to be installed.

Parameters

aromaticity_model [str, optional, default=DEFAULT_AROMATICITY_MODEL] The aromaticity model to use

Returns

rdmol [rkit.RDMol] An RDKit molecule

Examples

Convert a molecule to RDKit

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

classmethod from_openeye(*oemol*, *allow_undefined_stereo=False*)

Create a Molecule from an OpenEye molecule.

Requires the OpenEye toolkit to be installed.

Parameters

oemol [openeye.oechem.OEMol] An OpenEye molecule

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

Returns

molecule [openforcefield.topology.Molecule] An OpenFF molecule

Examples

Create a Molecule from an OpenEye OEMol

```
>>> from openeye import oechem
>>> from openforcefield.tests.utils import get_data_file_path
>>> ifs = oechem.oemolistream(get_data_file_path('systems/monomers/ethanol.mol2'))
>>> oemols = list(ifs.GetOEGraphMols())
>>> molecule = Molecule.from_openeye(oemols[0])
```

to_qcschema(*multiplicity=1*, *conformer=0*, *extras=None*)

Create a QCElemental Molecule.

Warning: This API is experimental and subject to change.

Parameters

multiplicity [int, default=1,] The multiplicity of the molecule; sets *molecular_multiplicity* field for QCElemental Molecule.

conformer [int, default=0,] The index of the conformer to use for the QCElemental Molecule geometry.

extras [dict, default=None] A dictionary that should be included in the *extras* field on the QCElemental Molecule. This can be used to include extra information, such as a smiles representation.

Returns

qcelemental.models.Molecule A validated QCElemental Molecule.

Raises

MissingDependencyError [qcelemental is not installed, the qcschema can not be validated.]

InvalidConformerError [no conformer found at the given index.]

Examples

Create a QCElemental Molecule:

```
>>> import qcelemental as qcel
>>> mol = Molecule.from_smiles('CC')
>>> mol.generate_conformers(n_conformers=1)
>>> qcemol = mol.to_qcschema()
```

classmethod from_mapped_smiles(*mapped_smiles*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit, allow_undefined_stereo=False*)

Create an openforcefield.topology.molecule.Molecule from a mapped SMILES made with cmiles. The molecule will be in the order of the indexing in the mapped smiles string.

Warning: This API is experimental and subject to change.

Parameters

mapped_smiles: str, A CMILES-style mapped smiles string with explicit hydrogens.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

Returns

offmol [openforcefield.topology.molecule.Molecule] An OpenFF molecule instance.

Raises

SmilesParsingError [if the given SMILES had no indexing picked up by the toolkits.]

`classmethod from_qcschema`(*qca_record*, *client=None*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*, *allow_undefined_stereo=False*)

Create a Molecule from a QCArchive molecule record or dataset entry based on attached cmiles information.

For a molecule record, a conformer will be set from its geometry.

For a dataset entry, if a corresponding client instance is provided, the starting geometry for that entry will be used as a conformer.

A QCElemental Molecule produced from *Molecule.to_qcschema* can be round-tripped through this method to produce a new, valid Molecule.

Parameters

qca_record [dict] A QCArchive molecule record or dataset entry.

client [optional, default=None,] A qcportal.FractalClient instance to use for fetching an initial geometry. Only used if *qca_record* is a dataset entry.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

Returns

molecule [openforcefield.topology.Molecule] An OpenFF molecule instance.

Raises

AttributeError [if the record dict can not be made from a record instance.] if a client is passed, because the client could not retrive the initial molecule.

KeyError [if the dict does not contain the canonical_isomeric_explicit_hydrogen_mapped_smiles.]

InvalidConformerError [silent error, if the conformer could not be attached.]

classmethod from_pdb_and_smiles(*file_path*, *smiles*, *allow_undefined_stereo=False*)
Create a Molecule from a pdb file and a SMILES string using RDKit.

Requires RDKit to be installed.

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

The molecule is created and sanitised based on the SMILES string, we then find a mapping between this molecule and one from the PDB based only on atomic number and connections. The SMILES molecule is then reindex to match the PDB, the conformer is attached and the molecule returned.

Parameters

file_path: str PDB file path

smiles [str] a valid smiles string for the pdb, used for seterochemistry and bond order

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

Returns

molecule [openforcefield.Molecule] An OFFMol instance with ordering the same as used in the PDB file.

Raises

InvalidConformerError [if the SMILES and PDB molecules are not isomorphic.]

canonical_order_atoms(*toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)
Canonical order the atoms in a copy of the molecule using a toolkit, returns a new copy.

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

Parameters

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion

Returns

molecule [openforcefield.topology.Molecule] An new OpenFF style molecule with atoms in the canonical order.

remap(*mapping_dict*, *current_to_new=True*)
Remap all of the indexes in the molecule to match the given mapping dict

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

Parameters

mapping_dict [dict,] A dictionary of the mapping between in the indexes, this should start from 0.

current_to_new [bool, default=True] The dict is {current_index: new_index} if True else {new_index: current_index}

Returns

new_molecule [openforcefield.topology.molecule.Molecule] An openforcefield.Molecule instance with all attributes transferred, in the PDB order.

to_openeye(*aromaticity_model='OEAroModel_MDL'*)

Create an OpenEye molecule

Requires the OpenEye toolkit to be installed.

Parameters

aromaticity_model [str, optional, default=DEFAULT_AROMATICITY_MODEL] The aromaticity model to use

Returns

oemol [openeye.oechem.OEMol] An OpenEye molecule

Examples

Create an OpenEye molecule from a Molecule

```
>>> molecule = Molecule.from_smiles('CC')
>>> oemol = molecule.to_openeye()
```

get_bond_between(*i*, *j*)

Returns the bond between two atoms

Parameters

i, j [int or Atom] Atoms or atom indices to check

Returns

bond [Bond] The bond between i and j.

classmethod from_bson(*serialized*)

Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_json(*serialized*)

Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_messagepack(*serialized*)
Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

classmethod from_toml(*serialized*)
Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_xml(*serialized*)
Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_yaml(*serialized*)
Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

property rings
Return the number of rings in this molecule.

Requires the RDKit to be installed.

Note: For systems containing some special cases of connected rings, this function may not be well-behaved and may report a different number rings than expected. Some problematic cases include networks of many (5+) rings or bicyclic moieties (i.e. norbornane).

---

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

to_pickle()
Return a pickle serialized representation.

Warning: This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

to_toml()
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

to_xml(*indent=2*)
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

to_yaml()
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

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

amber_impropers Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom first in each improper.

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 improper 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 possible 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_rings Return the number of rings found in the Molecule

n_virtual_particles The number of VirtualParticle objects.

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

rings Return the number of rings in this molecule.

smirnoff_impropers Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom second in each improper.

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, ...)	Add a virtual site representing the charge on a bond.
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.
apply_elf_conformer_selection([percentage, ...])	Applies the ELF method to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from a molecules conformers.
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([max_states])	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

continues on next page

Table 5 – continued from previous page

from_inchi(inchi[, allow_undefined_stereo, ...])	Construct a Molecule from a InChI representation
from_iupac(iupac_name[, toolkit_registry, ...])	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(oemol[, allow_undefined_stereo])	Create a Molecule from an OpenEye molecule.
from_pdb_and_smiles(file_path, smiles[, ...])	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 molecule record or dataset entry based on attached cmiles information.
from_rdkit(rdmol[, allow_undefined_stereo])	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 OpenFF 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
strip_atom_stereochemistry(smarts[, ...])	Delete stereochemistry information for certain atoms, if it is present.
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

continues on next page

Table 5 – continued from previous page

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([toolkit_registry])	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([aromaticity_model])	Create an OpenEye molecule
to_pickle()	Return a pickle serialized representation.
to_qcschema([multiplicity, conformer, extras])	Create a QCElemental Molecule.
to_rdkit([aromaticity_model])	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 OpenFF 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, ...)	Add a virtual site representing the charge on a bond.
`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.
`apply_elf_conformer_selection`([percentage, ...])	Applies the ELF method to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from a molecules conformers.

continues on next page

Table 6 – continued from previous page

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([max_states])	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(iupac_name[, toolkit_registry, ...])	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(oemol[, allow_undefined_stereo])	Create a Molecule from an OpenEye molecule.
from_pdb_and_smiles(file_path, smiles[, ...])	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 molecule record or dataset entry based on attached cmiles information.
from_rdkit(rdmol[, allow_undefined_stereo])	Create a Molecule from an RDKit molecule.

continues on next page

Table 6 – continued from previous page

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 OpenFF 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
strip_atom_stereochemistry(smarts[, ...])	Delete stereochemistry information for certain atoms, if it is present.
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([toolkit_registry])	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([aromaticity_model])	Create an OpenEye molecule
to_pickle()	Return a pickle serialized representation.
to_qcschema([multiplicity, conformer, extras])	Create a QCElemental Molecule.
to_rdkit([aromaticity_model])	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 OpenFF Topology representation containing one copy of this molecule
to_xml([indent])	Return an XML representation.

continues on next page

Table 6 – continued from previous page

to_yaml()	Return a YAML serialized representation.
visualize([backend, width, height])	Render a visualization of the molecule in Jupyter

Attributes

amber_impropers	Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom first in each improper.
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 improper 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 possible 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_rings	Return the number of rings found in the Molecule
n_virtual_particles	The number of VirtualParticle objects.
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
rings	Return the number of rings in this molecule.
smirnoff_impropers	Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom second in each improper.
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.

property amber_impropers

Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom first in each improper.

Note that it's possible that a trivalent center will not have an improper assigned. This will depend on the force field that is used.

Also note that this will return 6 possible atom orderings around each improper center. In current AMBER parameterization, one of these six orderings will be used for the actual assignment of the improper term and measurement of the angle. This method does not encode the logic to determine which of the six orderings AMBER would use.

Returns

impropers [set of tuple] An iterator of tuples, each containing the indices of atoms making up a possible improper torsion. The central atom is listed first in each tuple.

See also:

`impropers`, `smirnoff_impropers`

property angles

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

apply_elf_conformer_selection(*percentage: float = 2.0*, *limit: int = 10*, *toolkit_registry: Optional[Union[openforcefield.utils.toolkits.ToolkitRegistry, openforcefield.utils.toolkits.ToolkitWrapper]] = ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*, ***kwargs*)

Applies the ELF method to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from a molecules conformers.

Parameters

toolkit_registry The underlying toolkit to use to select the ELF conformers.

percentage The percentage of conformers with the lowest electrostatic interaction energies to greedily select from.

limit The maximum number of conformers to select.

See also:

`OpenEyeToolkitWrapper.apply_elf_conformer_selection`

`RDKitToolkitWrapper.apply_elf_conformer_selection`

Notes

- The input molecule should have a large set of conformers already generated to select the ELF conformers from.
- The selected conformers will be retained in the *conformers* list while unselected conformers will be discarded.

static are_isomorphic(*mol1*, *mol2*, *return_atom_map=False*, *aromatic_matching=True*, *formal_charge_matching=True*, *bond_order_matching=True*, *atom_stereochemistry_matching=True*, *bond_stereochemistry_matching=True*, *strip_pyrimidal_n_atom_stereo=True*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Determines whether the two molecules are isomorphic by comparing their graph representations and the chosen node/edge attributes. Minimally connections and atomic_number are checked.

If nx.Graphs() are given they must at least have atomic_number attributes on nodes. other optional attributes for nodes are: is_aromatic, formal_charge and stereochemistry. optional attributes for edges are: is_aromatic, bond_order and stereochemistry.

Warning: This API is experimental and subject to change.

Parameters

mol1 [an openforcefield.topology.molecule.FrozenMolecule or TopologyMolecule or nx.Graph()]

mol2 [an openforcefield.topology.molecule.FrozenMolecule or TopologyMolecule or nx.Graph()] The molecule to test for isomorphism.

return_atom_map: bool, default=False, optional will return an optional dict containing the atomic mapping.

aromatic_matching: bool, default=True, optional compare the aromatic attributes of bonds and atoms.

formal_charge_matching: bool, default=True, optional compare the formal charges attributes of the atoms.

bond_order_matching: bool, deafult=True, optional compare the bond order on attributes of the bonds.

atom_stereochemistry_matching [bool, default=True, optional] If `False`, atoms' stereochemistry is ignored for the purpose of determining equality.

bond_stereochemistry_matching [bool, default=True, optional] If `False`, bonds' stereochemistry is ignored for the purpose of determining equality.

strip_pyrimidal_n_atom_stereo: bool, default=True, optional If `True`, any stereochemistry defined around pyrimidal nitrogen stereocenters will be disregarded in the isomorphism check.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for removing stereochemistry from pyrimidal nitrogens.

Returns

molecules_are_isomorphic [bool]

atom_map [default=None, Optional,] [Dict[int,int]] ordered by mol1 indexing {mol1_index: mol2_index} If molecules are not isomorphic given input arguments, will return None instead of dict.

assign_fractional_bond_orders(*bond_order_model=None*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*, *use_conformers=None*)

Update and store list of bond orders this molecule. Bond orders are stored on each bond, in the *bond.fractional_bond_order* attribute.

Warning: This API is experimental and subject to change.

Parameters

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion

bond_order_model [string, optional. Default=None] The bond order model to use for fractional bond order calculation. If None, "am1-wiberg" will be used.

use_conformers [iterable of simtk.unit.Quantity(np.array) with shape (n_atoms, 3) and dimension of distance, optional, default=None] The conformers to use for fractional bond order calculation. If None, an appropriate number of conformers will be generated by an available ToolkitWrapper.

Raises

InvalidToolkitRegistryError If an invalid object is passed as the toolkit_registry parameter

Examples

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.assign_fractional_bond_orders()
```

assign_partial_charges(*partial_charge_method*, *strict_n_conformers=False*, *use_conformers=None*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Calculate partial atomic charges for this molecule using an underlying toolkit, and assign the new values to the partial_charges attribute.

Parameters

partial_charge_method [string] The partial charge calculation method to use for partial charge calculation.

strict_n_conformers [bool, default=False] Whether to raise an exception if an invalid number of conformers is provided for the given charge method. If this is False and an invalid number of conformers is found, a warning will be raised.

use_conformers [iterable of simtk.unit.Quantity-wrapped numpy arrays, each with shape (n_atoms, 3) and dimension of distance. Optional, default=None] Coordinates to use for partial charge calculation. If None, an appropriate number of conformers will be generated.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for the calculation.

Raises

InvalidToolkitRegistryError If an invalid object is passed as the toolkit_registry parameter

Examples

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.assign_partial_charges('am1-mulliken')
```

property atoms

Iterate over all Atom objects.

property bonds

Iterate over all Bond objects.

canonical_order_atoms(*toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Canonical order the atoms in a copy of the molecule using a toolkit, returns a new copy.

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

Parameters

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion

Returns

molecule [openforcefield.topology.Molecule] An new OpenFF style molecule with atoms in the canonical order.

chemical_environment_matches(*query*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Retrieve all matches for a given chemical environment query.

Parameters

query [str or ChemicalEnvironment] SMARTS string (with one or more tagged atoms) or `ChemicalEnvironment` query Query will internally be resolved to SMIRKS using `query.asSMIRKS()` if it has an `.asSMIRKS` method.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=GLOBAL_TOOLKIT_REGISTRY] `ToolkitRegistry` or `ToolkitWrapper` to use for chemical environment matches

Returns

matches [list of atom index tuples] A list of tuples, containing the indices of the matching atoms.

Examples

Retrieve all the carbon-carbon bond matches in a molecule

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> matches = molecule.chemical_environment_matches('[#6X3:1]~[#6X3:2]')
```

compute_partial_charges_am1bcc(*use_conformers=None*, *strict_n_conformers=False*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

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.

Parameters

strict_n_conformers [bool, default=False] Whether to raise an exception if an invalid number of conformers is provided for the given charge method. If this is False and an invalid number of conformers is found, a warning will be raised.

use_conformers [iterable of simtk.unit.Quantity-wrapped numpy arrays, each with shape (n_atoms, 3) and dimension of distance. Optional, default=None] Coordinates to use for partial charge calculation. If None, an appropriate number of conformers for the given charge method will be generated.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for the calculation

Raises

InvalidToolkitRegistryError If an invalid object is passed as the toolkit_registry parameter

Examples

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.generate_conformers()
>>> molecule.compute_partial_charges_am1bcc()
```

property conformers

Returns the list of conformers for this molecule. This returns a list of simtk.unit.Quantity-wrapped numpy arrays, of shape (3 x n_atoms) and with dimensions of distance. The return value is the actual list of conformers, and changes to the contents affect the original FrozenMolecule.

enumerate_protomers(*max_states=10*)

Enumerate the formal charges of a molecule to generate different protomoers.

Parameters

max_states: int optional, default=10, The maximum number of protomer states to be returned.

Returns

molecules: List[openforcefield.topology.Molecule], A list of the protomers of the input molecules not including the input.

enumerate_stereoisomers(*undefined_only=False*, *max_isomers=20*, *rationalise=True*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Enumerate the stereocenters and bonds of the current molecule.

Parameters

undefined_only: bool optional, default=False If we should enumerate all stereocenters and bonds or only those with undefined stereochemistry

max_isomers: int optional, default=20 The maximum amount of molecules that should be returned

rationalise: bool optional, default=True If we should try to build and rationalise the molecule to ensure it can exist

toolkit_registry: openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitW

optional, default=GLOBAL_TOOLKIT_REGISTRY *ToolkitRegistry* or `ToolkitWrapper` to use to enumerate the stereoisomers.

Returns

molecules: List[openforcefield.topology.Molecule] A list of openforcefield.topology.Molecule instances not including the input molecule.

enumerate_tautomers(*max_states=20*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Enumerate the possible tautomers of the current molecule

Parameters

max_states: int optional, default=20 The maximum amount of molecules that should be returned

toolkit_registry: openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitW

optional, default=GLOBAL_TOOLKIT_REGISTRY *ToolkitRegistry* or `ToolkitWrapper` to use to enumerate the tautomers.

Returns

molecules: List[openforcefield.topology.Molecule] A list of openforcefield.topology.Molecule instances not including the input molecule.

find_rotatable_bonds(*ignore_functional_groups=None*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Find all bonds classed as rotatable ignoring any matched to the `ignore_functional_groups` list.

Parameters

ignore_functional_groups: optional, List[str], default=None, A list of bond SMARTS patterns to be ignored when finding rotatable bonds.

toolkit_registry: openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitW
`ToolkitRegistry` or `ToolkitWrapper` to use for SMARTS matching

Returns

bonds: List[openforcefield.topology.molecule.Bond] The list of openforcefield.topology.molecule.Bond instances which are rotatable.

classmethod from_bson(*serialized*)

Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_dict(*molecule_dict*)

Create a new Molecule from a dictionary representation

Parameters

molecule_dict [OrderedDict] A dictionary representation of the molecule.

Returns

molecule [Molecule] A Molecule created from the dictionary representation

classmethod from_file(*file_path*, *file_format=None*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*, *allow_undefined_stereo=False*)

Create one or more molecules from a file

Parameters

file_path [str or file-like object] The path to the file or file-like object to stream one or more molecules from.

file_format [str, optional, default=None] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for your loaded toolkits for details.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper,]

optional, default=GLOBAL_TOOLKIT_REGISTRY `ToolkitRegistry` or `ToolkitWrapper` to use for file loading. If a Toolkit is passed, only the highest-precedence toolkit is used

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

Returns

molecules [Molecule or list of Molecules] If there is a single molecule in the file, a Molecule is returned; otherwise, a list of Molecule objects is returned.

Examples

```
>>> from openforcefield.tests.utils import get_monomer_mol2_file_path
>>> mol2_file_path = get_monomer_mol2_file_path('cyclohexane')
>>> molecule = Molecule.from_file(mol2_file_path)
```

classmethod from_inchi(*inchi*, *allow_undefined_stereo=False*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Construct a Molecule from a InChI representation

Parameters

inchi [str] The InChI representation of the molecule.

allow_undefined_stereo [bool, default=False] Whether to accept InChI with undefined stereochemistry. If False, an exception will be raised if a InChI with undefined stereochemistry is passed into this function.

toolkit_registry [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for InChI-to-molecule conversion

Returns

molecule [openforcefield.topology.Molecule]

Examples

make cis-1,2-Dichloroethene >>> molecule = Molecule.from_inchi('InChI=1S/C2H2Cl2/c3-1-2-4/h1-2H/b2-1-')

classmethod from_iupac(*iupac_name*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*, *allow_undefined_stereo=False*, ***kwargs*)

Generate a molecule from IUPAC or common name

Parameters

iupac_name [str] IUPAC name of molecule to be generated

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=GLOBAL_TOOLKIT_REGISTRY] `ToolkitRegistry` or `ToolkitWrapper` to use for chemical environment matches

allow_undefined_stereo [bool, default=False] If false, raises an exception if molecule contains undefined stereochemistry.

Returns

molecule [Molecule] The resulting molecule with position

Note: This method requires the OpenEye toolkit to be installed. ..

Examples

Create a molecule from an IUPAC name

```
>>> molecule = Molecule.from_iupac('4-[(4-methylpiperazin-1-yl)methyl]-N-(4-methyl-3-{[4-
↪(pyridin-3-yl)pyrimidin-2-yl]amino}phenyl)benzamide')
```

Create a molecule from a common name

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

classmethod from_json(*serialized*)

Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_mapped_smiles(*mapped_smiles*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*, *allow_undefined_stereo=False*)

Create an openforcefield.topology.molecule.Molecule from a mapped SMILES made with cmiles. The molecule will be in the order of the indexing in the mapped smiles string.

Warning: This API is experimental and subject to change.

Parameters

mapped_smiles: str, A CMILES-style mapped smiles string with explicit hydrogens.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

Returns

offmol [openforcefield.topology.molecule.Molecule] An OpenFF molecule instance.

Raises

SmilesParsingError [if the given SMILES had no indexing picked up by the toolkits.]

classmethod from_messagepack(*serialized*)

Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_openeye(*oemol*, *allow_undefined_stereo=False*)

Create a Molecule from an OpenEye molecule.

Requires the OpenEye toolkit to be installed.

Parameters

oemol [openeye.oechem.OEMol] An OpenEye molecule

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

Returns

molecule [openforcefield.topology.Molecule] An OpenFF molecule

Examples

Create a Molecule from an OpenEye OEMol

```
>>> from openeye import oechem
>>> from openforcefield.tests.utils import get_data_file_path
>>> ifs = oechem.oemolistream(get_data_file_path('systems/monomers/ethanol.mol2'))
>>> oemols = list(ifs.GetOEGraphMols())
>>> molecule = Molecule.from_openeye(oemols[0])
```

classmethod from_pdb_and_smiles(*file_path*, *smiles*, *allow_undefined_stereo=False*)
Create a Molecule from a pdb file and a SMILES string using RDKit.

Requires RDKit to be installed.

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

The molecule is created and sanitised based on the SMILES string, we then find a mapping between this molecule and one from the PDB based only on atomic number and connections. The SMILES molecule is then reindex to match the PDB, the conformer is attached and the molecule returned.

Parameters

file_path: str PDB file path

smiles [str] a valid smiles string for the pdb, used for seterochemistry and bond order

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

Returns

molecule [openforcefield.Molecule] An OFFMol instance with ordering the same as used in the PDB file.

Raises

InvalidConformerError [if the SMILES and PDB molecules are not isomorphic.]

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

classmethod from_qcschema(*qca_record*, *client=None*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*, *allow_undefined_stereo=False*)
Create a Molecule from a QCArchive molecule record or dataset entry based on attached cmiles information.

For a molecule record, a conformer will be set from its geometry.

For a dataset entry, if a corresponding client instance is provided, the starting geometry for that entry will be used as a conformer.

A QCElemental Molecule produced from *Molecule.to_qcschema* can be round-tripped through this method to produce a new, valid Molecule.

Parameters

qca_record [dict] A QCArchive molecule record or dataset entry.

client [optional, default=None,] A qcportal.FractalClient instance to use for fetching an initial geometry. Only used if *qca_record* is a dataset entry.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

Returns

molecule [openforcefield.topology.Molecule] An OpenFF molecule instance.

Raises

AttributeError [if the record dict can not be made from a record instance.] if a client is passed, because the client could not retrive the initial molecule.

KeyError [if the dict does not contain the canonical_isomeric_explicit_hydrogen_mapped_smiles.]

InvalidConformerError [silent error, if the conformer could not be attached.]

classmethod from_rdkit(*rdmol*, *allow_undefined_stereo=False*)

Create a Molecule from an RDKit molecule.

Requires the RDKit to be installed.

Parameters

rdmol [rkit.RDMol] An RDKit molecule

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

Returns

molecule [openforcefield.topology.Molecule] An OpenFF molecule

Examples

Create a molecule from an RDKit molecule

```
>>> from rdkit import Chem
>>> from openforcefield.tests.utils import get_data_file_path
>>> rdmol = Chem.MolFromMolFile(get_data_file_path('systems/monomers/ethanol.sdf'))
>>> molecule = Molecule.from_rdkit(rdmol)
```

classmethod from_smiles(*smiles*, *hydrogens_are_explicit=False*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*, *allow_undefined_stereo=False*)

Construct a Molecule from a SMILES representation

Parameters

smiles [str] The SMILES representation of the molecule.

hydrogens_are_explicit [bool, default = False] If False, the cheminformatics toolkit will perform hydrogen addition

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion

allow_undefined_stereo [bool, default=False] Whether to accept SMILES with undefined stereochemistry. If False, an exception will be raised if a SMILES with undefined stereochemistry is passed into this function.

Returns

molecule [openforcefield.topology.Molecule]

Examples

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

classmethod from_toml(*serialized*)

Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_topology(*topology*)

Return a Molecule representation of an OpenFF Topology containing a single Molecule object.

Parameters

topology [openforcefield.topology.Topology] The `Topology` object containing a single `Molecule` object. Note that OpenMM and MDTraj `Topology` objects are not supported.

Returns

molecule [openforcefield.topology.Molecule] The Molecule object in the topology

Raises

ValueError If the topology does not contain exactly one molecule.

Examples

Create a molecule from a Topology object that contains exactly one molecule

```
>>> molecule = Molecule.from_topology(topology)
```

classmethod from_xml(*serialized*)

Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_yaml(*serialized*)

Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

generate_conformers(*toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit, n_conformers=10, rms_cutoff=None, clear_existing=True*)

Generate conformers for this molecule using an underlying toolkit. If n_conformers=0, no toolkit wrapper will be called. If n_conformers=0 and clear_existing=True, molecule.conformers will be set to None.

Parameters

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion

n_conformers [int, default=1] The maximum number of conformers to produce

rms_cutoff [simtk.Quantity-wrapped float, in units of distance, optional, default=None] The minimum RMS value at which two conformers are considered redundant and one is deleted. Precise implementation of this cutoff may be toolkit-dependent. If None, the cutoff is set to be the default value for each ToolkitWrapper (generally 1 Angstrom).

clear_existing [bool, default=True] Whether to overwrite existing conformers for the molecule

Raises

InvalidToolkitRegistryError If an invalid object is passed as the toolkit_registry parameter

Examples

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.generate_conformers()
```

generate_unique_atom_names()

Generate unique atom names using element name and number of times that element has occurred e.g. 'C1', 'H1', 'O1', 'C2', ...

get_bond_between(*i*, *j*)

Returns the bond between two atoms

Parameters

i, j [int or Atom] Atoms or atom indices to check

Returns

bond [Bond] The bond between i and j.

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

property hill_formula
Get the Hill formula of the molecule

property impropers
Iterate over all improper torsions in the molecule.

Returns

impropers [set of tuple] An iterator of tuples, each containing the indices of atoms making up a possible improper torsion.

See also:

`smirnoff_impropers`, `amber_impropers`

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(). Full matching is done using the options described bellow.

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

Parameters

other: openforcefield.topology.Molecule or TopologyMolecule or nx.Graph()

return_atom_map: bool, default=False, optional will return an optional dict containing the atomic mapping.

aromatic_matching: bool, default=True, optional

compare the aromatic attributes of bonds and atoms.

formal_charge_matching: bool, default=True, optional

compare the formal charges attributes of the atoms.

bond_order_matching: bool, deafult=True, optional

compare the bond order on attributes of the bonds.

atom_stereochemistry_matching [bool, default=True, optional] If `False`, atoms' stereochemistry is ignored for the purpose of determining equality.

bond_stereochemistry_matching [bool, default=True, optional] If `False`, bonds' stereochemistry is ignored for the purpose of determining equality.

strip_pyrimidal_n_atom_stereo: bool, default=True, optional If `True`, any stereochemistry defined around pyrimidal nitrogen stereocenters will be disregarded in the isomorphism check.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for removing stereochemistry from pyrimidal nitrogens.

Returns

isomorphic [bool]

property n_angles
int: number of angles in the Molecule.

property n_atoms
The number of Atom objects.

property n_bonds
The number of Bond objects.

property n_conformers
Returns the number of conformers for this molecule.

property n_impropers
int: number of possible improper torsions in the Molecule.

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

property n_propers
int: number of proper torsions in the Molecule.

property n_rings
Return the number of rings found in the Molecule

Requires the RDKit to be installed.

Note: For systems containing some special cases of connected rings, this function may not be well-behaved and may report a different number rings than expected. Some problematic cases include networks of many (5+) rings or bicyclic moieties (i.e. norbornane).

---

property n_virtual_particles
The number of VirtualParticle objects.

property n_virtual_sites
The number of VirtualSite objects.

property name
The name (or title) of the molecule

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

Returns

partial_charges [a simtk.unit.Quantity - wrapped numpy array [1 x n_atoms] or None] The partial charges on this Molecule's atoms. Returns None if no charges have been specified.

property particles
Iterate over all Particle objects.

property propers
Iterate over all proper torsions in the molecule

property properties
The properties dictionary of the molecule

remap(*mapping_dict*, *current_to_new=True*)
Remap all of the indexes in the molecule to match the given mapping dict

Warning: This API is experimental and subject to change.

Parameters

mapping_dict [dict,] A dictionary of the mapping between in the indexes, this should start from 0.

current_to_new [bool, default=True] The dict is {current_index: new_index} if True else {new_index: current_index}

Returns

new_molecule [openforcefield.topology.molecule.Molecule] An openforcefield.Molecule instance with all attributes transferred, in the PDB order.

property rings

Return the number of rings in this molecule.

Requires the RDKit to be installed.

Note: For systems containing some special cases of connected rings, this function may not be well-behaved and may report a different number rings than expected. Some problematic cases include networks of many (5+) rings or bicyclic moieties (i.e. norbornane).

property smirnoff_impropers

Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom second in each improper.

Note that it's possible that a trivalent center will not have an improper assigned. This will depend on the force field that is used.

Also note that this will return 6 possible atom orderings around each improper center. In current SMIRNOFF parameterization, three of these six orderings will be used for the actual assignment of the improper term and measurement of the angles. These three orderings capture the three unique angles that could be calculated around the improper center, therefore the sum of these three terms will always return a consistent energy.

The exact three orderings that will be applied during parameterization can not be determined in this method, since it requires sorting the particle indices, and those indices may change when this molecule is added to a Topology.

For more details on the use of three-fold ('trefoil') impropers, see https://open-forcefield-toolkit.readthedocs.io/en/latest/smirnoff.html#impropertorsions

Returns

impropers [set of tuple] An iterator of tuples, each containing the indices of atoms making up a possible improper torsion. The central atom is listed second in each tuple.

See also:

impropers, **amber_impropers**

strip_atom_stereochemistry(*smarts*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Delete stereochemistry information for certain atoms, if it is present. This method can be used

to "normalize" molecules imported from different cheminformatics toolkits, which differ in which atom centers are considered stereogenic.

Parameters

smarts: str or ChemicalEnvironment Tagged SMARTS with a single atom with index 1. Any matches for this atom will have any assigned stereocheistry information removed.

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

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

to_dict()
Return a dictionary representation of the molecule.

Returns

molecule_dict [OrderedDict] A dictionary representation of the molecule.

to_file(*file_path*, *file_format*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)
Write the current molecule to a file or file-like object

Parameters

file_path [str or file-like object] A file-like object or the path to the file to be written.

file_format [str] Format specifier, one of ['MOL2', 'MOL2H', 'SDF', 'PDB', 'SMI', 'CAN', 'TDT'] Note that not all toolkits support all formats

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper,]

optional, default=GLOBAL_TOOLKIT_REGISTRY `ToolkitRegistry` or `ToolkitWrapper` to use for file writing. If a Toolkit is passed, only the highest-precedence toolkit is used

Raises

ValueError If the requested file_format is not supported by one of the installed cheminformatics toolkits

Examples

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> molecule.to_file('imatinib.mol2', file_format='mol2')
>>> molecule.to_file('imatinib.sdf', file_format='sdf')
>>> molecule.to_file('imatinib.pdb', file_format='pdb')
```

static to_hill_formula(*molecule*)
Generate the Hill formula from either a FrozenMolecule, TopologyMolecule or nx.Graph() of the molecule

Parameters

molecule [FrozenMolecule, TopologyMolecule or nx.Graph()]

Returns

formula [the Hill formula of the molecule]

Raises

NotImplementedError [if the molecule is not of one of the specified types.]

`to_inchi`(*fixed_hydrogens=False*, *toolkit_registry=ToolkitRegistry containing The RDKit*, *AmberTools*, *Built-in Toolkit*)

Create an InChI string for the molecule using the requested toolkit backend. InChI is a standardised representation that does not capture tautomers unless specified using the fixed hydrogen layer.

For information on InChi see here https://iupac.org/who-we-are/divisions/division-details/inchi/

Parameters

fixed_hydrogens: bool, default=False If a fixed hydrogen layer should be added to the InChI, if *True* this will produce a non standard specific InChI string of the molecule.

toolkit_registry [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for molecule-to-InChI conversion

Returns

inchi: str The InChI string of the molecule.

Raises

InvalidToolkitRegistryError If an invalid object is passed as the toolkit_registry parameter

`to_inchikey`(*fixed_hydrogens=False*, *toolkit_registry=ToolkitRegistry containing The RDKit*, *AmberTools*, *Built-in Toolkit*)

Create an InChIKey for the molecule using the requested toolkit backend. InChIKey is a standardised representation that does not capture tautomers unless specified using the fixed hydrogen layer.

For information on InChi see here https://iupac.org/who-we-are/divisions/division-details/inchi/

Parameters

fixed_hydrogens: bool, default=False If a fixed hydrogen layer should be added to the InChI, if *True* this will produce a non standard specific InChI string of the molecule.

toolkit_registry [openforcefield.utils.toolkits.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for molecule-to-InChIKey conversion

Returns

inchi_key: str The InChIKey representation of the molecule.

Raises

InvalidToolkitRegistryError If an invalid object is passed as the toolkit_registry parameter

to_iupac(*toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)
Generate IUPAC name from Molecule

Returns

iupac_name [str] IUPAC name of the molecule

Note: This method requires the OpenEye toolkit to be installed. ..

Examples

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

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

to_networkx()
Generate a NetworkX undirected graph from the Molecule.

Nodes are Atoms labeled with particle indices and atomic elements (via the `element` node attribute). Edges denote chemical bonds between Atoms. Virtual sites are not included, since they lack a concept of chemical connectivity.

Returns

graph [networkx.Graph] The resulting graph, with nodes (atoms) labeled with atom indices, elements, stereochemistry and aromaticity flags and bonds with two atom indices, bond order, stereochemistry, and aromaticity flags

Examples

Retrieve the bond graph for imatinib (OpenEye toolkit required)

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

to_openeye(*aromaticity_model='OEAroModel_MDL'*)
Create an OpenEye molecule

Requires the OpenEye toolkit to be installed.

Parameters

aromaticity_model [str, optional, default=DEFAULT_AROMATICITY_MODEL] The aromaticity model to use

Returns

oemol [openeye.oechem.OEMol] An OpenEye molecule

Examples

Create an OpenEye molecule from a Molecule

```
>>> molecule = Molecule.from_smiles('CC')
>>> oemol = molecule.to_openeye()
```

to_pickle()
Return a pickle serialized representation.

Warning: This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

to_qcschema(*multiplicity=1*, *conformer=0*, *extras=None*)
Create a QCElemental Molecule.

Warning: This API is experimental and subject to change.

Parameters

multiplicity [int, default=1,] The multiplicity of the molecule; sets *molecular_multiplicity* field for QCElemental Molecule.

conformer [int, default=0,] The index of the conformer to use for the QCElemental Molecule geometry.

extras [dict, default=None] A dictionary that should be included in the *extras* field on the QCElemental Molecule. This can be used to include extra information, such as a smiles representation.

Returns

qcelemental.models.Molecule A validated QCElemental Molecule.

Raises

MissingDependencyError [qcelemental is not installed, the qcschema can not be validated.]

InvalidConformerError [no conformer found at the given index.]

Examples

Create a QCElemental Molecule:

```
>>> import qcelemental as qcel
>>> mol = Molecule.from_smiles('CC')
>>> mol.generate_conformers(n_conformers=1)
>>> qcemol = mol.to_qcschema()
```

to_rdkit(*aromaticity_model='OEAroModel_MDL'*)
Create an RDKit molecule

Requires the RDKit to be installed.

Parameters

aromaticity_model [str, optional, default=DEFAULT_AROMATICITY_MODEL] The aromaticity model to use

Returns

rdmol [rkit.RDMol] An RDKit molecule

Examples

Convert a molecule to RDKit

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

to_smiles(*isomeric=True*, *explicit_hydrogens=True*, *mapped=False*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)
Return a canonical isomeric SMILES representation of the current molecule. A partially mapped smiles can also be generated for atoms of interest by supplying an *atom_map* to the properties dictionary.

Note: RDKit and OpenEye versions will not necessarily return the same representation.

Parameters

isomeric: bool optional, default= True return an isomeric smiles

explicit_hydrogens: bool optional, default=True return a smiles string containing all hydrogens explicitly

mapped: bool optional, default=False return a explicit hydrogen mapped smiles, the atoms to be mapped can be controlled by supplying an atom map into the properties dictionary. If no mapping is passed all atoms will be mapped in order, else an atom map dictionary from the current atom index to the map id should be supplied with no duplicates. The map ids (values) should start from 0 or 1.

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES conversion

Returns

smiles [str] Canonical isomeric explicit-hydrogen SMILES

Examples

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

to_toml()
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

to_topology()
Return an OpenFF Topology representation containing one copy of this molecule

Returns

topology [openforcefield.topology.Topology] A Topology representation of this molecule

Examples

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

to_xml(*indent=2*)
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

to_yaml()
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

property torsions
Get an iterator over all i-j-k-l torsions. Note that i-j-k-i torsions (cycles) are excluded.

Returns

torsions [iterable of 4-Atom tuples]

property total_charge
Return the total charge on the molecule

property virtual_sites
Iterate over all VirtualSite objects.

add_atom(*atomic_number*, *formal_charge*, *is_aromatic*, *stereochemistry=None*, *name=None*)
Add an atom

Parameters

atomic_number [int] Atomic number of the atom

formal_charge [int] 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 if stereochemistry is irrelevant

name [str, optional, default=None] An optional name for the atom

Returns

index [int] The index of the atom in the molecule

Examples

Define a methane molecule

```
>>> molecule = Molecule()
>>> molecule.name = 'methane'
>>> C = molecule.add_atom(6, 0, False)
>>> H1 = molecule.add_atom(1, 0, False)
>>> H2 = molecule.add_atom(1, 0, False)
>>> H3 = molecule.add_atom(1, 0, False)
>>> H4 = molecule.add_atom(1, 0, False)
>>> bond_idx = molecule.add_bond(C, H1, False, 1)
>>> bond_idx = molecule.add_bond(C, H2, False, 1)
>>> bond_idx = molecule.add_bond(C, H3, False, 1)
>>> bond_idx = molecule.add_bond(C, H4, False, 1)
```

add_bond_charge_virtual_site(*atoms*, *distance*, ***kwargs*)
Add a virtual site representing the charge on a bond.

Create a bond charge-type virtual site, in which the location of the charge is specified by the position of two atoms. 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.

Parameters

atoms [list of openforcefield.topology.molecule.Atom objects or ints of shape [N]] The atoms defining the virtual site's position or their indices

distance [float]

weights [list of floats of shape [N] or None, optional, default=None] weights[index] is the weight of particles[index] contributing to the position of the virtual site. Default is None

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.

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

sigma [float, default=None] Sigma 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=''] The name of this virtual site. Default is ''.

symmetric [bool, default=True] Whether to make virtual site symmetric by creating two particles instead of just one. As an example, for N_2 this should be set to True to model both lone pairs with the same parameters.

Returns

index [int] The index of the newly-added virtual site in the molecule

`add_monovalent_lone_pair_virtual_site`(*atoms*, *distance*, *out_of_plane_angle*, *in_plane_angle*, ***kwargs*)

Create a bond charge-type virtual site, in which the location of the charge is specified by the position of three atoms.

Parameters

atoms [list of three openforcefield.topology.molecule.Atom objects or ints] The three atoms defining the virtual site's position or their molecule atom indices

distance [float]

out_of_plane_angle [float]

in_plane_angle [float]

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

sigma [float, default=None] Sigma 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=''] The name of this virtual site. Default is ''.

symmetric [bool, default=False] Whether to make virtual site symmetric by creating two particles instead of just one. Note that because this site is defined is placed on the noncentral atom, setting this to True will place one particle on atom1, and the other on atom3.

Returns

index [int] The index of the newly-added virtual site in the molecule

add_divalent_lone_pair_virtual_site(*atoms*, *distance*, *out_of_plane_angle*, ***kwargs*)
Create a divalent lone pair-type virtual site, in which the location of the charge is specified by the position of three atoms.

Parameters

atoms [list of 3 openforcefield.topology.molecule.Atom objects or ints] The three atoms defining the virtual site's position or their molecule atom indices

distance [float]

out_of_plane_angle [float]

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

sigma [float, default=None] Sigma 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=''] The name of this virtual site. Default is ''.

symmetric [bool, default=True] Whether to make virtual site symmetric by creating two particles instead of just one. As an example, for TIP5 should be set to True to model both lone pairs with the same parameters.

Returns

index [int] The index of the newly-added virtual site in the molecule

add_trivalent_lone_pair_virtual_site(*atoms*, *distance*, ***kwargs*)
Create a trivalent lone pair-type virtual site, in which the location of the charge is specified by the position of four atoms.

Parameters

atoms [list of 4 openforcefield.topology.molecule.Atom objects or ints] The three atoms defining the virtual site's position or their molecule atom indices

distance [float]

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

sigma [float, default=None] Sigma 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=''] The name of this virtual site. Default is ''.

Returns

index [int] The index of the newly-added virtual site in the molecule

add_bond(*atom1*, *atom2*, *bond_order*, *is_aromatic*, *stereochemistry=None*, *fractional_bond_order=None*)
Add a bond between two specified atom indices

Parameters

atom1 [int or openforcefield.topology.molecule.Atom] Index of first atom

atom2 [int or openforcefield.topology.molecule.Atom] Index of second atom

bond_order [int] Integral bond order of Kekulized form

is_aromatic [bool] True if this bond is aromatic, False otherwise

stereochemistry [str, optional, default=None] Either 'E' or 'Z' for specified stereochemistry, or None if stereochemistry is irrelevant

fractional_bond_order [float, optional, default=None] The fractional (eg. Wiberg) bond order

Returns

index: int Index of the bond in this molecule

add_conformer(*coordinates*)
Add a conformation of the molecule

Parameters

coordinates: simtk.unit.Quantity(np.array) with shape (n_atoms, 3) and dimension of distance Coordinates of the new conformer, with the first dimension of the array corresponding to the atom index in the Molecule's indexing system.

Returns

index: int The index of this conformer

visualize(*backend='rdkit'*, *width=500*, *height=300*)
Render a visualization of the molecule in Jupyter

Parameters

backend [str, optional, default='rdkit'] Which visualization engine to use. Choose from: - rdkit - openeye - nglview (conformers needed)

width [int, optional, default=500] Width of the generated representation (only applicable to backend=openeye)

height [int, optional, default=300] Width of the generated representation (only applicable to backend=openeye)

Returns

object Depending on the backend chosen: - rdkit, openeye -> IPython.display.Image - nglview -> nglview.NGLWidget

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

amber_impropers Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom first in each improper.

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 possible improper torsions in this Topology.

is_periodic Return whether or not this Topology is intended to be described with periodic boundary conditions.

n_angles int: number of angles in this Topology.

n_impropers int: number of possible 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.

smirnoff_impropers Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom second in each improper.

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_file(filename, positions[, file_format, ...])	To save a PDB file with coordinates as well as topology from openforcefield topology object Reference: https://github.com/openforcefield/openforcefield/issues/502 Note: 1. This doesn't handle virtual sites (they're ignored) 2. Atom numbering may not remain same, for example if the atoms in water are numbered as 1001, 1002, 1003, they would change to 1, 2, 3. This doesn't affect the topology or coordinates or atom-ordering in anyway 3. Same issue with the amino acid names in the pdb file, they are not returned.
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_file(filename, positions[, file_format, ...])	To save a PDB file with coordinates as well as topology from openforcefield topology object Reference: https://github.com/openforcefield/openforcefield/issues/502 Note: 1. This doesn't handle virtual sites (they're ignored) 2. Atom numbering may not remain same, for example if the atoms in water are numbered as 1001, 1002, 1003, they would change to 1, 2, 3. This doesn't affect the topology or coordinates or atom-ordering in anyway 3. Same issue with the amino acid names in the pdb file, they are not returned.
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

amber_impropers	Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom first in each improper.
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 of shape (3, 3) 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 possible improper torsions in this Topology.
is_periodic	Return whether or not this Topology is intended to be described with periodic boundary conditions.
n_angles	int: number of angles in this Topology.
n_impropers	int: number of possible 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.
smirnoff_impropers	Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom second in each improper.
topology_atoms	Returns an iterator over the atoms in this Topology.

continues on next page

Table 10 – continued from previous page

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

property reference_molecules
Get an iterator of reference molecules in this Topology.

Returns

iterable of openforcefield.topology.Molecule

classmethod from_molecules(*molecules*)
Create a new Topology object containing one copy of each of the specified molecule(s).

Parameters

molecules [Molecule or iterable of Molecules] One or more molecules to be added to the Topology

Returns

topology [Topology] The Topology created from the specified molecule(s)

assert_bonded(*atom1*, *atom2*)
Raise an exception if the specified atoms are not bonded in the topology.

Parameters

atom1, atom2 [openforcefield.topology.Atom or int] The atoms or atom topology indices to check to ensure they are bonded

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

Returns

aromaticity_model [str] Aromaticity model in use.

property box_vectors
Return the box vectors of the topology, if specified Returns ——- box_vectors : simtk.unit.Quantity wrapped numpy array of shape (3, 3)

The unit-wrapped box vectors of this topology

property is_periodic
Return whether or not this Topology is intended to be described with periodic boundary conditions.

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

Returns

charge_model [str] Charge model used for all molecules in the Topology.

property constrained_atom_pairs
Returns the constrained atom pairs of the Topology

Returns

constrained_atom_pairs [dict] dictionary of the form d[(atom1_topology_index, atom2_topology_index)] = distance (float)

property fractional_bond_order_model

Get the fractional bond order model for the Topology.

Returns

fractional_bond_order_model [str] Fractional bond order model in use.

property n_reference_molecules

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

Returns

n_reference_molecules [int]

property n_topology_molecules

Returns the number of topology molecules in in this Topology.

Returns

n_topology_molecules [int]

property topology_molecules

Returns an iterator over all the TopologyMolecules in this Topology

Returns

topology_molecules [Iterable of TopologyMolecule]

property n_topology_atoms

Returns the number of topology atoms in in this Topology.

Returns

n_topology_atoms [int]

property topology_atoms

Returns an iterator over the atoms in this Topology. These will be in ascending order of topology index (Note that this is not necessarily the same as the reference molecule index)

Returns

topology_atoms [Iterable of TopologyAtom]

property n_topology_bonds

Returns the number of TopologyBonds in in this Topology.

Returns

n_bonds [int]

property topology_bonds

Returns an iterator over the bonds in this Topology

Returns

topology_bonds [Iterable of TopologyBond]

property n_topology_particles

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

Returns

n_topology_particles [int]

property topology_particles
Returns an iterator over the particles (TopologyAtoms and TopologyVirtualSites) in this Topology. The TopologyAtoms will be in order of ascending Topology index (Note that this may differ from the order of atoms in the reference molecule index).

Returns

topology_particles [Iterable of TopologyAtom and TopologyVirtualSite]

property n_topology_virtual_sites
Returns the number of TopologyVirtualSites in in this Topology.

Returns

n_virtual_sites [iterable of TopologyVirtualSites]

property topology_virtual_sites
Get an iterator over the virtual sites in this Topology

Returns

topology_virtual_sites [Iterable of TopologyVirtualSite]

property n_angles
int: number of angles in this Topology.

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

property n_propers
int: number of proper torsions in this Topology.

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

property n_impropers
int: number of possible improper torsions in this Topology.

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

property smirnoff_impropers
Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom second in each improper.

Note that it's possible that a trivalent center will not have an improper assigned. This will depend on the force field that is used.

Also note that this will return 6 possible atom orderings around each improper center. In current SMIRNOFF parameterization, three of these six orderings will be used for the actual assignment of the improper term and measurement of the angles. These three orderings capture the three unique angles that could be calculated around the improper center, therefore the sum of these three terms will always return a consistent energy.

For more details on the use of three-fold ('trefoil') impropers, see https://open-forcefield-toolkit.readthedocs.io/en/latest/smirnoff.html#impropertorsions

Returns

impropers [set of tuple] An iterator of tuples, each containing the indices of atoms making up a possible improper torsion. The central atom is listed second in each tuple.

See also:

`impropers`, `amber_impropers`

property amber_impropers

Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom first in each improper.

Note that it's possible that a trivalent center will not have an improper assigned. This will depend on the force field that is used.

Also note that this will return 6 possible atom orderings around each improper center. In current AMBER parameterization, one of these six orderings will be used for the actual assignment of the improper term and measurement of the angle. This method does not encode the logic to determine which of the six orderings AMBER would use.

Returns

impropers [set of tuple] An iterator of tuples, each containing the indices of atoms making up a possible improper torsion. The central atom is listed first in each tuple.

See also:

`impropers`, `smirnoff_impropers`

chemical_environment_matches(*query*, *aromaticity_model='MDL'*, *toolkit_registry=ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit*)

Retrieve all matches for a given chemical environment query.

TODO: * Do we want to generalize this to other kinds of queries too, like mdtraj DSL, pymol selections, atom index slices, etc?

We could just call it topology.matches(query)

Parameters

query [str or ChemicalEnvironment] SMARTS string (with one or more tagged atoms) or `ChemicalEnvironment` query Query will internally be resolved to SMARTS using `query.as_smarts()` if it has an `.as_smarts` method.

aromaticity_model [str] Override the default aromaticity model for this topology and use the specified aromaticity model instead. Allowed values: ['MDL']

Returns

matches [list of Topology._ChemicalEnvironmentMatch] A list of tuples, containing the topology indices of the matching atoms.

to_dict()

Convert to dictionary representation.

classmethod from_dict(*d*)

Static constructor from dictionary representation.

classmethod from_openmm(*openmm_topology*, *unique_molecules=None*)

Construct an openforcefield Topology object from an OpenMM Topology object.

Parameters

openmm_topology [simtk.openmm.app.Topology] An OpenMM Topology object

unique_molecules [iterable of objects that can be used to construct unique Molecule objects] All unique molecules must be provided, in any order, though multiple copies of each molecule are allowed. The atomic elements and bond connectivity will be used to match the reference molecules to molecule graphs appearing in the OpenMM `Topology`. If bond orders are present in the OpenMM `Topology`, these will be used in matching as well.

Returns

topology [openforcefield.topology.Topology] An openforcefield Topology object

to_openmm(*ensure_unique_atom_names=True*)
Create an OpenMM Topology object.

The OpenMM `Topology` object will have one residue per topology molecule. Currently, the number of chains depends on how many copies of the same molecule are in the `Topology`. Molecules with more than 5 copies are all assigned to a single chain, otherwise one chain is created for each molecule. This behavior may change in the future.

Parameters

ensure_unique_atom_names [bool, optional. Default=True] Whether to check that the molecules in each molecule have unique atom names, and regenerate them if not. Note that this looks only at molecules, and does not guarantee uniqueness in the entire Topology.

Returns

openmm_topology [simtk.openmm.app.Topology] An OpenMM Topology object

to_file(*filename*, *positions*, *file_format='PDB'*, *keepIds=False*)
To save a PDB file with coordinates as well as topology from openforcefield topology object Reference: https://github.com/openforcefield/openforcefield/issues/502 Note: 1. This doesn't handle virtual sites (they're ignored)

2. Atom numbering may not remain same, for example if the atoms in water are numbered as 1001, 1002, 1003, they would change to 1, 2, 3. This doesn't affect the topology or coordinates or atom-ordering in anyway

3. Same issue with the amino acid names in the pdb file, they are not returned

Parameters

filename [str] name of the pdb file to write to

positions [n_atoms x 3 numpy array or simtk.unit.Quantity-wrapped n_atoms x 3 iterable] Can be an openmm 'quantity' object which has atomic positions as a list of Vec3s along with associated units, otherwise a 3D array of UNITLESS numbers are considered as "Angstroms" by default

file_format [str] Output file format. Case insensitive. Currently only supported value is "pdb".

static from_mdtraj(*mdtraj_topology*, *unique_molecules=None*)
Construct an openforcefield Topology object from an MDTraj Topology object.

Parameters

mdtraj_topology [mdtraj.Topology] An MDTraj Topology object

unique_molecules [iterable of objects that can be used to construct unique Molecule objects] All unique molecules must be provided, in any order, though multiple

copies of each molecule are allowed. The atomic elements and bond connectivity will be used to match the reference molecules to molecule graphs appearing in the MDTraj `Topology`. If bond orders are present in the MDTraj `Topology`, these will be used in matching as well.

Returns

topology [openforcefield.Topology] An openforcefield Topology object

static from_parmed(*parmed_structure*, *unique_molecules=None*)

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

Construct an openforcefield Topology object from a ParmEd Structure object.

Parameters

parmed_structure [parmed.Structure] A ParmEd structure object

unique_molecules [iterable of objects that can be used to construct unique Molecule objects] All unique molecules must be provided, in any order, though multiple copies of each molecule are allowed. The atomic elements and bond connectivity will be used to match the reference molecules to molecule graphs appearing in the structure's `topology` object. If bond orders are present in the structure's `topology` object, these will be used in matching as well.

Returns

topology [openforcefield.Topology] An openforcefield Topology object

to_parmed()

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

Create a ParmEd Structure object.

Returns

parmed_structure [parmed.Structure] A ParmEd Structure objecft

get_bond_between(*i*, *j*)

Returns the bond between two atoms

Parameters

i, j [int or TopologyAtom] Atoms or atom indices to check

Returns

bond [TopologyBond] The bond between i and j.

is_bonded(*i*, *j*)

Returns True if the two atoms are bonded

Parameters

i, j [int or TopologyAtom] Atoms or atom indices to check

Returns

is_bonded [bool] True if atoms are bonded, False otherwise.

atom(*atom_topology_index*)
Get the TopologyAtom at a given Topology atom index.

Parameters

atom_topology_index [int] The index of the TopologyAtom in this Topology

Returns

An openforcefield.topology.TopologyAtom

virtual_site(*vsite_topology_index*)
Get the TopologyAtom at a given Topology atom index.

Parameters

vsite_topology_index [int] The index of the TopologyVirtualSite in this Topology

Returns

An openforcefield.topology.TopologyVirtualSite

bond(*bond_topology_index*)
Get the TopologyBond at a given Topology bond index.

Parameters

bond_topology_index [int] The index of the TopologyBond in this Topology

Returns

An openforcefield.topology.TopologyBond

classmethod from_bson(*serialized*)
Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_json(*serialized*)
Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_messagepack(*serialized*)
Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

classmethod from_toml(*serialized*)
Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_xml(*serialized*)
Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_yaml(*serialized*)
Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

to_pickle()
Return a pickle serialized representation.

Warning: This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

to_toml()
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

to_xml(*indent=2*)
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

to_yaml()
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

add_particle(*particle*)
Add a Particle to the Topology.

Parameters

particle [Particle] The Particle to be added. The Topology will take ownership of the Particle.

add_molecule(*molecule*, *local_topology_to_reference_index=None*)
Add a Molecule to the Topology. You can optionally request that the atoms be added to the Topology in a different order than they appear in the Molecule.

Parameters

molecule [Molecule] The Molecule to be added.

local_topology_to_reference_index: dict, optional, default = None Dictionary of {TopologyMolecule_atom_index : Molecule_atom_index} for the Topology-Molecule that will be built. If None, this function will add the atoms to the Topology in the order that they appear in the reference molecule.

Returns

index [int] The index of this molecule in the topology

add_constraint(*iatom*, *jatom*, *distance=True*)
Mark a pair of atoms as constrained.

Constraints between atoms that are not bonded (e.g., rigid waters) are permissible.

Parameters

iatom, jatom [Atom] Atoms to mark as constrained These atoms may be bonded or not in the Topology

distance [simtk.unit.Quantity, optional, default=True] Constraint distance `True` if distance has yet to be determined `False` if constraint is to be removed

is_constrained(*iatom*, *jatom*)
Check if a pair of atoms are marked as constrained.

Parameters

iatom, jatom [int] Indices of atoms to mark as constrained.

Returns

distance [simtk.unit.Quantity or bool] True if constrained but constraints have not yet been applied Distance if constraint has already been added to System

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

`amber_impropers` Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom first in each improper.

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

`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 possible improper torsions in this TopologyMolecule.

`n_angles` int: number of angles in this TopologyMolecule.

`n_atoms` The number of atoms in this topology.

`n_bonds` Get the number of bonds in this TopologyMolecule

`n_impropers` int: number of possible improper torsions in this TopologyMolecule.

`n_particles` Get the number of particles in this TopologyMolecule

`n_propers` int: number of proper torsions in this TopologyMolecule.

`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 TopologyMolecule.

`reference_molecule` Get the reference molecule for this TopologyMolecule

`smirnoff_impropers` Note that it's possible that a trivalent center will not have an improper assigned.

`topology` Get the topology that this TopologyMolecule belongs to

`virtual_particle_start_topology_index` Get the topology index of the first virtual particle in this TopologyMolecule

`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 TopologyMolecule 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

amber_impropers	Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom first in each improper.
angles	Iterable of Tuple[TopologyAtom]: iterator over the angles in this TopologyMolecule.
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 possible improper torsions in this TopologyMolecule.
n_angles	int: number of angles in this TopologyMolecule.
n_atoms	The number of atoms in this topology.
n_bonds	Get the number of bonds in this TopologyMolecule
n_impropers	int: number of possible improper torsions in this TopologyMolecule.
n_particles	Get the number of particles in this TopologyMolecule
n_propers	int: number of proper torsions in this TopologyMolecule.
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 TopologyMolecule.
reference_molecule	Get the reference molecule for this TopologyMolecule
smirnoff_impropers	Note that it's possible that a trivalent center will not have an improper assigned.
topology	Get the topology that this TopologyMolecule belongs to
virtual_particle_start_topology_index	Get the topology index of the first virtual particle in this TopologyMolecule
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

property topology

Get the topology that this TopologyMolecule belongs to

Returns

an openforcefield.topology.Topology

property reference_molecule
Get the reference molecule for this TopologyMolecule

Returns

an openforcefield.topology.molecule.Molecule

property n_atoms
The number of atoms in this topology.

Returns

int

atom(*index*)
Get the TopologyAtom with a given topology atom index in this TopologyMolecule.

Parameters

index [int] Index of the TopologyAtom within this TopologyMolecule to retrieve

Returns

an openforcefield.topology.TopologyAtom

property atoms
Return an iterator of all the TopologyAtoms in this TopologyMolecule

Returns

an iterator of openforcefield.topology.TopologyAtoms

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

property virtual_particle_start_topology_index
Get the topology index of the first virtual particle in this TopologyMolecule

bond(*index*)
Get the TopologyBond with a given reference molecule index in this TopologyMolecule

Parameters

index [int] Index of the TopologyBond within this TopologyMolecule to retrieve

Returns

an openforcefield.topology.TopologyBond

property bonds
Return an iterator of all the TopologyBonds in this TopologyMolecule

Returns

an iterator of openforcefield.topology.TopologyBonds

property n_bonds
Get the number of bonds in this TopologyMolecule

Returns

int [number of bonds]

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

particle(*index*)
Get the TopologyParticle with a given reference molecule index in this TopologyMolecule

Parameters

index [int] Index of the TopologyParticle within this TopologyMolecule to retrieve

Returns

an openforcefield.topology.TopologyParticle

`property particles`
Return an iterator of all the TopologyParticles in this TopologyMolecules

Returns

an iterator of openforcefield.topology.TopologyParticle

`property n_particles`
Get the number of particles in this TopologyMolecule

Returns

int [The number of particles]

`virtual_site`(*index*)
Get the TopologyVirtualSite with a given reference molecule index in this TopologyMolecule

Parameters

index [int] Index of the TopologyVirtualSite within this TopologyMolecule to retrieve

Returns

an openforcefield.topology.TopologyVirtualSite

`property virtual_sites`
Return an iterator of all the TopologyVirtualSites in this TopologyMolecules

Returns

an iterator of openforcefield.topology.TopologyVirtualSite

`property n_virtual_sites`
Get the number of virtual sites in this TopologyMolecule

Returns

int

`property angles`
Iterable of Tuple[TopologyAtom]: iterator over the angles in this TopologyMolecule.

`property n_angles`
int: number of angles in this TopologyMolecule.

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

`property n_propers`
int: number of proper torsions in this TopologyMolecule.

`property impropers`
Iterable of Tuple[TopologyAtom]: iterator over the possible improper torsions in this TopologyMolecule.

`property n_impropers`
int: number of possible improper torsions in this TopologyMolecule.

property smirnoff_impropers
Note that it's possible that a trivalent center will not have an improper assigned. This will depend on the force field that is used.

Also note that this will return 6 possible atom orderings around each improper center. In current SMIRNOFF parameterization, three of these six orderings will be used for the actual assignment of the improper term and measurement of the angles. These three orderings capture the three unique angles that could be calculated around the improper center, therefore the sum of these three terms will always return a consistent energy.

For more details on the use of three-fold ('trefoil') impropers, see https://open-forcefield-toolkit.readthedocs.io/en/latest/smirnoff.html#impropertorsions

Returns

impropers [set of tuple] An iterator of tuples, each containing the indices of atoms making up a possible improper torsion. The central atom is listed second in each tuple.

property amber_impropers
Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom first in each improper.

Note that it's possible that a trivalent center will not have an improper assigned. This will depend on the force field that is used.

Also note that this will return 6 possible atom orderings around each improper center. In current AMBER parameterization, one of these six orderings will be used for the actual assignment of the improper term and measurement of the angle. This method does not encode the logic to determine which of the six orderings AMBER would use.

Returns

impropers [set of tuple] An iterator of tuples, each containing the indices of atoms making up a possible improper torsion. The central atom is listed first in each tuple.

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

to_dict()
Convert to dictionary representation.

classmethod from_dict(*d*)
Static constructor from dictionary representation.

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 container representing one or more virtual particles whose positions are defined in terms of Atom positions.
VirtualParticle	A single particle owned by a VirtualSite

continues on next page

Table 14 – continued from previous page

TopologyVirtualParticle	Attributes
BondChargeVirtualSite	A particle representing a “Bond Charge”-type virtual site, in which the location of the charge is specified by the positions of two atoms.
MonovalentLonePairVirtualSite	A particle representing a “Monovalent Lone Pair”-type virtual site, in which the location of the charge is specified by the positions of three atoms.
DivalentLonePairVirtualSite	A particle representing a “Divalent Lone Pair”-type virtual site, in which the location of the charge is specified by the positions of three atoms.
TrivalentLonePairVirtualSite	A particle representing a “Trivalent Lone Pair”-type virtual site, in which the location of the charge is specified by the positions of four atoms.
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.
ValenceDict	Enforce uniqueness in atom indices.
ImproperDict	Symmetrize improper torsions.

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

property molecule
The Molecule this particle is part of.

property molecule_particle_index
Returns the index of this particle in its molecule

property name
The name of the particle

to_dict()
Convert to dictionary representation.

classmethod from_dict(*d*)
Static constructor from dictionary representation.

classmethod from_bson(*serialized*)
Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_json(*serialized*)
Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_messagepack(*serialized*)
Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

classmethod from_toml(*serialized*)
Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_xml(*serialized*)
Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_yaml(*serialized*)
Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

to_pickle()
Return a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

to_toml()
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

to_xml(*indent=2*)
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

to_yaml()
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

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

is_in_ring Return whether or not this atom is in a ring(s) (of any size)

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

molecule The `Molecule` this particle 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.

continues on next page

Table 18 – continued from previous page

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
`is_aromatic`	The atom's is_aromatic flag
`is_in_ring`	Return whether or not this atom is in a ring(s) (of any size)
`mass`	The standard atomic weight (abundance-weighted isotopic mass) of the atomic site.
`molecule`	The `Molecule` this particle 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`.

continues on next page

Table 20 – continued from previous page

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.

add_bond(*bond*)
Adds a bond that this atom is involved in .. todo :: Is this how we want to keep records?

Parameters

bond: an openforcefield.topology.molecule.Bond A bond involving this atom

add_virtual_site(*vsite*)
Adds a bond that this atom is involved in .. todo :: Is this how we want to keep records?

Parameters

bond: an openforcefield.topology.molecule.Bond A bond involving this atom

to_dict()
Return a dict representation of the atom.

classmethod from_dict(*atom_dict*)
Create an Atom from a dict representation.

property formal_charge
The atom's formal charge

property partial_charge
The partial charge of the atom, if any.

Returns

simtk.unit.Quantity with dimension of atomic charge, or None if no charge has been specified

property is_aromatic
The atom's is_aromatic flag

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

property element
The element name

property atomic_number
The integer atomic number of the atom.

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

TODO (from jeff): Are there atoms that have different chemical properties based on their isotopes?

property name
The name of this atom, if any

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

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

is_bonded_to(*atom2*)
Determine whether this atom is bound to another atom

Parameters

atom2: openforcefield.topology.molecule.Atom a different atom in the same molecule

Returns

bool Whether this atom is bound to atom2

property is_in_ring
Return whether or not this atom is in a ring(s) (of any size)

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

property molecule_atom_index
The index of this Atom within the the list of atoms in `Molecules`. Note that this can be different from `molecule_particle_index`.

property molecule_particle_index
The index of this Atom within the the list of particles in the parent `Molecule`. Note that this can be different from `molecule_atom_index`.

classmethod from_bson(*serialized*)
Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_json(*serialized*)
Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_messagepack(*serialized*)
Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

classmethod from_toml(*serialized*)
Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_xml(*serialized*)
Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_yaml(*serialized*)
Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

property molecule
The `Molecule` this particle is part of.

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

to_pickle()
Return a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

to_toml()
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

to_xml(*indent=2*)
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

to_yaml()
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

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.
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`

continues on next page

Table 23 – continued from previous page

is_in_ring	Return whether or not this bond is in a ring(s) (of any size)
molecule	
molecule_bond_index	The index of this Bond within the the list of bonds in Molecules.
stereochemistry	

to_dict()
Return a dict representation of the bond.

classmethod from_dict(*molecule*, *d*)
Create a Bond from a dict representation.

property molecule_bond_index
The index of this Bond within the the list of bonds in Molecules.

property is_in_ring
Return whether or not this bond is in a ring(s) (of any size)

classmethod from_bson(*serialized*)
Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_json(*serialized*)
Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_messagepack(*serialized*)
Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

classmethod from_toml(*serialized*)
Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_xml(*serialized*)
Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_yaml(*serialized*)
Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

to_pickle()
Return a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

to_toml()
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

to_xml(*indent=2*)
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

to_yaml()
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

openforcefield.topology.VirtualSite

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

A container representing one or more virtual particles whose positions are defined in terms of `Atom` positions. This container enables the coupling of particles that are symmetric about some axis/plane of the underlying atoms. For example, a single virtual site can represent two lone pairs of a water molecule, where the angle and distance parameters are expected to stay coupled, and are reflections across the plane of symmetry.

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 particle is part of.

molecule_particle_index Returns the index of this particle in its 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`.

n_particles The number of particles that the virtual site represents

name The name of this VirtualSite

orientations

particles Particles owned by 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.

continues on next page

Table 24 – continued from previous page

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.

index_of_orientation

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

continues on next page

Table 25 – continued from previous page

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.
index_of_orientation(virtual_particle)	
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 particle is part of.
molecule_particle_index	Returns the index of this particle in its 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.
n_particles	The number of particles that the virtual site represents
name	The name of this VirtualSite
orientations	
particles	Particles owned by 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)

to_dict()
Return a dict representation of the virtual site.

classmethod from_dict(*vsite_dict*)
Create a virtual site from a dict representation.

property particles
Particles owned by this VirtualSite

property n_particles

The number of particles that the virtual site represents

property 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`.

property atoms
Atoms on whose position this VirtualSite depends.

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

property epsilon
The VdW epsilon term of this VirtualSite

property sigma
The VdW sigma term of this VirtualSite

property rmin_half
The VdW rmin_half term of this VirtualSite

property name
The name of this VirtualSite

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

classmethod from_bson(*serialized*)
Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_json(*serialized*)
Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_messagepack(*serialized*)
Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

classmethod from_toml(*serialized*)
Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_xml(*serialized*)
Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_yaml(*serialized*)
Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

property molecule
The `Molecule` this particle is part of.

property molecule_particle_index
Returns the index of this particle in its molecule

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

to_pickle()
Return a pickle serialized representation.

Warning: This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

to_toml()
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

to_xml(*indent=2*)
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

to_yaml()
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

openforcefield.topology.VirtualParticle

class openforcefield.topology.**VirtualParticle**(*vsite*, *orientation*, *name=None*)

A single particle owned by a VirtualSite

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

Attributes

molecule The `Molecule` this particle is part of.

molecule_particle_index Returns the index of this particle in its molecule

name The name of the particle

orientation

virtual_site

virtual_site_particle_index The index of the particle relative to its owning virtual site.

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__(*vsite*, *orientation*, *name=None*)
A single particle owned by a VirtualSite

Parameters

vsite [openforcefield.topology.VirtualSite] The parent VirtualSite of this VirtualParticle

orientation [tuple of int] Molecule atom indices of parent atoms

name [str, optional] The name of the particle

Methods

__init__(vsite, orientation[, name])	A single particle owned by a VirtualSite
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

molecule	The Molecule this particle is part of.
molecule_particle_index	Returns the index of this particle in its molecule
name	The name of the particle
orientation	
virtual_site	
virtual_site_particle_index	The index of the particle relative to its owning virtual site.

property virtual_site_particle_index
The index of the particle relative to its owning virtual site. Normally this should either be 0 or 1.

classmethod from_bson(*serialized*)
Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_dict(*d*)
Static constructor from dictionary representation.

classmethod from_json(*serialized*)
Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_messagepack(*serialized*)
Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

classmethod from_toml(*serialized*)
Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_xml(*serialized*)
Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_yaml(*serialized*)
Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

property molecule
The `Molecule` this particle is part of.

property molecule_particle_index
Returns the index of this particle in its molecule

property name
The name of the particle

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

to_dict()
Convert to dictionary representation.

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

to_pickle()
Return a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

to_toml()
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

to_xml(*indent=2*)
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

to_yaml()
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

openforcefield.topology.TopologyVirtualParticle

class openforcefield.topology.**TopologyVirtualParticle**(*virtual_site*, *virtual_particle*, *topology_molecule*)

Attributes

atoms Get the TopologyAtoms involved in this TopologyVirtualSite.

molecule Get the reference Molecule that this TopologyVirtualSite belongs to.

n_particles Get the number of particles represented by this VirtualSite

particles Get an iterator to the reference particles that this TopologyVirtualSite contains.

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_particle_start_index Get the index of the first virtual site 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.
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.

continues on next page

Table 30 – continued from previous page

to_yaml()	Return a YAML serialized representation.

invalidate_cached_data

__init__(*virtual_site*, *virtual_particle*, *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, virtual_particle, ...)	**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.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
invalidate_cached_data()	
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.
n_particles	Get the number of particles represented by this VirtualSite
particles	Get an iterator to the reference particles that this TopologyVirtualSite contains.
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_particle_start_index	Get the index of the first virtual site 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.

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

Returns

idx [int] The index of this particle in its parent topology.

atom(*index*)
Get the atom at a specific index in this TopologyVirtualSite

Parameters

index [int] The index of the atom in the reference VirtualSite to retrieve

Returns

TopologyAtom

property atoms
Get the TopologyAtoms involved in this TopologyVirtualSite.

Returns

iterator of openforcefield.topology.TopologyAtom

classmethod from_bson(*serialized*)
Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_dict(*d*)
Static constructor from dictionary representation.

classmethod from_json(*serialized*)
Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_messagepack(*serialized*)
Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

classmethod from_toml(*serialized*)
Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_xml(*serialized*)
Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_yaml(*serialized*)
Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

property molecule
Get the reference Molecule that this TopologyVirtualSite belongs to.

Returns

openforcefield.topology.molecule.Molecule

property n_particles
Get the number of particles represented by this VirtualSite

Returns

int [The number of particles]

property particles
Get an iterator to the reference particles that this TopologyVirtualSite contains.

Returns

iterator of TopologyVirtualParticles

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

to_dict()
Convert to dictionary representation.

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

`to_pickle()`
Return a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

`to_toml()`
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

`to_xml(indent=2)`
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

`to_yaml()`
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

`property topology_molecule`
Get the TopologyMolecule that this TopologyVirtualSite belongs to.

Returns

openforcefield.topology.TopologyMolecule

`property topology_virtual_particle_start_index`
Get the index of the first virtual site particle in its parent Topology.

Returns

int The index of this particle in its parent topology.

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

Returns

int The index of this virtual site in its parent topology.

property type
Get the type of this virtual site

Returns

str [The class name of this virtual site]

property virtual_site
Get the reference VirtualSite for this TopologyVirtualSite.

Returns

an openforcefield.topology.molecule.VirtualSite

openforcefield.topology.BondChargeVirtualSite

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

A particle representing a "Bond Charge"-type virtual site, in which the location of the charge is specified by the positions of two atoms. 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.

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

distance The distance parameter of the virtual site

epsilon The VdW epsilon term of this VirtualSite

molecule The `Molecule` this particle is part of.

molecule_particle_index Returns the index of this particle in its 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`.

n_particles The number of particles that the virtual site represents

name The name of this VirtualSite

orientations

particles Particles owned by 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.
get_openmm_virtual_site(atoms)	Returns the OpenMMVirtualSite corresponding to this BondChargeVirtualSite.
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.

index_of_orientation

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

Create a bond charge-type virtual site, in which the location of the charge is specified by the position of two atoms. 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.

TODO: One of the examples on https://open-forcefield-toolkit.readthedocs.io/en/topology/smirnoff.html#virtualsites-virtual-sites-for-off-atom-charges has a BondCharge defined with three atoms – How does that work?

Parameters

atoms [list of openforcefield.topology.molecule.Atom objects of shape [N]] The atoms defining the virtual site's position

distance [float]

weights [list of floats of shape [N] or None, optional, default=None] weights[index] is the weight of particles[index] contributing to the position of the virtual site. Default is None

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 Virtu-

alSite. Indexing in this list should match the ordering in the atoms list. Default is None.

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

sigma [float, default=None] Sigma 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.

orientations [list of tuples of 3 Atoms or ints] The permutations of the matched atoms that should be used to define the orientation of each virtual site particle

Methods

`__init__`(atoms, distance[, ...])	Create a bond charge-type virtual site, in which the location of the charge is specified by the position of two atoms.
`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.
`get_openmm_virtual_site`(atoms)	Returns the OpenMMVirtualSite corresponding to this BondChargeVirtualSite.
`index_of_orientation`(virtual_particle)	
`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
distance	The distance parameter of the virtual site
epsilon	The VdW epsilon term of this VirtualSite
molecule	The Molecule this particle is part of.
molecule_particle_index	Returns the index of this particle in its 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.
n_particles	The number of particles that the virtual site represents
name	The name of this VirtualSite
orientations	
particles	Particles owned by 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)

to_dict()
Return a dict representation of the virtual site.

classmethod from_dict(*vsite_dict*)
Create a virtual site from a dict representation.

property distance
The distance parameter of the virtual site

get_openmm_virtual_site(*atoms*)
Returns the OpenMMVirtualSite corresponding to this BondChargeVirtualSite.

Parameters

atoms [iterable of int] The indices of the atoms involved in this virtual site (not assumed to be the same as the molecule indices as this method may be accessed with regard to particles in a Topology).

Returns

virtual_site [a simtk.openmm LocalCoordinatesSite]

property atoms
Atoms on whose position this VirtualSite depends.

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

property epsilon
The VdW epsilon term of this VirtualSite

classmethod from_bson(*serialized*)
Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_json(*serialized*)
Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_messagepack(*serialized*)
Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

classmethod from_toml(*serialized*)
Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_xml(*serialized*)
Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_yaml(*serialized*)
Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

property molecule
The `Molecule` this particle is part of.

property molecule_particle_index
Returns the index of this particle in its molecule

property 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`.

property n_particles
The number of particles that the virtual site represents

property name
The name of this VirtualSite

property particles
Particles owned by this VirtualSite

property rmin_half
The VdW rmin_half term of this VirtualSite

property sigma
The VdW sigma term of this VirtualSite

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

to_pickle()
Return a pickle serialized representation.

Warning: This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

to_toml()
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

to_xml(*indent=2*)
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

to_yaml()
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

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

openforcefield.topology.MonovalentLonePairVirtualSite

class openforcefield.topology.**MonovalentLonePairVirtualSite**(*atoms*, *distance*, *out_of_plane_angle*, *in_plane_angle*, *charge_increments=None*, *epsilon=None*, *sigma=None*, *rmin_half=None*, *name=None*, *orientations=None*)

A particle representing a "Monovalent Lone Pair"-type virtual site, in which the location of the charge is specified by the positions of three atoms. This is originally intended for situations like a carbonyl, and allows placement of a virtual site S at a specified distance d, in_plane_angle, and out_of_plane_angle relative to a central atom and two connected atoms.

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

distance The distance parameter of the virtual site

epsilon The VdW epsilon term of this VirtualSite

in_plane_angle The in_plane_angle parameter of the virtual site

molecule The `Molecule` this particle is part of.

molecule_particle_index Returns the index of this particle in its 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`.

n_particles The number of particles that the virtual site represents

name The name of this VirtualSite

orientations

out_of_plane_angle The out_of_plane_angle parameter of the virtual site

particles Particles owned by 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)	Construct a new MonovalentLonePairVirtualSite from an serialized 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.
get_openmm_virtual_site(atoms)	Returns the OpenMMVirtualSite corresponding to this MonovalentLonePairVirtualSite.
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.

index_of_orientation	

__init__(*atoms*, *distance*, *out_of_plane_angle*, *in_plane_angle*, *charge_increments=None*, *epsilon=None*, *sigma=None*, *rmin_half=None*, *name=None*, *orientations=None*)

Create a bond charge-type virtual site, in which the location of the charge is specified by the position of three atoms.

Parameters

atoms [list of three openforcefield.topology.molecule.Atom objects] The three atoms defining the virtual site's position

distance [float]

out_of_plane_angle [float]

in_plane_angle [float]

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

sigma [float, default=None] Sigma 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.

orientations [list of tuples of 3 Atoms or ints] The permutations of the matched atoms that should be used to define the orientation of each virtual site particle

Methods

`__init__`(atoms, distance, ... [, ...])	Create a bond charge-type virtual site, in which the location of the charge is specified by the position of three atoms.
`from_bson`(serialized)	Instantiate an object from a BSON serialized representation.
`from_dict`(vsite_dict)	Construct a new MonovalentLonePairVirtualSite from an serialized 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.
`get_openmm_virtual_site`(atoms)	Returns the OpenMMVirtualSite corresponding to this MonovalentLonePairVirtualSite.
`index_of_orientation`(virtual_particle)	
`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
`distance`	The distance parameter of the virtual site
`epsilon`	The VdW epsilon term of this VirtualSite
`in_plane_angle`	The in_plane_angle parameter of the virtual site
`molecule`	The `Molecule` this particle is part of.
`molecule_particle_index`	Returns the index of this particle in its molecule

continues on next page

Table 38 – continued from previous page

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.
n_particles	The number of particles that the virtual site represents
name	The name of this VirtualSite
orientations	
out_of_plane_angle	The out_of_plane_angle parameter of the virtual site
particles	Particles owned by 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)

to_dict()
Return a dict representation of the virtual site.

classmethod from_dict(*vsite_dict*)
Construct a new MonovalentLonePairVirtualSite from an serialized dictionary representation.

Parameters

vsite_dict [dict] The VirtualSite to deserialize.

Returns

The newly created MonovalentLonePairVirtualSite

property distance
The distance parameter of the virtual site

property in_plane_angle
The in_plane_angle parameter of the virtual site

property out_of_plane_angle
The out_of_plane_angle parameter of the virtual site

get_openmm_virtual_site(*atoms*)
Returns the OpenMMVirtualSite corresponding to this MonovalentLonePairVirtualSite.

Parameters

atoms [iterable of int] The indices of the atoms involved in this virtual site (not assumed to be the same as the molecule indices as this method may be accessed with regard to particles in a Topology).

Returns

virtual_site [a simtk.openmm LocalCoordinatesSite]

property atoms
Atoms on whose position this VirtualSite depends.

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

property epsilon
The VdW epsilon term of this VirtualSite

classmethod from_bson(*serialized*)
Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_json(*serialized*)
Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_messagepack(*serialized*)
Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

classmethod from_toml(*serialized*)
Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_xml(*serialized*)
Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_yaml(*serialized*)
Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

property molecule
The `Molecule` this particle is part of.

property molecule_particle_index
Returns the index of this particle in its molecule

property 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`.

property n_particles
The number of particles that the virtual site represents

property name
The name of this VirtualSite

property particles
Particles owned by this VirtualSite

property rmin_half
The VdW rmin_half term of this VirtualSite

property sigma
The VdW sigma term of this VirtualSite

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

to_pickle()
Return a pickle serialized representation.

Warning: This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

to_toml()
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

to_xml(*indent=2*)
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

to_yaml()
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

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

openforcefield.topology.DivalentLonePairVirtualSite

class openforcefield.topology.**DivalentLonePairVirtualSite**(*atoms*, *distance*, *out_of_plane_angle*, *charge_increments=None*, *epsilon=None*, *sigma=None*, *rmin_half=None*, *name=None*, *orientations=None*)

A particle representing a "Divalent Lone Pair"-type virtual site, in which the location of the charge is specified by the positions of three atoms. 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 along the bisector of the angle between the atoms (if out_of_plane_angle is zero) or out of the plane by the specified angle (if out_of_plane_angle is nonzero) with its projection along the bisector. For positive values of the distance d the virtual site lies outside the 2-1-3 angle and for negative values it lies inside.

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

distance The distance parameter of the virtual site

epsilon The VdW epsilon term of this VirtualSite

molecule The `Molecule` this particle is part of.

molecule_particle_index Returns the index of this particle in its 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`.

n_particles The number of particles that the virtual site represents

name The name of this VirtualSite

orientations

out_of_plane_angle The out_of_plane_angle parameter of the virtual site

particles Particles owned by 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)	Construct a new DivalentLonePairVirtualSite from an serialized 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.
get_openmm_virtual_site(atoms)	Returns the OpenMMVirtualSite corresponding to this DivalentLonePairVirtualSite.
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.

index_of_orientation	

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

Create a divalent lone pair-type virtual site, in which the location of the charge is specified by the position of three atoms.

Parameters

atoms [list of 3 openforcefield.topology.molecule.Atom objects] The three atoms defining the virtual site's position

distance [float]

out_of_plane_angle [float]

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

sigma [float, default=None] Sigma 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.

orientations [list of tuples of 3 Atoms or ints] The permutations of the matched atoms that should be used to define the orientation of each virtual site particle

Methods

`__init__`(atoms, distance, out_of_plane_angle)	Create a divalent lone pair-type virtual site, in which the location of the charge is specified by the position of three atoms.
`from_bson`(serialized)	Instantiate an object from a BSON serialized representation.
`from_dict`(vsite_dict)	Construct a new DivalentLonePairVirtualSite from an serialized 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.
`get_openmm_virtual_site`(atoms)	Returns the OpenMMVirtualSite corresponding to this DivalentLonePairVirtualSite.
`index_of_orientation`(virtual_particle)	
`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
`distance`	The distance parameter of the virtual site
`epsilon`	The VdW epsilon term of this VirtualSite
`molecule`	The `Molecule` this particle is part of.
`molecule_particle_index`	Returns the index of this particle in its 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`.
`n_particles`	The number of particles that the virtual site represents

continues on next page

Table 41 – continued from previous page

name	The name of this VirtualSite
orientations	
out_of_plane_angle	The out_of_plane_angle parameter of the virtual site
particles	Particles owned by 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)

to_dict()
Return a dict representation of the virtual site.

classmethod from_dict(*vsite_dict*)
Construct a new DivalentLonePairVirtualSite from an serialized dictionary representation.

Parameters

vsite_dict [dict] The VirtualSite to deserialize.

Returns

The newly created DivalentLonePairVirtualSite

property distance
The distance parameter of the virtual site

property out_of_plane_angle
The out_of_plane_angle parameter of the virtual site

get_openmm_virtual_site(*atoms*)
Returns the OpenMMVirtualSite corresponding to this DivalentLonePairVirtualSite.

Parameters

atoms [iterable of int] The indices of the atoms involved in this virtual site (not assumed to be the same as the molecule indices as this method may be accessed with regard to particles in a Topology).

Returns

virtual_site [a simtk.openmm LocalCoordinatesSite]

property atoms
Atoms on whose position this VirtualSite depends.

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

property epsilon
The VdW epsilon term of this VirtualSite

classmethod from_bson(*serialized*)
Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_json(*serialized*)
Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_messagepack(*serialized*)
Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

classmethod from_toml(*serialized*)
Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_xml(*serialized*)
Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_yaml(*serialized*)
Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

property molecule
The `Molecule` this particle is part of.

property molecule_particle_index
Returns the index of this particle in its molecule

property 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`.

property n_particles
The number of particles that the virtual site represents

property name
The name of this VirtualSite

property particles
Particles owned by this VirtualSite

property rmin_half
The VdW rmin_half term of this VirtualSite

property sigma
The VdW sigma term of this VirtualSite

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

to_pickle()
Return a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

to_toml()
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

to_xml(*indent=2*)
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

to_yaml()
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

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

openforcefield.topology.TrivalentLonePairVirtualSite

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

A particle representing a "Trivalent Lone Pair"-type virtual site, in which the location of the charge is specified by the positions of four atoms. This is suitable for planar or tetrahedral nitrogen lone pairs; a charge site S lies above the central atom (e.g. nitrogen 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.

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

distance The distance parameter of the virtual site

epsilon The VdW epsilon term of this VirtualSite

molecule The `Molecule` this particle is part of.

molecule_particle_index Returns the index of this particle in its 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`.

n_particles The number of particles that the virtual site represents

name The name of this VirtualSite

orientations

particles Particles owned by 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)	Construct a new TrivalentPairVirtualSite from an serialized dictionary representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.

continues on next page

Table 42 – continued from previous page

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_openmm_virtual_site(atoms)	Returns the OpenMMVirtualSite corresponding to this TrivalentLonePairVirtualSite.
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.

index_of_orientation

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

Create a trivalent lone pair-type virtual site, in which the location of the charge is specified by the position of four atoms.

Parameters

atoms [list of 4 openforcefield.topology.molecule.Atom objects] The three atoms defining the virtual site's position

distance [float]

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

sigma [float, default=None] Sigma 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.

orientations [list of tuples of 3 Atoms or ints] The permutations of the matched atoms that should be used to define the orientation of each virtual site particle

Methods

__init__(atoms, distance[, ...])	Create a trivalent lone pair-type virtual site, in which the location of the charge is specified by the position of four atoms.
from_bson(serialized)	Instantiate an object from a BSON serialized representation.

continues on next page

Table 43 – continued from previous page

from_dict(vsite_dict)	Construct a new TrivalentPairVirtualSite from an serialized 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.
get_openmm_virtual_site(atoms)	Returns the OpenMMVirtualSite corresponding to this TrivalentLonePairVirtualSite.
index_of_orientation(virtual_particle)	
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
distance	The distance parameter of the virtual site
epsilon	The VdW epsilon term of this VirtualSite
molecule	The Molecule this particle is part of.
molecule_particle_index	Returns the index of this particle in its 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.
n_particles	The number of particles that the virtual site represents
name	The name of this VirtualSite
orientations	
particles	Particles owned by this VirtualSite
rmin_half	The VdW rmin_half term of this VirtualSite
sigma	The VdW sigma term of this VirtualSite

continues on next page

Table 44 – continued from previous page

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

to_dict()
Return a dict representation of the virtual site.

classmethod from_dict(*vsite_dict*)
Construct a new TrivalentPairVirtualSite from an serialized dictionary representation.

Parameters

vsite_dict [dict] The VirtualSite to deserialize.

Returns

The newly created TrivalentLonePairVirtualSite

property distance
The distance parameter of the virtual site

get_openmm_virtual_site(*atoms*)
Returns the OpenMMVirtualSite corresponding to this TrivalentLonePairVirtualSite.

Parameters

atoms [iterable of int] The indices of the atoms involved in this virtual site (not assumed to be the same as the molecule indices as this method amy be accessed with regard to particles in a Topology).

Returns

virtual_site [a simtk.openmm LocalCoordinatesSite]

property atoms
Atoms on whose position this VirtualSite depends.

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

property epsilon
The VdW epsilon term of this VirtualSite

classmethod from_bson(*serialized*)
Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_json(*serialized*)
Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_messagepack(*serialized*)
Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

classmethod from_toml(*serialized*)
Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_xml(*serialized*)
Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_yaml(*serialized*)
Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

property molecule
The `Molecule` this particle is part of.

property molecule_particle_index
Returns the index of this particle in its molecule

property 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`.

property n_particles
The number of particles that the virtual site represents

property name
The name of this VirtualSite

property particles
Particles owned by this VirtualSite

property rmin_half
The VdW rmin_half term of this VirtualSite

property sigma
The VdW sigma term of this VirtualSite

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

to_pickle()
Return a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

to_toml()
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

to_xml(*indent=2*)
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

to_yaml()
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

property 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.
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.
`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.

property atom
Get the reference Atom for this TopologyAtom.

Returns

an openforcefield.topology.molecule.Atom

property atomic_number
Get the atomic number of this atom

Returns

int

property topology_molecule
Get the TopologyMolecule that this TopologyAtom belongs to.

Returns

openforcefield.topology.TopologyMolecule

property molecule
Get the reference Molecule that this TopologyAtom belongs to.

Returns

openforcefield.topology.molecule.Molecule

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

Returns

int The index of this atom in its parent topology.

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

Returns

int The index of this atom in its parent topology.

property topology_bonds
Get the TopologyBonds connected to this TopologyAtom.

Returns

iterator of openforcefield.topology.TopologyBonds

to_dict()
Convert to dictionary representation.

classmethod from_dict(*d*)
Static constructor from dictionary representation.

classmethod from_bson(*serialized*)
Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_json(*serialized*)
Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_messagepack(*serialized*)
Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

classmethod from_toml(*serialized*)
Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_xml(*serialized*)
Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_yaml(*serialized*)
Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

to_pickle()
Return a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

to_toml()
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

to_xml(*indent=2*)
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

to_yaml()
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

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.

continues on next page

Table 48 – 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()	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.

property bond
Get the reference Bond for this TopologyBond.

Returns

an openforcefield.topology.molecule.Bond

property topology_molecule
Get the TopologyMolecule that this TopologyBond belongs to.

Returns

openforcefield.topology.TopologyMolecule

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

Returns

int The index of this bond in its parent topology.

property molecule
Get the reference Molecule that this TopologyBond belongs to.

Returns

openforcefield.topology.molecule.Molecule

property bond_order
Get the order of this TopologyBond.

Returns

int [bond order]

property atoms
Get the TopologyAtoms connected to this TopologyBond.

Returns

iterator of openforcefield.topology.TopologyAtom

to_dict()
Convert to dictionary representation.

classmethod from_dict(*d*)
Static constructor from dictionary representation.

classmethod from_bson(*serialized*)
Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_json(*serialized*)
Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_messagepack(*serialized*)
Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

classmethod from_toml(*serialized*)
Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_xml(*serialized*)
Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_yaml(*serialized*)
Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

to_pickle()
Return a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

`to_toml`()
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

`to_xml`(*indent=2*)
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

`to_yaml`()
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

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.

`n_particles` Get the number of particles represented by this VirtualSite

`particles` Get an iterator to the reference particles that this TopologyVirtualSite contains.

`topology_molecule` Get the TopologyMolecule that this TopologyVirtualSite belongs to.

topology_virtual_particle_start_index Get the index of the first virtual site 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.
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.

invalidate_cached_data	

__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.
`from_yaml`(serialized)	Instantiate from a YAML serialized representation.
`invalidate_cached_data`()	
`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.
`n_particles`	Get the number of particles represented by this VirtualSite
`particles`	Get an iterator to the reference particles that this TopologyVirtualSite contains.
`topology_molecule`	Get the TopologyMolecule that this TopologyVirtualSite belongs to.
`topology_virtual_particle_start_index`	Get the index of the first virtual site particle in its parent Topology.
`topology_virtual_site_index`	Get the index of this virtual site in its parent Topology.

continues on next page

Table 53 – continued from previous page

type	Get the type of this virtual site
virtual_site	Get the reference VirtualSite for this TopologyVirtualSite.

atom(*index*)
Get the atom at a specific index in this TopologyVirtualSite

Parameters

index [int] The index of the atom in the reference VirtualSite to retrieve

Returns

TopologyAtom

property atoms
Get the TopologyAtoms involved in this TopologyVirtualSite.

Returns

iterator of openforcefield.topology.TopologyAtom

property virtual_site
Get the reference VirtualSite for this TopologyVirtualSite.

Returns

an openforcefield.topology.molecule.VirtualSite

property topology_molecule
Get the TopologyMolecule that this TopologyVirtualSite belongs to.

Returns

openforcefield.topology.TopologyMolecule

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

Returns

int The index of this virtual site in its parent topology.

property n_particles
Get the number of particles represented by this VirtualSite

Returns

int [The number of particles]

property topology_virtual_particle_start_index
Get the index of the first virtual site particle in its parent Topology.

Returns

int The index of this particle in its parent topology.

property particles
Get an iterator to the reference particles that this TopologyVirtualSite contains.

Returns

iterator of TopologyVirtualParticles

property molecule
Get the reference Molecule that this TopologyVirtualSite belongs to.

Returns

openforcefield.topology.molecule.Molecule

property type
Get the type of this virtual site

Returns

str [The class name of this virtual site]

to_dict()
Convert to dictionary representation.

classmethod from_dict(*d*)
Static constructor from dictionary representation.

classmethod from_bson(*serialized*)
Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_json(*serialized*)
Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_messagepack(*serialized*)
Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

classmethod from_toml(*serialized*)
Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

classmethod from_xml(*serialized*)
Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

classmethod from_yaml(*serialized*)
Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

to_pickle()
Return a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

to_toml()
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

to_xml(*indent=2*)
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

to_yaml()
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

openforcefield.topology.ValenceDict

class openforcefield.topology.**ValenceDict**(**args*, ***kwargs*)
Enforce uniqueness in atom indices.

Methods

clear()	
get(k[,d])	
index_of(key[, possible])	Generates a canonical ordering of the equivalent permutations of key (equivalent rearrangements of indices) and identifies which of those possible orderings this particular ordering is.
items()	
key_transform(key)	Reverse tuple if first element is larger than last element.
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
setdefault(k[,d])	
update([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values()	

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

Methods

__init__(*args, **kwargs)	Initialize self.
clear()	
get(k[,d])	
index_of(key[, possible])	Generates a canonical ordering of the equivalent permutations of key (equivalent rearrangements of indices) and identifies which of those possible orderings this particular ordering is.

continues on next page

Table 55 – continued from previous page

items()	
key_transform(key)	Reverse tuple if first element is larger than last element.
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
setdefault(k[,d])	
update([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values()	

static key_transform(*key*)
Reverse tuple if first element is larger than last element.

static index_of(*key*, *possible=None*)
Generates a canonical ordering of the equivalent permutations of `key` (equivalent rearrangements of indices) and identifies which of those possible orderings this particular ordering is. This method is useful when multiple SMARTS patterns might match the same atoms, but local molecular symmetry or the use of wildcards in the SMARTS could make the matches occur in arbitrary order.

This method can be restricted to a subset of the canonical orderings, by providing the optional `possible` keyword argument. If provided, the index returned by this method will be the index of the element in `possible` after undergoing the same canonical sorting as above.

Parameters

key [iterable of int] A valid key for ValenceDict

possible [iterable of iterable of int, optional. default=``None``] A subset of the possible orderings that this match might take.

Returns

index [int]

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

openforcefield.topology.ImproperDict

class openforcefield.topology.**ImproperDict**(**args*, ***kwargs*)
Symmetrize improper torsions.

Methods

clear()	
get(k[,d])	
index_of(key[, possible])	Generates a canonical ordering of the equivalent permutations of key (equivalent rearrangements of indices) and identifies which of those possible orderings this particular ordering is.
items()	
key_transform(key)	Reorder tuple in numerical order except for element[1] which is the central atom; it retains its position.
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
setdefault(k[,d])	
update([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values()	

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

Methods

`__init__`(*args, **kwargs)	Initialize self.
`clear`()	
`get`(k[,d])	
`index_of`(key[, possible])	Generates a canonical ordering of the equivalent permutations of key (equivalent rearrangements of indices) and identifies which of those possible orderings this particular ordering is.
`items`()	
`key_transform`(key)	Reorder tuple in numerical order except for element[1] which is the central atom; it retains its position.
`keys`()	
`pop`(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
`popitem`()	as a 2-tuple; but raise KeyError if D is empty.
`setdefault`(k[,d])	
`update`([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
`values`()	

static key_transform(*key*)
Reorder tuple in numerical order except for element[1] which is the central atom; it retains its position.

static index_of(*key*, *possible=None*)
Generates a canonical ordering of the equivalent permutations of key (equivalent rearrangements of indices) and identifies which of those possible orderings this particular ordering is. This method is useful when multiple SMARTS patterns might match the same atoms, but local molecular symmetry or the use of wildcards in the SMARTS could make the matches occur in arbitrary order.

This method can be restricted to a subset of the canonical orderings, by providing the optional possible keyword argument. If provided, the index returned by this method will be the index of the element in possible after undergoing the same canonical sorting as above.

Parameters

key [iterable of int] A valid key for ValenceDict

possible [iterable of iterable of int, optional. default=``None``] A subset of the possible orderings that this match might take.

Returns

index [int]

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

2.2 Forcefield typing tools

2.2.1 Chemical environments

Tools for representing and operating on chemical environments

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.

validate(*validate_valence_type=True*, *toolkit_registry=None*)

Returns True if the underlying smirks is the correct valence type, False otherwise. If the expected type is None, this method always returns True.

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

toolkit_registry = ToolkitWrapper or ToolkitRegistry. Default = None Either a ToolkitRegistry or ToolkitWrapper, 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

classmethod validate_smirks(*smirks*, *validate_parsable=True*, *validate_valence_type=True*, *toolkit_registry=None*)

Check the provided SMIRKS string is valid, and if requested, tags atoms appropriate to the specified valence type.

Parameters

smirks [str] The SMIRKS expression to validate

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

get_type(*toolkit_registry=None*)

Return the valence type implied by the connectivity of the bound atoms in this ChemicalEnvironment.

Parameters

toolkit_registry [openforcefield.utils.ToolkitRegistry or openforcefield.utils.ToolkitWrapper] The cheminformatics toolkit to use for parsing the smirks

Returns

valence_type [str] One of "Atom", "Bond", "Angle", "ProperTorsion", "ImproperTorsion", or None. If tagged atoms are not connected in a known pattern this method will return None.

Raises

SMIRKSParsingError if smirks was unparsable

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
`get_available_force_fields`	Get the filenames of all available .offxml force field files.

openforcefield.typing.engines.smirnoff.forcefield.ForceField

class openforcefield.typing.engines.smirnoff.forcefield.**ForceField**(**sources*, *aromaticity_model='OEAroModel_MDL'*, *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/test_forcefield.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
deregister_parameter_handler(handler)	Deregister a parameter handler specified by tag name, class, or instance.
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.

__init__(**sources, aromaticity_model='OEAroModel_MDL', 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.

aromaticity_model [string, default='OEAroModel_MDL'] The aromaticity model used by the force field. Currently, only 'OEAroModel_MDL' is supported

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/test_forcefield.offxml')
```

Load multiple SMIRNOFF parameter sets:

forcefield = ForceField('test_forcefields/test_forcefield.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[, aromaticity_model, ...])	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
`deregister_parameter_handler`(handler)	Deregister a parameter handler specified by tag name, class, or instance.
`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

`aromaticity_model`	Returns the aromaticity model for this ForceField object.
`author`	Returns the author data for this ForceField object.
`date`	Returns the date data for this ForceField object.
`registered_parameter_handlers`	Return the list of registered parameter handlers by name

property aromaticity_model

Returns the aromaticity model for this ForceField object.

Returns

aromaticity_model The aromaticity model for this force field.

property author

Returns the author data for this ForceField object. If not defined in any loaded files, this will be

None.

Returns

author [str] The author data for this forcefield.

property date

Returns the date data for this ForceField object. If not defined in any loaded files, this will be None.

Returns

date [str] The date data for this forcefield.

register_parameter_handler(*parameter_handler*)

Register a new ParameterHandler for a specific tag, making it available for lookup in the ForceField.

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

Parameters

parameter_handler [A ParameterHandler object] The ParameterHandler to register. The TAGNAME attribute of this object will be used as the key for registration.

register_parameter_io_handler(*parameter_io_handler*)

Register a new ParameterIOHandler, making it available for lookup in the ForceField.

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

Parameters

parameter_io_handler [A ParameterIOHandler object] The ParameterIOHandler to register. The FORMAT attribute of this object will be used to associate it to a file format/suffix.

property registered_parameter_handlers

Return the list of registered parameter handlers by name

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

Returns

registered_parameter_handlers: iterable of names of ParameterHandler objects in this ForceField

get_parameter_handler(*tagname*, *handler_kwargs=None*, *allow_cosmetic_attributes=False*)

Retrieve the parameter handlers associated with the provided tagname.

If the parameter handler has not yet been instantiated, it will be created and returned. If a parameter handler object already exists, it will be checked for compatibility and an Exception raised if it is incompatible with the provided kwargs. If compatible, the existing ParameterHandler will be returned.

Parameters

tagname [str] The name of the parameter to be handled.

handler_kwargs [dict, optional. Default = None] Dict to be passed to the handler for construction or checking compatibility. If this is None and no existing ParameterHandler exists for the desired tag, a handler will be initialized with all default values. If this is None and a handler for the desired tag exists, the existing ParameterHandler will be returned.

allow_cosmetic_attributes [bool, optional. Default = False] Whether to permit non-spec kwargs in smirnoff_data.

Returns

handler [An openforcefield.engines.typing.smirnoff.ParameterHandler]

Raises

KeyError if there is no ParameterHandler for the given tagname

`get_parameter_io_handler`(*io_format*)

Retrieve the parameter handlers associated with the provided tagname. If the parameter IO handler has not yet been instantiated, it will be created.

Parameters

io_format [str] The name of the io format to be handled.

Returns

io_handler [An openforcefield.engines.typing.smirnoff.ParameterIOHandler]

Raises

KeyError if there is no ParameterIOHandler for the given tagname

`deregister_parameter_handler`(*handler*)

Deregister a parameter handler specified by tag name, class, or instance.

Parameters

handler: str, openforcefield.typing.engines.smirnoff.ParameterHandler-derived type or object The handler to deregister.

`parse_sources`(*sources*, *allow_cosmetic_attributes=True*)

Parse a SMIRNOFF force field definition.

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.

allow_cosmetic_attributes [bool, optional. Default = False] Whether to permit non-spec kwargs present in the source.

.. notes ::

- New SMIRNOFF sections are handled independently, as if they were specified in the same file.
- If a SMIRNOFF section that has already been read appears again, its definitions are appended to the end of the previously-read definitions if the sections are configured with compatible attributes; otherwise, an `IncompatibleTagException` is raised.

parse_smirnoff_from_source(*source*)

Reads a SMIRNOFF data structure from a source, which can be one of many types.

Parameters

source [str or bytes] 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.

Returns

smirnoff_data [OrderedDict] A representation of a SMIRNOFF-format data structure. Begins at top-level 'SMIRNOFF' key.

to_string(*io_format='XML'*, *discard_cosmetic_attributes=False*)

Write this Forcefield and all its associated parameters to a string in a given format which complies with the SMIRNOFF spec.

Parameters

io_format [str or ParameterIOHandler, optional. Default='XML'] The serialization format to write to

discard_cosmetic_attributes [bool, default=False] Whether to discard any non-spec attributes stored in the ForceField.

Returns

forcefield_string [str] The string representation of the serialized forcefield

to_file(*filename*, *io_format=None*, *discard_cosmetic_attributes=False*)

Write this Forcefield and all its associated parameters to a string in a given format which complies with the SMIRNOFF spec.

Parameters

filename [str] The filename to write to

io_format [str or ParameterIOHandler, optional. Default=None] The serialization format to write out. If None, will attempt to be inferred from the filename.

discard_cosmetic_attributes [bool, default=False] Whether to discard any non-spec attributes stored in the ForceField.

Returns

forcefield_string [str] The string representation of the serialized forcefield

create_openmm_system(*topology*, ***kwargs*)

Create an OpenMM System representing the interactions for the specified Topology with the current force field

Parameters

topology [openforcefield.topology.Topology] The `Topology` corresponding to the system to be parameterized

charge_from_molecules [List[openforcefield.molecule.Molecule], optional. default =[]] If specified, partial charges will be taken from the given molecules instead of being determined by the force field.

partial_bond_orders_from_molecules [List[openforcefield.molecule.Molecule], optional. default=[]] If specified, partial bond orders will be taken from the given molecules instead of being determined by the force field. **All** bonds on each molecule given must have `fractional_bond_order` specified. A *ValueError* will be raised if any bonds have `fractional_bond_order=None`. Molecules in the topology not represented in this list will have fractional bond orders calculated using underlying toolkits as needed.

return_topology [bool, optional. default=False] If `True`, return tuple of `(system, topology)`, where `topology` is the processed topology. Default `False`. This topology will have the final partial charges assigned on its reference_molecules attribute, as well as partial bond orders (if they were calculated).

toolkit_registry [openforcefield.utils.toolkits.ToolkitRegistry, optional. default=GLOBAL_TOOLKIT_REGISTRY] The toolkit registry to use for operations like conformer generation and partial charge assignment.

Returns

system [simtk.openmm.System] The newly created OpenMM System corresponding to the specified `topology`

topology [openforcefield.topology.Topology, optional.] If the *return_topology* keyword argument is used, this method will also return a Topology. This can be used to inspect the partial charges and partial bond orders assigned to the molecules during parameterization.

create_parmed_structure(*topology*, *positions*, ***kwargs*)

Create a ParmEd Structure object representing the interactions for the specified Topology with the current force field

This method creates a ParmEd `Structure` object containing a topology, positions, and parameters.

Parameters

topology [openforcefield.topology.Topology] The `Topology` corresponding to the `System` object to be created.

positions [simtk.unit.Quantity of dimension (natoms,3) with units compatible with angstroms] The positions corresponding to the `System` object to be created

Returns

structure [parmed.Structure] The newly created `parmed.Structure` object

label_molecules(*topology*)

Return labels for a list of molecules corresponding to parameters from this force field. For each molecule, a dictionary of force types is returned, and for each force type, each force term is provided with the atoms involved, the parameter id assigned, and the corresponding SMIRKS.

Parameters

topology [openforcefield.topology.Topology] A Topology object containing one or more unique molecules to be labeled

Returns

molecule_labels [list] List of labels for unique molecules. Each entry in the list corresponds to one unique molecule in the Topology and is a dictionary keyed by force type, i.e., `molecule_labels[0]['HarmonicBondForce']` gives details for the harmonic bond parameters for the first molecule. Each element is a list of the form: `[ ( [ atom1, ..., atomN], parameter_id, SMIRKS), ... ]`.

openforcefield.typing.engines.smirnoff.forcefield.get_available_force_fields

openforcefield.typing.engines.smirnoff.forcefield.**get_available_force_fields**(*full_paths=False*)

Get the filenames of all available .offxml force field files.

Availability is determined by what is discovered through the

openforcefield.smirnoff_forcefield_directory entry point. If the *openforcefields* package is installed, this should include several .offxml files such as *openff-1.0.0.offxml*.

Parameters

full_paths [bool, default=False] If False, return the name of each available *.offxml file. If True, return the full path to each available .offxml file.

Returns

available_force_fields [List[str]] List of available force field files

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)
...     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
```

(continues on next page)

(continued from previous page)

```
...
>>> 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.

add_cosmetic_attribute(*attr_name*, *attr_value*)
Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the attribute to define for this object.

attr_value [str] The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)
Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] The attribute name to check

Returns

is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)
Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)
Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters

discard_cosmetic_attributes [bool, optional. Default = False] Whether to discard non-spec attributes of this object

duplicate_attributes [list of string, optional. Default = None] A list of names of attributes that redundantly decsribe data and should be discarded during serialization

Returns

smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this object.

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
`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.
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.

append(*parameter*)

Add a ParameterType object to the end of the ParameterList

Parameters

parameter [a ParameterType object]

extend(*other*)
Add a ParameterList object to the end of the ParameterList

Parameters

other [a ParameterList]

index(*item*)
Get the numerical index of a ParameterType object or SMIRKS in this ParameterList. Raises ParameterLookupError if the item is not found.

Parameters

item [ParameterType object or str] The parameter or SMIRKS to look up in this ParameterList

Returns

index [int] The index of the found item

Raises

ParameterLookupError if SMIRKS pattern is passed in but not found

insert(*index*, *parameter*)
Add a ParameterType object as if this were a list

Parameters

index [int] The numerical position to insert the parameter at

parameter [a ParameterType object] The parameter to insert

to_list(*discard_cosmetic_attributes=True*)
Render this ParameterList to a normal list, serializing each ParameterType object in it to dict.

Parameters

discard_cosmetic_attributes [bool, optional. Default = True] Whether to discard non-spec attributes of each ParameterType object.

Returns

parameter_list [List[dict]] A serialized representation of a ParameterList, with each ParameterType it contains converted to dict.

clear(*/*)
Remove all items from list.

copy(*/*)
Return a shallow copy of the list.

count(*value*, */*)
Return number of occurrences of value.

pop(*index=-1*, */*)
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

remove(*value*, */*)
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse(*/*)
Reverse *IN PLACE*.

sort(***, *key=None*, *reverse=False*)
Stable sort *IN PLACE*.

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

TAGNAME The name of this ParameterHandler corresponding to the SMIRNOFF tag name

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
```

(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

Methods

add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter([parameter_kwargs, parameter, ...])	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.

assign_partial_bond_orders_from_molecules	
check_partial_bond_orders_from_molecules_duplicates	

__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, parameter, ...])	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.
`check_handler_compatibility`(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
`check_partial_bond_orders_from_molecules_duplicates`(pb_mols)	
`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

`TAGNAME`	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
`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.

property parameters
The ParameterList that holds this ParameterHandler's parameter objects

property TAGNAME
The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns

handler_name [str] The name of this parameter handler

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

check_handler_compatibility(*handler_kwargs*)

Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters

handler_kwargs [dict] The kwargs that would be used to construct

Raises

IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs: dict, optional The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

parameter: ParameterType, optional A ParameterType to add to the ParameterHandler

after [str or int, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added

before [str, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added

Note that one of (parameter_kwargs, parameter) must be specified

Note that when `before` and `after` are both None, the new parameter will be appended to the END of the parameter list.

Note that when `before` and `after` are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from simtk import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie_per_mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

get_parameter(*parameter_attrs*)
Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters

parameter_attrs [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns

params [list of ParameterType objects] A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from simtk import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie_per_mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

find_matches(*entity*)
Find the elements of the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology] Topology to search.

Returns

matches [ValenceDict[Tuple[int], ParameterHandler._Match]] `matches[particle_indices]` is the `ParameterType` object matching the tuple of particle indices in `entity`.

assign_parameters(*topology*, *system*)
Assign parameters for the given Topology to the specified System object.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

postprocess_system(*topology*, *system*, ***kwargs*)
Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

to_dict(*discard_cosmetic_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the attribute to define for this object.

attr_value [str] The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] The attribute name to check

Returns

is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the cosmetic attribute to delete.

openforcefield.typing.engines.smirnoff.parameters.BondHandler

class openforcefield.typing.engines.smirnoff.parameters.**BondHandler**(***kwargs*)
Handle SMIRNOFF <Bonds> tags

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

Attributes

TAGNAME The name of this ParameterHandler corresponding to the SMIRNOFF tag name

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

BondType(**kwargs)	A SMIRNOFF bond type
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter([parameter_kwargs, parameter, ...])	Add a parameter to the forcefield, ensuring all parameters are valid.

continues on next page

Table 75 – 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__(***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__(**kwargs)	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, parameter, ...])	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(...)	

continues on next page

Table 76 – continued from previous page

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.
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

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
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.

class BondType(***kwargs*)

A SMIRNOFF bond type

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

Attributes

id 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()
```

(continues on next page)

(continued from previous page)

```
...     @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

k 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()
```

(continues on next page)

(continued from previous page)

```
...     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'
```

(continues on next page)

(continued from previous page)

```
>>> 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

k_bondorder The attribute of a parameter in which each term is a mapping.

The substantial difference with `IndexedParameterAttribute` is that, unlike indexing, the mapping can be based on artbitrary references, like indices but can starting at non-zero values and include non-adjacent keys.

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 mappings where values are quantities with compatible units are allowed to be set.

converter [callable, optional] An optional function that can be used to validate and cast each component of each element of the sequence before setting the attribute.

IndexedParameterAttribute A parameter attribute representing a sequence.

IndexedMappedParameterAttribute A parameter attribute representing a sequence, each term of which is a mapping.

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = MappedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Like other ParameterAttribute objects, strings are parsed into Quantity objects.

```
>>> my_par.length = {1:'1.5 * angstrom', 2: '1.4 * angstrom'}
>>> my_par.length[1]
Quantity(value=1.5, unit=angstrom)
```

Unlike other ParameterAttribute objects, the reference points can do not need ot be zero-indexed, non-adjancent, such as interpolating defining a bond parameter for interpolation by defining references values and bond orders 2 and 3:

```
>>> my_par.length = {2:'1.42 * angstrom', 3: '1.35 * angstrom'}
>>> my_par.length[2]
Quantity(value=1.42, unit=angstrom)
```

length 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

length_bondorder The attribute of a parameter in which each term is a mapping.

The substantial difference with `IndexedParameterAttribute` is that, unlike indexing, the mapping can be based on artbitrary references, like indices but can starting at non-zero values and include non-adjacent keys.

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 mappings where values are quantities with compatible units are allowed to be set.

converter [callable, optional] An optional function that can be used to validate and cast each component of each element of the sequence before setting the attribute.

IndexedParameterAttribute A parameter attribute representing a sequence.

IndexedMappedParameterAttribute A parameter attribute representing a sequence, each term of which is a mapping.

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = MappedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
```

(continues on next page)

(continued from previous page)

```
>>> my_par.length is None
True
```

Like other ParameterAttribute objects, strings are parsed into Quantity objects.

```
>>> my_par.length = {1:'1.5 * angstrom', 2: '1.4 * angstrom'}
>>> my_par.length[1]
Quantity(value=1.5, unit=angstrom)
```

Unlike other ParameterAttribute objects, the reference points can do not need ot be zero-indexed, non-adjancent, such as interpolating defining a bond parameter for interpolation by defining references values and bond orders 2 and 3:

```
>>> my_par.length = {2:'1.42 * angstrom', 3: '1.35 * angstrom'}
>>> my_par.length[2]
Quantity(value=1.42, unit=angstrom)
```

parent_id 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

smirks 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.
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.

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the attribute to define for this object.

attr_value [str] The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters
attr_name [str] The attribute name to check
Returns
is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)
Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters
attr_name [str] Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)
Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters
discard_cosmetic_attributes [bool, optional. Default = False] Whether to discard non-spec attributes of this object
duplicate_attributes [list of string, optional. Default = None] A list of names of attributes that redundantly decsribe data and should be discarded during serialization
Returns
smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler*)
Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters

other_handler [a ParameterHandler object] The handler to compare to.

Raises

IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.

property TAGNAME
The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns

handler_name [str] The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)
Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

Warning: The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the attribute to define for this object.

attr_value [str] The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)
Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs: dict, optional The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

parameter: ParameterType, optional A ParameterType to add to the ParameterHandler

after [str or int, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added

before [str, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added

Note that one of (parameter_kwargs, parameter) must be specified

Note that when `before` and `after` are both None, the new parameter will be appended to the END of the parameter list.

Note that when `before` and `after` are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from simtk import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie_per_mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

assign_parameters(*topology*, *system*)
Assign parameters for the given Topology to the specified System object.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

attribute_is_cosmetic(*attr_name*)
Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] The attribute name to check

Returns

is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)
Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the cosmetic attribute to delete.

find_matches(*entity*)
Find the elements of the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology] Topology to search.

Returns

matches [ValenceDict[Tuple[int], ParameterHandler._Match]] `matches[particle_indices]` is the `ParameterType` object matching the tuple of particle indices in `entity`.

get_parameter(*parameter_attrs*)
Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters

parameter_attrs [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns

params [list of ParameterType objects] A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from simtk import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie_per_mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

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

property parameters
The ParameterList that holds this ParameterHandler's parameter objects

postprocess_system(*topology*, *system*, ***kwargs*)
Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

to_dict(*discard_cosmetic_attributes=False*)
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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

TAGNAME The name of this ParameterHandler corresponding to the SMIRNOFF tag name

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)
...     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, parameter, ...])	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.

continues on next page

Table 79 – continued from previous page

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, parameter, ...])	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.
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.

continues on next page

Table 80 – continued from previous page

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

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
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.

class AngleType(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

A SMIRNOFF angle type.

Warning: This API is experimental and subject to change.

Attributes

angle 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

id 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

k 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

parent_id 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

smirks 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

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.

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters
attr_name [str] Name of the attribute to define for this object.
attr_value [str] The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)
Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters
attr_name [str] The attribute name to check
Returns
is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)
Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters
attr_name [str] Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)
Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters
discard_cosmetic_attributes [bool, optional. Default = False] Whether to discard non-spec attributes of this object
duplicate_attributes [list of string, optional. Default = None] A list of names of attributes that redundantly decsribe data and should be discarded during serialization
Returns
smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler*)
Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters

other_handler [a ParameterHandler object] The handler to compare to.

Raises

IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.

property TAGNAME
The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns

handler_name [str] The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)
Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the attribute to define for this object.

attr_value [str] The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)
Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs: dict, optional The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

parameter: ParameterType, optional A ParameterType to add to the ParameterHandler

after [str or int, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added

before [str, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added

Note that one of (parameter_kwargs, parameter) must be specified

Note that when `before` and `after` are both None, the new parameter will be appended to the END of the parameter list.

Note that when `before` and `after` are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from simtk import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie_per_mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2'})
```

(continues on next page)

(continued from previous page)

```
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

assign_parameters(*topology*, *system*)

Assign parameters for the given Topology to the specified System object.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] The attribute name to check

Returns

is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the cosmetic attribute to delete.

find_matches(*entity*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology] Topology to search.

Returns

matches [ValenceDict[Tuple[int], ParameterHandler._Match]] `matches[particle_indices]` is the `ParameterType` object matching the tuple of particle indices in `entity`.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters

parameter_attrs [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns

params [list of ParameterType objects] A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from simtk import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie_per_mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

property known_kwargs

List of kwargs that can be parsed by the function.

property parameters

The ParameterList that holds this ParameterHandler's parameter objects

postprocess_system(*topology*, *system*, ***kwargs*)

Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

to_dict(*discard_cosmetic_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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

TAGNAME The name of this ParameterHandler corresponding to the SMIRNOFF tag name

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
...
>>> 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):
```

(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

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
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, parameter, ...])	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.

continues on next page

Table 83 – continued from previous page

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, parameter, ...])	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.
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.

continues on next page

Table 84 – continued from previous page

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

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
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.

class ProperTorsionType(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

A SMIRNOFF torsion type for proper torsions.

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

Attributes

id 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

idivf 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.

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.

ParameterAttribute A simple parameter attribute.

MappedParameterAttribute A parameter attribute representing a mapping.

IndexedMappedParameterAttribute A parameter attribute representing a sequence, each term of which is a mapping.

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]
```

k 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.

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.

ParameterAttribute A simple parameter attribute.

MappedParameterAttribute A parameter attribute representing a mapping.

IndexedMappedParameterAttribute A parameter attribute representing a sequence, each term of which is a mapping.

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]
```

k_bondorder The attribute of a parameter with an unspecified number of terms, where each term is a mapping.

Some parameters can be associated to multiple terms, where those terms have multiple components. For example, torsions with fractional bond orders have parameters such as k1_bondorder1, k1_bondorder2, k2_bondorder1, k2_bondorder2, . . . , and `IndexedMappedParameterAttribute` can be used to encapsulate the sequence of terms as mappings (typically, `dict`s) of their components.

The only substantial difference with `IndexedParameterAttribute` is that only sequences of mappings are supported as values and converters and units are checked on each component of each element in 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.

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 mappings where values are quantities with compatible units are allowed to be set.

converter [callable, optional] An optional function that can be used to validate and cast each component of each element of the sequence before setting the attribute.

IndexedParameterAttribute A parameter attribute representing a sequence.

MappedParameterAttribute A parameter attribute representing a mapping.

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedMappedParameterAttribute(default=None, unit=unit.
↪angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into Quantity objects.

```
>>> my_par.length = [{1:'1 * angstrom'}, {1: 0.5 * unit.nanometer}]
>>> my_par.length[0]
{1: 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 = IndexedMappedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [{1: 1}, {2: '1.0', 3: '1e-2'}, {4: 4.0}]
>>> my_par.attr_indexed
[{1: 1.0}, {2: 1.0, 3: 0.01}, {4: 4.0}]
```

parent_id 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

periodicity 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.

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.

ParameterAttribute A simple parameter attribute.

MappedParameterAttribute A parameter attribute representing a mapping.

IndexedMappedParameterAttribute A parameter attribute representing a sequence, each term of which is a mapping.

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]
```

phase 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.

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.

ParameterAttribute A simple parameter attribute.

MappedParameterAttribute A parameter attribute representing a mapping.

IndexedMappedParameterAttribute A parameter attribute representing a sequence, each term of which is a mapping.

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]
```

smirks 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

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.

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the attribute to define for this object.

attr_value [str] The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] The attribute name to check

Returns

is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)
Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters
attr_name [str] Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)
Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters
discard_cosmetic_attributes [bool, optional. Default = False] Whether to discard non-spec attributes of this object
duplicate_attributes [list of string, optional. Default = None] A list of names of attributes that redundantly decsribe data and should be discarded during serialization

Returns
smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler*)
Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters

other_handler [a ParameterHandler object] The handler to compare to.

Raises

IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.

property TAGNAME
The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns

handler_name [str] The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)
Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the attribute to define for this object.

attr_value [str] The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)
Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs: dict, optional The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

parameter: ParameterType, optional A ParameterType to add to the ParameterHandler

after [str or int, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added

before [str, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added

Note that one of (parameter_kwargs, parameter) must be specified

Note that when `before` and `after` are both None, the new parameter will be appended to the END of the parameter list.

Note that when `before` and `after` are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from simtk import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie_per_mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

assign_parameters(*topology*, *system*)
Assign parameters for the given Topology to the specified System object.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

attribute_is_cosmetic(*attr_name*)
Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] The attribute name to check

Returns

is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)
Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the cosmetic attribute to delete.

find_matches(*entity*)
Find the elements of the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology] Topology to search.

Returns

matches [ValenceDict[Tuple[int], ParameterHandler._Match]] `matches[particle_indices]` is the `ParameterType` object matching the tuple of particle indices in `entity`.

get_parameter(*parameter_attrs*)
Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters

parameter_attrs [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*](:4)", "id": "t105"})

Returns

params [list of ParameterType objects] A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from simtk import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie_per_mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

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

property parameters
The ParameterList that holds this ParameterHandler's parameter objects

postprocess_system(*topology*, *system*, ***kwargs*)
Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

to_dict(*discard_cosmetic_attributes=False*)
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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

TAGNAME The name of this ParameterHandler corresponding to the SMIRNOFF tag name

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
...
>>> 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):
```

(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

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, parameter, ...])	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 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 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, parameter, ...])	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.
`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 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

`TAGNAME`	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
`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.

class ImproperTorsionType(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

A SMIRNOFF torsion type for improper torsions.

Warning: This API is experimental and subject to change.

Attributes

id 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

idivf 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.

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.

ParameterAttribute A simple parameter attribute.

MappedParameterAttribute A parameter attribute representing a mapping.

IndexedMappedParameterAttribute A parameter attribute representing a sequence, each term of which is a mapping.

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]
```

k 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.

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.

ParameterAttribute A simple parameter attribute.

MappedParameterAttribute A parameter attribute representing a mapping.

IndexedMappedParameterAttribute A parameter attribute representing a sequence, each term of which is a mapping.

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]
```

parent_id 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

periodicity 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.

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.

ParameterAttribute A simple parameter attribute.

MappedParameterAttribute A parameter attribute representing a mapping.

IndexedMappedParameterAttribute A parameter attribute representing a sequence, each term of which is a mapping.

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]
```

phase 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.

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.

ParameterAttribute A simple parameter attribute.

MappedParameterAttribute A parameter attribute representing a mapping.

IndexedMappedParameterAttribute A parameter attribute representing a sequence, each term of which is a mapping.

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]
```

smirks 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.
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.

add_cosmetic_attribute(*attr_name*, *attr_value*)
Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters
attr_name [str] Name of the attribute to define for this object.
attr_value [str] The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)
Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters
attr_name [str] The attribute name to check
Returns
is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)
Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters
attr_name [str] Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)
Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters
discard_cosmetic_attributes [bool, optional. Default = False] Whether to discard non-spec attributes of this object

duplicate_attributes [list of string, optional. Default = None] A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns

smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters

other_handler [a ParameterHandler object] The handler to compare to.

Raises

IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.

find_matches(*entity*)

Find the improper torsions in the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology] Topology to search.

Returns

matches [ImproperDict[Tuple[int], ParameterHandler._Match]] `matches[atom_indices]` is the `ParameterType` object matching the 4-tuple of atom indices in `entity`.

property TAGNAME

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns

handler_name [str] The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the attribute to define for this object.

attr_value [str] The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs: dict, optional The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

parameter: ParameterType, optional A ParameterType to add to the ParameterHandler

after [str or int, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added

before [str, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added

Note that one of (parameter_kwargs, parameter) must be specified

Note that when `before` and `after` are both None, the new parameter will be appended to the END of the parameter list.

Note that when `before` and `after` are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from simtk import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie_per_mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

assign_parameters(*topology*, *system*)

Assign parameters for the given Topology to the specified System object.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

Warning: The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] The attribute name to check

Returns

is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the cosmetic attribute to delete.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters

parameter_attrs [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns

params [list of ParameterType objects] A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from simtk import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie_per_mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

property known_kwargs

List of kwargs that can be parsed by the function.

property parameters

The ParameterList that holds this ParameterHandler's parameter objects

postprocess_system(*topology*, *system*, ***kwargs*)
Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

to_dict(*discard_cosmetic_attributes=False*)
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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

TAGNAME The name of this ParameterHandler corresponding to the SMIRNOFF tag name

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, parameter, ...])	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(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.
vdWType(**kwargs)	A SMIRNOFF vdWForce type.

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, parameter, ...])	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.
`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.
`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.
`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

`TAGNAME`	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
`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.

continues on next page

Table 93 – continued from previous page

scale15	A descriptor for ParameterType attributes.
switch_width	A descriptor for ParameterType attributes.
version	A descriptor for ParameterType attributes.

class vdWType(***kwargs*)
A SMIRNOFF vdWForce type.

Warning: This API is experimental and subject to change.

Attributes

epsilon 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

id 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

parent_id 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

rmin_half 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

sigma 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

smirks 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.
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.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)
Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters
discard_cosmetic_attributes [bool, optional. Default = False] Whether to discard non-spec attributes of this object
duplicate_attributes [list of string, optional. Default = None] A list of names of attributes that redundantly decsribe data and should be discarded during serialization

Returns
smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this object.

add_cosmetic_attribute(*attr_name*, *attr_value*)
Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters
attr_name [str] Name of the attribute to define for this object.
attr_value [str] The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)
Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters
attr_name [str] The attribute name to check
Returns
is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)
Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the cosmetic attribute to delete.

`check_handler_compatibility`(*other_handler*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters

other_handler [a ParameterHandler object] The handler to compare to.

Raises

IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.

`property TAGNAME`

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns

handler_name [str] The name of this parameter handler

`add_cosmetic_attribute`(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the attribute to define for this object.

attr_value [str] The value of the attribute to define for this object.

`add_parameter`(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs: dict, optional The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

parameter: ParameterType, optional A ParameterType to add to the ParameterHandler

after [str or int, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added

before [str, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added

Note that one of (parameter_kwargs, parameter) must be specified

Note that when `before` and `after` are both None, the new parameter will be appended to the END of the parameter list.

Note that when `before` and `after` are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from simtk import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie_per_mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

assign_parameters(*topology*, *system*)
Assign parameters for the given Topology to the specified System object.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

attribute_is_cosmetic(*attr_name*)
Determine whether an attribute of this object is cosmetic.

Warning: The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] The attribute name to check

Returns

is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

static check_charges_assigned(*ref_mol*, *topology*)
Check whether charges have been assigned for a reference molecule.

Parameters

ref_mol [openforcefield.topology.Molecule] The molecule to check for having charges assigned

topology [openforcefield.topology.Topology] The topology to query for this information

Returns

charges_assigned [bool] Whether charges have already been assigned to this molecule

delete_cosmetic_attribute(*attr_name*)
Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the cosmetic attribute to delete.

find_matches(*entity*)
Find the elements of the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology] Topology to search.

Returns

matches [ValenceDict[Tuple[int], ParameterHandler._Match]] `matches[particle_indices]` is the `ParameterType` object matching the tuple of particle indices in `entity`.

get_parameter(*parameter_attrs*)
Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters

parameter_attrs [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns

params [list of ParameterType objects] A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from simtk import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie_per_mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

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

mark_charges_assigned(*ref_mol*, *topology*)
Record that charges have been assigned for a reference molecule.

Parameters

ref_mol [openforcefield.topology.Molecule] The molecule to mark as having charges assigned

topology [openforcefield.topology.Topology] The topology to record this information on.

property parameters
The ParameterList that holds this ParameterHandler's parameter objects

postprocess_system(*topology*, *system*, ***kwargs*)
Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

to_dict(*discard_cosmetic_attributes=False*)
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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

TAGNAME The name of this ParameterHandler corresponding to the SMIRNOFF tag name

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)
...     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
```

(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

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, parameter, ...])	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.

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, parameter, ...])	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.
`assign_partial_bond_orders_from_molecules`(...)	
`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.
`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.
`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

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
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.

check_handler_compatibility(*other_handler*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters

other_handler [a ParameterHandler object] The handler to compare to.

Raises

IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.

assign_charge_from_molecules(*molecule*, *charge_mols*)

Given an input molecule, checks against a list of molecules for an isomorphic match. If found, assigns partial charges from the match to the input molecule.

Parameters

molecule [an openforcefield.topology.FrozenMolecule] The molecule to have partial charges assigned if a match is found.

charge_mols [list of [openforcefield.topology.FrozenMolecule]] A list of molecules with charges already assigned.

Returns

match_found [bool] Whether a match was found. If True, the input molecule will have been modified in-place.

postprocess_system(*system*, *topology*, ***kwargs*)

Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

property TAGNAME
The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns

handler_name [str] The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)
Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the attribute to define for this object.

attr_value [str] The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)
Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs: dict, optional The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

parameter: ParameterType, optional A ParameterType to add to the ParameterHandler

after [str or int, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added

before [str, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added

Note that one of (parameter_kwargs, parameter) must be specified

Note that when `before` and `after` are both None, the new parameter will be appended to the END of the parameter list.

Note that when `before` and `after` are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from simtk import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie_per_mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2'})
```

(continues on next page)

(continued from previous page)

```
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

assign_parameters(*topology*, *system*)
Assign parameters for the given Topology to the specified System object.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

attribute_is_cosmetic(*attr_name*)
Determine whether an attribute of this object is cosmetic.

Warning: The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] The attribute name to check

Returns

is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

static check_charges_assigned(*ref_mol*, *topology*)
Check whether charges have been assigned for a reference molecule.

Parameters

ref_mol [openforcefield.topology.Molecule] The molecule to check for having charges assigned

topology [openforcefield.topology.Topology] The topology to query for this information

Returns

charges_assigned [bool] Whether charges have already been assigned to this molecule

delete_cosmetic_attribute(*attr_name*)
Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the cosmetic attribute to delete.

find_matches(*entity*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology] Topology to search.

Returns

matches [ValenceDict[Tuple[int], ParameterHandler._Match]] `matches[particle_indices]` is the `ParameterType` object matching the tuple of particle indices in `entity`.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters

parameter_attrs [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "*[:1]~[#16:2]=,:[#6:3]~[*:4]*", "id": "t105"})

Returns

params [list of ParameterType objects] A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from simtk import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie_per_mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

property known_kwargs

List of kwargs that can be parsed by the function.

mark_charges_assigned(*ref_mol*, *topology*)

Record that charges have been assigned for a reference molecule.

Parameters

ref_mol [openforcefield.topology.Molecule] The molecule to mark as having charges assigned

topology [openforcefield.topology.Topology] The topology to record this information on.

property parameters
The ParameterList that holds this ParameterHandler's parameter objects

to_dict(*discard_cosmetic_attributes=False*)
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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

TAGNAME The name of this ParameterHandler corresponding to the SMIRNOFF tag name

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, parameter, ...])	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.

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, parameter, ...])	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.
`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.
`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.
`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

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
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.

class LibraryChargeType(***kwargs*)

A SMIRNOFF Library Charge type.

Warning: This API is experimental and subject to change.

Attributes

charge 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.

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.

ParameterAttribute A simple parameter attribute.

MappedParameterAttribute A parameter attribute representing a mapping.

IndexedMappedParameterAttribute A parameter attribute representing a sequence, each term of which is a mapping.

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]
```

id 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

name 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

parent_id 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

smirks 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.
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.

add_cosmetic_attribute(*attr_name*, *attr_value*)
Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters
attr_name [str] Name of the attribute to define for this object.
attr_value [str] The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)
Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters
attr_name [str] The attribute name to check
Returns
is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)
Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters
attr_name [str] Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)
Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters
discard_cosmetic_attributes [bool, optional. Default = False] Whether to discard non-spec attributes of this object
duplicate_attributes [list of string, optional. Default = None] A list of names of attributes that redundantly decsribe data and should be discarded during serialization
Returns
smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this object.

find_matches(*entity*)
Find the elements of the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology] Topology to search.

Returns

matches [ValenceDict[Tuple[int], ParameterHandler._Match]] `matches[particle_indices]` is the `ParameterType` object matching the tuple of particle indices in `entity`.

property TAGNAME

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns

handler_name [str] The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the attribute to define for this object.

attr_value [str] The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs: dict, optional The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

parameter: ParameterType, optional A ParameterType to add to the ParameterHandler

after [str or int, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added

before [str, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added

Note that one of (parameter_kwargs, parameter) must be specified

Note that when `before` and `after` are both None, the new parameter will be appended to the END of the parameter list.

Note that when `before` and `after` are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from simtk import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie_per_mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

assign_parameters(*topology*, *system*)
Assign parameters for the given Topology to the specified System object.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

attribute_is_cosmetic(*attr_name*)
Determine whether an attribute of this object is cosmetic.

Warning: The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] The attribute name to check

Returns

is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

static check_charges_assigned(*ref_mol*, *topology*)
Check whether charges have been assigned for a reference molecule.

Parameters

ref_mol [openforcefield.topology.Molecule] The molecule to check for having charges assigned

topology [openforcefield.topology.Topology] The topology to query for this information

Returns

charges_assigned [bool] Whether charges have already been assigned to this molecule

check_handler_compatibility(*handler_kwargs*)

Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters

handler_kwargs [dict] The kwargs that would be used to construct

Raises

IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the cosmetic attribute to delete.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters

parameter_attrs [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns

params [list of ParameterType objects] A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from simtk import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie_per_mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

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

mark_charges_assigned(*ref_mol*, *topology*)
Record that charges have been assigned for a reference molecule.

Parameters

ref_mol [openforcefield.topology.Molecule] The molecule to mark as having charges assigned

topology [openforcefield.topology.Topology] The topology to record this information on.

property parameters
The ParameterList that holds this ParameterHandler's parameter objects

postprocess_system(*topology*, *system*, ***kwargs*)
Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

to_dict(*discard_cosmetic_attributes=False*)
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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

TAGNAME The name of this ParameterHandler corresponding to the SMIRNOFF tag name

`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'
```

(continues on next page)

(continued from previous page)

```
>>> 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, parameter, ...])	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.

continues on next page

Table 102 – continued from previous page

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.

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, parameter, ...])	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 103 – continued from previous page

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.
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.
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

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
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.

check_handler_compatibility(*other_handler*, *assume_missing_is_default=True*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters

other_handler [a ParameterHandler object] The handler to compare to.

Raises

IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.

postprocess_system(*system*, *topology*, ***kwargs*)

Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

property TAGNAME
The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns

handler_name [str] The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)
Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the attribute to define for this object.

attr_value [str] The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)
Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs: dict, optional The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

parameter: ParameterType, optional A ParameterType to add to the ParameterHandler

after [str or int, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added

before [str, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added

Note that one of (parameter_kwargs, parameter) must be specified

Note that when `before` and `after` are both None, the new parameter will be appended to the END of the parameter list.

Note that when `before` and `after` are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from simtk import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie_per_mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

assign_parameters(*topology*, *system*)
Assign parameters for the given Topology to the specified System object.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

attribute_is_cosmetic(*attr_name*)
Determine whether an attribute of this object is cosmetic.

Warning: The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] The attribute name to check

Returns

is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

static check_charges_assigned(*ref_mol*, *topology*)
Check whether charges have been assigned for a reference molecule.

Parameters

ref_mol [openforcefield.topology.Molecule] The molecule to check for having charges assigned

topology [openforcefield.topology.Topology] The topology to query for this information

Returns

charges_assigned [bool] Whether charges have already been assigned to this molecule

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the cosmetic attribute to delete.

find_matches(*entity*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology] Topology to search.

Returns

matches [ValenceDict[Tuple[int], ParameterHandler._Match]] `matches[particle_indices]` is the `ParameterType` object matching the tuple of particle indices in `entity`.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters

parameter_attrs [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns

params [list of ParameterType objects] A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from simtk import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie_per_mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

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

mark_charges_assigned(*ref_mol*, *topology*)
Record that charges have been assigned for a reference molecule.

Parameters

ref_mol [openforcefield.topology.Molecule] The molecule to mark as having charges assigned

topology [openforcefield.topology.Topology] The topology to record this information on.

property parameters
The ParameterList that holds this ParameterHandler's parameter objects

to_dict(*discard_cosmetic_attributes=False*)
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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

TAGNAME The name of this ParameterHandler corresponding to the SMIRNOFF tag name

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):
...
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

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):
...             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

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):
```

(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

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
>>> 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):
```

(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

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, parameter, ...])	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.

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, parameter, ...])	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.
`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

`TAGNAME`	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
`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.
`solvent_dielectric`	A descriptor for `ParameterType` attributes.

continues on next page

Table 107 – continued from previous page

solvent_radius	A descriptor for ParameterType attributes.
surface_area_penalty	A descriptor for ParameterType attributes.
version	A descriptor for ParameterType attributes.

class GBSAType(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

A SMIRNOFF GBSA type.

Warning: This API is experimental and subject to change.

Attributes

id 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

parent_id 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

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

scale 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

smirks 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

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.

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the attribute to define for this object.

attr_value [str] The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters
attr_name [str] The attribute name to check
Returns
is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)
Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters
attr_name [str] Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)
Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters
discard_cosmetic_attributes [bool, optional. Default = False] Whether to discard non-spec attributes of this object
duplicate_attributes [list of string, optional. Default = None] A list of names of attributes that redundantly decsribe data and should be discarded during serialization
Returns
smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler*)
Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters

other_handler [a ParameterHandler object] The handler to compare to.

Raises

IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.

property TAGNAME
The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns

handler_name [str] The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)
Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the Open Force Field toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the attribute to define for this object.

attr_value [str] The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)
Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs: dict, optional The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

parameter: ParameterType, optional A ParameterType to add to the ParameterHandler

after [str or int, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added

before [str, optional] The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added

Note that one of (parameter_kwargs, parameter) must be specified

Note that when `before` and `after` are both None, the new parameter will be appended to the END of the parameter list.

Note that when `before` and `after` are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from simtk import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie_per_mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

assign_parameters(*topology*, *system*)
Assign parameters for the given Topology to the specified System object.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

attribute_is_cosmetic(*attr_name*)
Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] The attribute name to check

Returns

is_cosmetic [bool] Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)
Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

attr_name [str] Name of the cosmetic attribute to delete.

find_matches(*entity*)
Find the elements of the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology] Topology to search.

Returns

matches [ValenceDict[Tuple[int], ParameterHandler._Match]] `matches[particle_indices]` is the `ParameterType` object matching the tuple of particle indices in `entity`.

get_parameter(*parameter_attrs*)
Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters

parameter_attrs [dict of {attr: value}] The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns

params [list of ParameterType objects] A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from simtk import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie_per_mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

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

property parameters
The ParameterList that holds this ParameterHandler's parameter objects

postprocess_system(*topology*, *system*, ***kwargs*)
Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.

Parameters

topology [openforcefield.topology.Topology] The Topology for which parameters are to be assigned. Either a new Force will be created or parameters will be appended to an existing Force.

system [simtk.openmm.System] The OpenMM System object to add the Force (or append new parameters) to.

to_dict(*discard_cosmetic_attributes=False*)
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

discard_cosmetic_attributes [bool, optional. Default = False.] Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns

smirnoff_data [OrderedDict] SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

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

parse_file(*file_path*)

Parameters

file_path

parse_string(*data*)

Parse a SMIRNOFF force field definition in a seriaized format

Parameters

data

to_file(*file_path*, *smirnoff_data*)
Write the current forcefield parameter set to a file.

Parameters

file_path [str] The path to the file to write to.

smirnoff_data [dict] A dictionary structured in compliance with the SMIRNOFF spec

to_string(*smirnoff_data*)
Render the forcefield parameter set to a string

Parameters

smirnoff_data [dict] A dictionary structured in compliance with the SMIRNOFF spec

Returns

str

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.

parse_file(*source*)

Parse a SMIRNOFF force field definition in XML format, read from a file.

Parameters

source [str or io.RawIOBase] File path of file-like object implementing a `read()` method specifying a SMIRNOFF force field definition in the SMIRNOFF XML format.

Raises

ParseError If the XML cannot be processed.

FileNotFoundError If the file could not found.

parse_string(*data*)

Parse a SMIRNOFF force field definition in XML format.

A `ParseError` is raised if the XML cannot be processed.

Parameters

data [str] A SMIRNOFF force field definition in the SMIRNOFF XML format.

to_file(*file_path*, *smirnoff_data*)

Write the current forcefield parameter set to a file.

Parameters

file_path [str] The path to the file to be written. The *.offxml* or *.xml* file extension must be present.

smirnoff_data [dict] A dict structured in compliance with the SMIRNOFF data spec.

to_string(*smirnoff_data*)

Write the current forcefield parameter set to an XML string.

Parameters

smirnoff_data [dict] A dictionary structured in compliance with the SMIRNOFF spec

Returns

serialized_forcefield [str] XML String representation of this forcefield.

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.
`MappedParameterAttribute`	The attribute of a parameter in which each term is a mapping.
`IndexedMappedParameterAttribute`	The attribute of a parameter with an unspecified number of terms, where each term is a mapping.

openforcefield.typing.engines.smirnoff.parameters.ParameterAttribute

class openforcefield.typing.engines.smirnoff.parameters.**ParameterAttribute**(*default=<class 'openforcefield.typing.engines.smirnoff.parameters.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

class UNDEFINED
Custom type used by ParameterAttribute to differentiate between None and undeclared default.

converter(*converter*)
Create a new ParameterAttribute with an associated converter.

This is meant to be used as a decorator (see main examples).

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.

`MappedParameterAttribute` A parameter attribute representing a mapping.

`IndexedMappedParameterAttribute` A parameter attribute representing a sequence, each term of which is a mapping.

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

class UNDEFINED
Custom type used by ParameterAttribute to differentiate between None and undeclared default.

converter(*converter*)
Create a new ParameterAttribute with an associated converter.

This is meant to be used as a decorator (see main examples).

openforcefield.typing.engines.smirnoff.parameters.MappedParameterAttribute

class openforcefield.typing.engines.smirnoff.parameters.**MappedParameterAttribute**(*default=<class 'openforcefield.typing.engines.smirnoff.para unit=None*, *converter=None*)

The attribute of a parameter in which each term is a mapping.

The substantial difference with IndexedParameterAttribute is that, unlike indexing, the mapping can be based on artbitrary references, like indices but can starting at non-zero values and include non-adjacent keys.

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 mappings where values are quantities with compatible units are allowed to be set.

converter [callable, optional] An optional function that can be used to validate and cast each component of each element of the sequence before setting the attribute.

See also:

`IndexedParameterAttribute` A parameter attribute representing a sequence.

`IndexedMappedParameterAttribute` A parameter attribute representing a sequence, each term of which is a mapping.

Examples

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = MappedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Like other ParameterAttribute objects, strings are parsed into Quantity objects.

```
>>> my_par.length = {1:'1.5 * angstrom', 2: '1.4 * angstrom'}
>>> my_par.length[1]
Quantity(value=1.5, unit=angstrom)
```

Unlike other ParameterAttribute objects, the reference points can do not need ot be zero-indexed, non-adjancent, such as interpolating defining a bond parameter for interpolation by defining references values and bond orders 2 and 3:

```
>>> my_par.length = {2:'1.42 * angstrom', 3: '1.35 * angstrom'}
>>> my_par.length[2]
Quantity(value=1.42, unit=angstrom)
```

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`

class UNDEFINED
Custom type used by `ParameterAttribute` to differentiate between `None` and undeclared default.

converter(*converter*)
Create a new ParameterAttribute with an associated converter.

This is meant to be used as a decorator (see main examples).

openforcefield.typing.engines.smirnoff.parameters.IndexedMappedParameterAttribute

class openforcefield.typing.engines.smirnoff.parameters.**IndexedMappedParameterAttribute**(*default=<class 'openforcefield.typing.engines.smir*, *unit=None*, *converter=None*)

The attribute of a parameter with an unspecified number of terms, where each term is a mapping.

Some parameters can be associated to multiple terms, where those terms have multiple components. For example, torsions with fractional bond orders have parameters such as k1_bondorder1, k1_bondorder2, k2_bondorder1, k2_bondorder2, ..., and `IndexedMappedParameterAttribute` can be used to encapsulate the sequence of terms as mappings (typically, `dict`s) of their components.

The only substantial difference with `IndexedParameterAttribute` is that only sequences of mappings are supported as values and converters and units are checked on each component of each element in 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 mappings where values are quantities with compatible units are allowed to be set.

converter [callable, optional] An optional function that can be used to validate and cast each component of each element of the sequence before setting the attribute.

See also:

IndexedParameterAttribute A parameter attribute representing a sequence.

MappedParameterAttribute A parameter attribute representing a mapping.

Examples

Create an optional indexed attribute with unit of angstrom.

```
>>> from simtk import unit
>>> class MyParameter:
...     length = IndexedMappedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into Quantity objects.

```
>>> my_par.length = [{1:'1 * angstrom'}, {1: 0.5 * unit.nanometer}]
>>> my_par.length[0]
{1: 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 = IndexedMappedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [{1: 1}, {2: '1.0', 3: '1e-2'}, {4: 4.0}]
>>> my_par.attr_indexed
[{1: 1.0}, {2: 1.0, 3: 0.01}, {4: 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`

class UNDEFINED
Custom type used by `ParameterAttribute` to differentiate between `None` and undeclared default.

converter(*converter*)
Create a new ParameterAttribute with an associated converter.

This is meant to be used as a decorator (see main examples).

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, OpenEyeToolkitWrapper,␣
↪RDKitToolkitWrapper, AmberToolsToolkitWrapper
>>> toolkit_registry = ToolkitRegistry()
>>> toolkit_precedence = [OpenEyeToolkitWrapper, RDKitToolkitWrapper, AmberToolsToolkitWrapper]
>>> [ toolkit_registry.register_toolkit(toolkit) for toolkit in toolkit_precedence if toolkit.is_
↪available() ]
[None, None, None]
```

The toolkit wrappers can then be accessed through the registry:

```
>>> from openforcefield.utils.toolkits import GLOBAL_TOOLKIT_REGISTRY as toolkit_registry
>>> from openforcefield.topology.molecule import Molecule
>>> molecule = Molecule.from_smiles('Cc1ccccc1')
>>> smiles = toolkit_registry.call('to_smiles', molecule)
```

The order of toolkits, as specified in `toolkit_precedence` above, determines the order in which the called method is resolved, i.e. if the toolkit with highest precedence has a `to_smiles` method, that is the toolkit that will be called. If the toolkit with highest precedence does not have such a method, it is attempted with other toolkits until one is found. By default, if a toolkit with an appropriately-named method raises an exception of any type, then iteration over the registered toolkits stops and that exception is raised. To continue iteration if specific exceptions are encountered, customize this behavior using the optional `raise_exception_types` keyword argument to `ToolkitRegistry.call`. If no registered toolkits have the method, a ValueError is raised, containing a message listing the registered toolkits and exceptions (if any) that were ignored.

Alternatively, the global toolkit registry (which will attempt to register any available toolkits) can be used:

```
>>> from openforcefield.utils.toolkits import GLOBAL_TOOLKIT_REGISTRY as toolkit_registry
>>> len(toolkit_registry.registered_toolkits)
4
```

Individual toolkits can be registered or deregistered to control the backend that ToolkitRegistry calls resolve to. This can be useful for debugging and exploring subtley different behavior between toolkit wrappers.

```
from openforcefield.utils.toolkits import OpenEyeToolkitWrapper, BuiltInToolkitWrapper
from openforcefield.utils.toolkits import GLOBAL_TOOLKIT_REGISTRY as toolkit_registry
toolkit_registry.deregister_toolkit(OpenEyeToolkitWrapper)
toolkit_registry.register_toolkit(BuiltInToolkitWrapper)
toolkit_registry.registered_toolkits
```

For example, differences in `to_smiles` functionality between OpenEye toolkits and The RDKit can be explored by selecting which toolkit(s) are and are not registered.

```
>>> from openforcefield.utils.toolkits import OpenEyeToolkitWrapper, GLOBAL_TOOLKIT_REGISTRY as toolkit_
↪registry
>>> from openforcefield.topology.molecule import Molecule
>>> molecule = Molecule.from_smiles('Cc1ccccc1')
>>> smiles_via_openeye = toolkit_registry.call('to_smiles', molecule)
>>> print(smiles_via_openeye)
[H]c1c(c(c(c(c1[H])[H])C([H])([H])[H])[H])[H]

>>> toolkit_registry.deregister_toolkit(OpenEyeToolkitWrapper)
>>> smiles_via_rdkit = toolkit_registry.call('to_smiles', molecule)
>>> print(smiles_via_rdkit)
[H][c]1[c]([H])[c]([H])[c]([C]([H])([H])[H])[c]([H])[c]1[H]
```

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**(*toolkit_precedence=[]*, *exception_if_unavailable=True*, *_register_imported_toolkit_wrappers=False*)

Registry for ToolkitWrapper objects

Examples

Register toolkits in a specified order, skipping if unavailable

```
>>> from openforcefield.utils.toolkits import ToolkitRegistry
>>> toolkit_precedence = [OpenEyeToolkitWrapper, RDKitToolkitWrapper, AmberToolsToolkitWrapper]
>>> toolkit_registry = ToolkitRegistry(toolkit_precedence)
>>> toolkit_registry
ToolkitRegistry containing OpenEye Toolkit, The RDKit, AmberTools
```

Register all available toolkits (in the order OpenEye, RDKit, AmberTools, built-in)

```
>>> toolkits = [OpenEyeToolkitWrapper, RDKitToolkitWrapper, AmberToolsToolkitWrapper,␣
↪BuiltInToolkitWrapper]
>>> toolkit_registry = ToolkitRegistry(toolkit_precedence=toolkits)
>>> toolkit_registry
ToolkitRegistry containing OpenEye Toolkit, The RDKit, AmberTools, Built-in 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
>>> toolkit_registry
ToolkitRegistry containing OpenEye Toolkit, The RDKit, AmberTools, Built-in Toolkit
```

Note that this will contain different ToolkitWrapper objects based on what toolkits are currently installed.

Warning: This API is experimental and subject to change.

Attributes

registered_toolkit_versions Return a dict containing the version of each registered toolkit.

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.
deregister_toolkit(toolkit_wrapper)	Remove a ToolkitWrapper from the list of toolkits in this ToolkitRegistry
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__(*toolkit_precedence=[]*, *exception_if_unavailable=True*, *register_imported_toolkit_wrappers=False*)

Create an empty toolkit registry.

Parameters

toolkit_precedence [list, default=[]] List of toolkit wrapper classes, in order of desired precedence when performing molecule operations. If None, no toolkits will be registered.

exception_if_unavailable [bool, optional, default=True] If True, an exception will be raised if the toolkit is unavailable

_register_imported_toolkit_wrappers [bool, optional, default=False] If True, will attempt to register all imported ToolkitWrapper subclasses that can be found in the order of toolkit_precedence, if specified. If toolkit_precedence is not specified, the default order is [OpenEyeToolkitWrapper, RDKitToolkitWrapper, AmberToolsToolkitWrapper, BuiltInToolkitWrapper].

Methods

`__init__`([toolkit_precedence, ...])	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.
`deregister_toolkit`(toolkit_wrapper)	Remove a ToolkitWrapper from the list of toolkits in this ToolkitRegistry
`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_toolkit_versions`	Return a dict containing the version of each registered toolkit.
`registered_toolkits`	List registered toolkits.

property registered_toolkits
List registered toolkits.

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

Returns

toolkits [iterable of toolkit objects]

property registered_toolkit_versions
Return a dict containing the version of each registered toolkit.

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

Returns

toolkit_versions [dict[str, str]] A dictionary mapping names and versions of wrapped toolkits

`register_toolkit`(*toolkit_wrapper*, *exception_if_unavailable=True*)

Register the provided toolkit wrapper class, instantiating an object of it.

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

Parameters

toolkit_wrapper [instance or subclass of ToolkitWrapper] The toolkit wrapper to register or its class.

exception_if_unavailable [bool, optional, default=True] If True, an exception will be raised if the toolkit is unavailable

`deregister_toolkit`(*toolkit_wrapper*)

Remove a ToolkitWrapper from the list of toolkits in this ToolkitRegistry

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

Parameters

toolkit_wrapper [instance or subclass of ToolkitWrapper] The toolkit wrapper to remove from the registry

Raises

InvalidToolkitError If toolkit_wrapper is not a ToolkitWrapper or subclass

ToolkitUnavailableException If toolkit_wrapper is not found in the registry

`add_toolkit`(*toolkit_wrapper*)

Append a ToolkitWrapper onto the list of toolkits in this ToolkitRegistry

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

Parameters

toolkit_wrapper [openforcefield.utils.ToolkitWrapper] The ToolkitWrapper object to add to the list of registered toolkits

Raises

InvalidToolkitError If toolkit_wrapper is not a ToolkitWrapper or subclass

`resolve`(*method_name*)

Resolve the requested method name by checking all registered toolkits in order of precedence for one that provides the requested method.

Parameters

method_name [str] The name of the method to resolve

Returns

method The method of the first registered toolkit that provides the requested method name

Raises

NotImplementedError if the requested method cannot be found among the registered toolkits

Examples

Create a molecule, and call the toolkit `to_smiles()` method directly

```
>>> from openforcefield.topology import Molecule
>>> molecule = Molecule.from_smiles('Cc1ccccc1')
>>> toolkit_registry = ToolkitRegistry([OpenEyeToolkitWrapper, RDKitToolkitWrapper,␣
↪AmberToolsToolkitWrapper])
>>> method = toolkit_registry.resolve('to_smiles')
>>> smiles = method(molecule)
```

call(*method_name*, **args*, *raise_exception_types=None*, ***kwargs*)

Execute the requested method by attempting to use all registered toolkits in order of precedence.

`*args` and `**kwargs` are passed to the desired method, and return values of the method are returned

This is a convenient shorthand for `toolkit_registry.resolve_method(method_name)(*args, **kwargs)`

Parameters

method_name [str] The name of the method to execute

raise_exception_types [list of Exception subclasses, default=None] A list of exception-derived types to catch and raise immediately. If None, this will be set to [Exception], which will raise an error immediately if the first ToolkitWrapper in the registry fails. To try each ToolkitWrapper that provides a suitably-named method, set this to the empty list ([]). If all ToolkitWrappers run without raising any exceptions in this list, a single ValueError will be raised containing the each ToolkitWrapper that was tried and the exception it raised.

Raises

NotImplementedError if the requested method cannot be found among the registered toolkits

ValueError if no exceptions in the raise_exception_types list were raised by ToolkitWrappers, and

all ToolkitWrappers in the ToolkitRegistry were tried.

Other forms of exceptions are possible if raise_exception_types is specified.

These are defined by the ToolkitWrapper method being called.

Examples

Create a molecule, and call the toolkit `to_smiles()` method directly

```
>>> from openforcefield.topology import Molecule
>>> molecule = Molecule.from_smiles('Cc1ccccc1')
>>> toolkit_registry = ToolkitRegistry([OpenEyeToolkitWrapper, RDKitToolkitWrapper])
>>> smiles = toolkit_registry.call('to_smiles', molecule)
```

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 Return the name of the toolkit wrapped by this class as a str

toolkit_version Return the version of the wrapped toolkit as a str

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`	Return the name of the toolkit wrapped by this class as a str
`toolkit_version`	Return the version of the wrapped toolkit as a str

property toolkit_name
Return the name of the toolkit wrapped by this class as a str

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

Returns

toolkit_name [str] The name of the wrapped toolkit

property toolkit_installation_instructions
classmethod(function) -> method

Convert a function to be a class method.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

class C: @classmethod def f(cls, arg1, arg2, ...):

...

It can be called either on the class (e.g. C.f()) or on an instance (e.g. C().f()). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see the staticmethod builtin.

property toolkit_file_read_formats
List of file formats that this toolkit can read.

property toolkit_file_write_formats
List of file formats that this toolkit can write.

classmethod is_available()
Check whether the corresponding toolkit can be imported

Returns

is_installed [bool] True if corresponding toolkit is installed, False otherwise.

property toolkit_version
Return the version of the wrapped toolkit as a str

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

Returns

toolkit_version [str] The version of the wrapped toolkit

from_file(*file_path*, *file_format*, *allow_undefined_stereo=False*)
Return an openforcefield.topology.Molecule from a file using this toolkit.

Parameters

file_path [str] The file to read the molecule from

file_format [str] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.

allow_undefined_stereo [bool, default=False] If false, raises an exception if any molecules contain undefined stereochemistry.

_cls [class] Molecule constructor

Returns

———-

molecules [Molecule or list of Molecules] a list of Molecule objects is returned.

from_file_obj(*file_obj*, *file_format*, *allow_undefined_stereo=False*, *_cls=None*)

Return an openforcefield.topology.Molecule from a file-like object (an object with a ".read()" method using
toolkit.

Parameters

file_obj [file-like object] The file-like object to read the molecule from

file_format [str] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.

allow_undefined_stereo [bool, default=False] If false, raises an exception if any molecules contain undefined stereochemistry. If false, the function skips loading the molecule.

_cls [class] Molecule constructor

Returns

molecules [Molecule or list of Molecules] a list of Molecule objects is returned.

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 Return the name of the toolkit wrapped by this class as a str

toolkit_version Return the version of the wrapped toolkit as a str

Methods

apply_elf_conformer_selection(molecule[, ...])	Applies the `ELF method<https://docs.eyesopen.com/toolkits/python/quacpactk/molchargetheory.html#elf-conformer-selection>`_ to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from a molecules conformers.
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, _cls])	Construct a Molecule from a InChI representation

continues on next page

Table 134 – continued from previous page

from_iupac(iupac_name[, ...])	Construct a Molecule from an IUPAC name
from_object(obj[, allow_undefined_stereo, _cls])	If given an OEMol (or OEMol-derived object), this function will load it into an openforcefield.topology.molecule
from_openeye(oemol[, ...])	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()	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_iupac(molecule)	Generate IUPAC name from Molecule
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.
apply_elf_conformer_selection(molecule[, ...])	Applies the `ELF method<https://docs.eyesopen.com/toolkits/python/quacpactk/molchargetheory.html#elf-conformer-selection>`_ to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from a molecules conformers.
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.

continues on next page

Table 135 – continued from previous page

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, _cls])	Construct a Molecule from a InChI representation
from_iupac(iupac_name[, ...])	Construct a Molecule from an IUPAC name
from_object(obj[, allow_undefined_stereo, _cls])	If given an OEMol (or OEMol-derived object), this function will load it into an openforcefield.topology.molecule
from_openeye(oemol[, ...])	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()	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_iupac(molecule)	Generate IUPAC name from Molecule
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	Return the name of the toolkit wrapped by this class as a str
toolkit_version	Return the version of the wrapped toolkit as a str

classmethod is_available()

Check if the given OpenEye toolkit components are available.

If the OpenEye toolkit is not installed or no license is found for at least one the required toolkits , `False` is returned.

Returns

all_installed [bool] `True` if all required OpenEye tools are installed and licensed, `False` otherwise

from_object(*obj*, *allow_undefined_stereo=False*, *_cls=None*)

If given an OEMol (or OEMol-derived object), this function will load it into an openforcefield.topology.molecule

Parameters

obj [A molecule-like object] An object to by type-checked.

allow_undefined_stereo [bool, default=False] Whether to accept molecules with undefined stereocenters. If False, an exception will be raised if a molecule with undefined stereochemistry is passed into this function.

_cls [class] Molecule constructor

Returns

——-

Molecule An openforcefield.topology.molecule Molecule.

Raises

NotImplementedError If the object could not be converted into a Molecule.

from_file(*file_path*, *file_format*, *allow_undefined_stereo=False*, *_cls=None*)

Return an openforcefield.topology.Molecule from a file using this toolkit.

Parameters

file_path [str] The file to read the molecule from

file_format [str] Format specifier, usually file suffix (eg. ‘MOL2’, ‘SMI’) Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

_cls [class] Molecule constructor

Returns

molecules [List[Molecule]] The list of `Molecule` objects in the file.

Raises

GAFFAtomTypeWarning If the loaded mol2 file possibly uses GAFF atom types, which are not supported.

Examples

Load a mol2 file into an OpenFF `Molecule` object.

```
>>> from openforcefield.utils import get_data_file_path
>>> mol2_file_path = get_data_file_path('molecules/cyclohexane.mol2')
>>> toolkit = OpenEyeToolkitWrapper()
>>> molecule = toolkit.from_file(mol2_file_path, file_format='mol2')
```

from_file_obj(*file_obj*, *file_format*, *allow_undefined_stereo=False*, *_cls=None*)
Return an openforcefield.topology.Molecule from a file-like object (an object with a ".read()" method using this toolkit.

Parameters

file_obj [file-like object] The file-like object to read the molecule from

file_format [str] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

_cls [class] Molecule constructor

Returns

molecules [List[Molecule]] The list of Molecule objects in the file object.

Raises

GAFFAtomTypeWarning If the loaded mol2 file possibly uses GAFF atom types, which are not supported.

to_file_obj(*molecule*, *file_obj*, *file_format*)
Writes an OpenFF Molecule to a file-like object

Parameters

molecule [an OpenFF Molecule] The molecule to write

file_obj The file-like object to write to

file_format The format for writing the molecule data

to_file(*molecule*, *file_path*, *file_format*)
Writes an OpenFF Molecule to a file-like object

Parameters

molecule [an OpenFF Molecule] The molecule to write

file_path The file path to write to.

file_format The format for writing the molecule data

enumerate_protomers(*molecule*, *max_states=10*)
Enumerate the formal charges of a molecule to generate different protomoers.

Parameters

molecule: openforcefield.topology.Molecule The molecule whose state we should enumerate

max_states: int optional, default=10, The maximum number of protomer states to be returned.

Returns

molecules: List[openforcefield.topology.Molecule], A list of the protomers of the input molecules not including the input.

enumerate_stereoisomers(*molecule*, *undefined_only=False*, *max_isomers=20*, *rationalise=True*)
Enumerate the stereocenters and bonds of the current molecule.

Parameters

molecule: openforcefield.topology.Molecule The molecule whose state we should enumerate

undefined_only: bool optional, default=False If we should enumerate all stereocenters and bonds or only those with undefined stereochemistry

max_isomers: int optional, default=20 The maximum amount of molecules that should be returned

rationalise: bool optional, default=True If we should try to build and rationalise the molecule to ensure it can exist

Returns

molecules: List[openforcefield.topology.Molecule] A list of openforcefield.topology.Molecule instances

enumerate_tautomers(*molecule*, *max_states=20*)
Enumerate the possible tautomers of the current molecule

Parameters

molecule: openforcefield.topology.Molecule The molecule whose state we should enumerate

max_states: int optional, default=20 The maximum amount of molecules that should be returned

Returns

molecules: List[openforcefield.topology.Molecule] A list of openforcefield.topology.Molecule instances excluding the input molecule.

static from_openeye(*oemol*, *allow_undefined_stereo=False*, *_cls=None*)
Create a Molecule from an OpenEye molecule. If the OpenEye molecule has implicit hydrogens, this function will make them explicit.

`OEAtom` s have a different set of allowed value for partial charges than `openforcefield.topology.Molecule` s. In the OpenEye toolkits, partial charges are stored on individual `OEAtom` s, and their values are initialized to `0.0`. In the Open Force Field Toolkit, an `openforcefield.topology.Molecule`'s `partial_charges` attribute is initialized to `None` and can be set to a `simtk.unit.Quantity`-wrapped numpy array with units of elementary charge. The Open Force Field Toolkit considers an `OEMol` where every `OEAtom` has a partial charge of `float('nan')` to be equivalent

to an Open Force Field Molecule's `partial_charges = None`. This assumption is made in both `to_openeye` and `from_openeye`.

Warning: This API is experimental and subject to change.

Parameters

oemol [openeye.oechem.OEMol] An OpenEye molecule

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

_cls [class] Molecule constructor

Returns

molecule [openforcefield.topology.Molecule] An openforcefield molecule

Examples

Create a Molecule from an OpenEye OEMol

```
>>> from openeye import oechem
>>> from openforcefield.tests.utils import get_data_file_path
>>> ifs = oechem.oemolistream(get_data_file_path('systems/monomers/ethanol.mol2'))
>>> oemols = list(ifs.GetOEGraphMols())
```

```
>>> toolkit_wrapper = OpenEyeToolkitWrapper()
>>> molecule = toolkit_wrapper.from_openeye(oemols[0])
```

static to_openeye(*molecule*, *aromaticity_model='OEAroModel_MDL'*)

Create an OpenEye molecule using the specified aromaticity model

`OEAtom` s have a different set of allowed value for partial charges than `openforcefield.topology.Molecule` s. In the OpenEye toolkits, partial charges are stored on individual `OEAtom` s, and their values are initialized to `0.0`. In the Open Force Field Toolkit, an `openforcefield.topology.Molecule`'s `partial_charges` attribute is initialized to `None` and can be set to a `simtk.unit.Quantity`-wrapped numpy array with units of elementary charge. The Open Force Field Toolkit considers an `OEMol` where every `OEAtom` has a partial charge of `float('nan')` to be equivalent to an Open Force Field Molecule's `partial_charges = None`. This assumption is made in both `to_openeye` and `from_openeye`.

Warning: This API is experimental and subject to change.

Parameters

molecule [openforcefield.topology.molecule.Molecule object]

The molecule to convert to an OEMol

aromaticity_model [str, optional, default=DEFAULT_AROMATICITY_MODEL] The aromaticity model to use

Returns

oemol [openeye.oechem.OEMol] An OpenEye molecule

Examples

Create an OpenEye molecule from a Molecule

```
>>> from openforcefield.topology import Molecule
>>> toolkit_wrapper = OpenEyeToolkitWrapper()
>>> molecule = Molecule.from_smiles('CC')
>>> oemol = toolkit_wrapper.to_openeye(molecule)
```

to_smiles(*molecule*, *isomeric=True*, *explicit_hydrogens=True*, *mapped=False*)

Uses the OpenEye toolkit to convert a Molecule into a SMILES string. A partially mapped smiles can also be generated for atoms of interest by supplying an *atom_map* to the properties dictionary.

Parameters

molecule [An openforcefield.topology.Molecule] The molecule to convert into a SMILES.

isomeric: bool optional, default= True return an isomeric smiles

explicit_hydrogens: bool optional, default=True return a smiles string containing all hydrogens explicitly

mapped: bool optional, default=False return a explicit hydrogen mapped smiles, the atoms to be mapped can be controlled by supplying an atom map into the properties dictionary. If no mapping is passed all atoms will be mapped in order, else an atom map dictionary from the current atom index to the map id should be supplied with no duplicates. The map ids (values) should start from 0 or 1.

Returns

smiles [str] The SMILES of the input molecule.

to_inchi(*molecule*, *fixed_hydrogens=False*)

Create an InChI string for the molecule using the RDKit Toolkit. InChI is a standardised representation that does not capture tautomers unless specified using the fixed hydrogen layer.

For information on InChi see here https://iupac.org/who-we-are/divisions/division-details/inchi/

Parameters

molecule [An openforcefield.topology.Molecule] The molecule to convert into a SMILES.

fixed_hydrogens: bool, default=False If a fixed hydrogen layer should be added to the InChI, if *True* this will produce a non standard specific InChI string of the molecule.

Returns

inchi: str The InChI string of the molecule.

to_inchikey(*molecule*, *fixed_hydrogens=False*)

Create an InChIKey for the molecule using the RDKit Toolkit. InChIKey is a standardised representation that does not capture tautomers unless specified using the fixed hydrogen layer.

For information on InChi see here https://iupac.org/who-we-are/divisions/division-details/inchi/

Parameters

molecule [An openforcefield.topology.Molecule] The molecule to convert into a SMILES.

fixed_hydrogens: bool, default=False If a fixed hydrogen layer should be added to the InChI, if *True* this will produce a non standard specific InChI string of the molecule.

Returns

inchi_key: str The InChIKey representation of the molecule.

to_iupac(*molecule*)

Generate IUPAC name from Molecule

Parameters

molecule [An openforcefield.topology.Molecule] The molecule to convert into a SMILES.

Returns

iupac_name [str] IUPAC name of the molecule

Examples

```
>>> from openforcefield.topology import Molecule
>>> from openforcefield.utils import get_data_file_path
>>> sdf_filepath = get_data_file_path('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
>>> toolkit = OpenEyeToolkitWrapper()
>>> iupac_name = toolkit.to_iupac(molecule)
```

canonical_order_atoms(*molecule*)

Canonical order the atoms in the molecule using the OpenEye toolkit.

Parameters

molecule: openforcefield.topology.Molecule The input molecule

Returns

molecule [openforcefield.topology.Molecule] The input molecule, with canonically-indexed atoms and bonds.

from_smiles(*smiles*, *hydrogens_are_explicit=False*, *allow_undefined_stereo=False*, *_cls=None*)

Create a Molecule from a SMILES string using the OpenEye toolkit.

Warning: This API is experimental and subject to change.

Parameters

smiles [str] The SMILES string to turn into a molecule

hydrogens_are_explicit [bool, default = False] If False, OE will perform hydrogen addition using OEAddExplicitHydrogens

allow_undefined_stereo [bool, default=False] Whether to accept SMILES with undefined stereochemistry. If False, an exception will be raised if a SMILES with undefined stereochemistry is passed into this function.

_cls [class] Molecule constructor

Returns

———-

molecule [openforcefield.topology.Molecule] An openforcefield-style molecule.

from_inchi(*inchi*, *allow_undefined_stereo=False*, *_cls=None*)
Construct a Molecule from a InChI representation

Parameters

inchi [str] The InChI representation of the molecule.

allow_undefined_stereo [bool, default=False] Whether to accept InChI with undefined stereochemistry. If False, an exception will be raised if a InChI with undefined stereochemistry is passed into this function.

_cls [class] Molecule constructor

Returns

molecule [openforcefield.topology.Molecule]

from_iupac(*iupac_name*, *allow_undefined_stereo=False*, *_cls=None*, ***kwargs*)
Construct a Molecule from an IUPAC name

Parameters

iupac_name [str] The IUPAC or common name of the molecule.

allow_undefined_stereo [bool, default=False] Whether to accept a molecule name with undefined stereochemistry. If False, an exception will be raised if a molecule name with undefined stereochemistry is passed into this function.

_cls [class] Molecule constructor

Returns

molecule [openforcefield.topology.Molecule]

generate_conformers(*molecule*, *n_conformers=1*, *rms_cutoff=None*, *clear_existing=True*)
Generate molecule conformers using OpenEye Omega.

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

molecule [a `Molecule`] The molecule to generate conformers for.

n_conformers [int, default=1] The maximum number of conformers to generate.

rms_cutoff [simtk.Quantity-wrapped float, in units of distance, optional, default=None] The minimum RMS value at which two conformers are considered redundant and one is deleted. If None, the cutoff is set to 1 Angstrom

clear_existing [bool, default=True] Whether to overwrite existing conformers for the molecule

apply_elf_conformer_selection(*molecule: Molecule*, *percentage: float = 2.0*, *limit: int = 10*)

Applies the `ELF method<https://docs.eyesopen.com/toolkits/python/quacpactk/molchargetheory.html#elf-conformer-selection>`_ to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from a molecules conformers.

Parameters

molecule The molecule which contains the set of conformers to select from.

percentage The percentage of conformers with the lowest electrostatic interaction energies to greedily select from.

limit The maximum number of conformers to select.

See also:

`RDKitToolkitWrapper.apply_elf_conformer_selection`

Notes

- The input molecule should have a large set of conformers already generated to select the ELF conformers from.
- The selected conformers will be retained in the *molecule.conformers* list while unselected conformers will be discarded.

assign_partial_charges(*molecule*, *partial_charge_method=None*, *use_conformers=None*, *strict_n_conformers=False*, *_cls=None*)

Compute partial charges with OpenEye quacpac, and assign the new values to the partial_charges attribute.

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

Parameters

molecule [openforcefield.topology.Molecule] Molecule for which partial charges are to be computed

partial_charge_method [str, optional, default=None] The charge model to use. One of ['amberff94', 'mmff', 'mmff94', *am1-mulliken*, 'am1bcc', 'am1bccnosymspt', 'am1bccelf10'] If None, 'am1-mulliken' will be used.

use_conformers [iterable of simtk.unit.Quantity-wrapped numpy arrays, each with shape (n_atoms, 3) and dimension of distance. Optional, default = None] Coordinates to use for partial charge calculation. If None, an appropriate number of conformers will be generated.

strict_n_conformers [bool, default=False] Whether to raise an exception if an invalid number of conformers is provided for the given charge method. If this is False and an invalid number of conformers is found, a warning will be raised.

_cls [class] Molecule constructor

Raises

ChargeMethodUnavailableError if the requested charge method can not be handled by this toolkit

ChargeCalculationError if the charge method is supported by this toolkit, but fails

compute_partial_charges_am1bcc(*molecule*, *use_conformers=None*, *strict_n_conformers=False*)
Compute AM1BCC partial charges with OpenEye quacpac. This function will attempt to use the OEAM1BCCELF10 charge generation method, but may print a warning and fall back to normal OEAM1BCC if an error is encountered. This error is known to occur with some carboxylic acids, and is under investigation by OpenEye.

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

Parameters

molecule [Molecule] Molecule for which partial charges are to be computed

use_conformers [iterable of simtk.unit.Quantity-wrapped numpy arrays, each with shape (n_atoms, 3) and dimension of distance. Optional, default = None] Coordinates to use for partial charge calculation. If None, an appropriate number of conformers will be generated.

strict_n_conformers [bool, default=False] Whether to raise an exception if an invalid number of conformers is provided. If this is False and an invalid number of conformers is found, a warning will be raised instead of an Exception.

Returns

charges [numpy.array of shape (natoms) of type float] The partial charges

assign_fractional_bond_orders(*molecule*, *bond_order_model=None*, *use_conformers=None*, *_cls=None*)
Update and store list of bond orders this molecule. Bond orders are stored on each bond, in the *bond.fractional_bond_order* attribute.

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

Parameters

molecule [openforcefield.topology.molecule Molecule] The molecule to assign wiberg bond orders to

bond_order_model [str, optional, default=None] The charge model to use. One of ['am1-wiberg', 'am1-wiberg-elf10', 'pm3-wiberg', 'pm3-wiberg-elf10']. If None, 'am1-wiberg' will be used.

use_conformers [iterable of simtk.unit.Quantity(np.array) with shape (n_atoms, 3) and dimension of distance, optional, default=None] The conformers to use for fractional bond order calculation. If None, an appropriate number of conformers will be generated by an available ToolkitWrapper. If the chosen `bond_order_model` is an ELF variant, the ELF conformer selection method will be applied to the provided conformers.

_cls [class] Molecule constructor

get_tagged_smarts_connectivity(*smarts*)
Returns a tuple of tuples indicating connectivity between tagged atoms in a SMARTS string. Does not return bond order.

Parameters

smarts [str] The tagged SMARTS to analyze

Returns

unique_tags [tuple of int] A sorted tuple of all unique tagged atom map indices.

tagged_atom_connectivity [tuple of tuples of int, shape n_tagged_bonds x 2] A tuple of tuples, where each inner tuple is a pair of tagged atoms (tag_idx_1, tag_idx_2) which are bonded. The inner tuples are ordered smallest-to-largest, and the tuple of tuples is ordered lexically. So the return value for an improper torsion would be ((1, 2), (2, 3), (2, 4)).

Raises

SMIRKSParsingError If OpenEye toolkit was unable to parse the provided smirks/tagged smarts

find_smarts_matches(*molecule*, *smarts*, *aromaticity_model='OEAroModel_MDL'*)
Find all SMARTS matches for the specified molecule, using the specified aromaticity model.

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

Parameters

molecule [openforcefield.topology.Molecule] The molecule for which all specified SMARTS matches are to be located

smarts [str] SMARTS string with optional SMIRKS-style atom tagging

aromaticity_model [str, optional, default='OEAroModel_MDL'] Molecule is prepared with this aromaticity model prior to querying.

.. note :: Currently, the only supported ``aromaticity_model`` is ``OEAroModel_MDL``

property toolkit_file_read_formats
List of file formats that this toolkit can read.

property toolkit_file_write_formats
List of file formats that this toolkit can write.

property toolkit_installation_instructions
classmethod(function) -> method

Convert a function to be a class method.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

class C: @classmethod def f(cls, arg1, arg2, ...):

...

It can be called either on the class (e.g. C.f()) or on an instance (e.g. C().f()). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see the staticmethod builtin.

property toolkit_name
Return the name of the toolkit wrapped by this class as a str

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

Returns

toolkit_name [str] The name of the wrapped toolkit

property toolkit_version
Return the version of the wrapped toolkit as a str

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

Returns

toolkit_version [str] The version of the wrapped toolkit

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	Return the name of the toolkit wrapped by this class as a str
toolkit_version	Return the version of the wrapped toolkit as a str

Methods

apply_elf_conformer_selection(molecule[, ...])	Applies the `ELF method<https://docs.eyesopen.com/toolkits/python/quacpactk/molchargetheory.html#elf-conformer-selection>`_ to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from a molecules conformers.

continues on next page

Table 137 – continued from previous page

assign_partial_charges(molecule[, ...])	Compute partial charges with RDKit, and assign the new values to the partial_charges attribute.
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_rings(molecule)	Find the rings in a given 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, _cls])	Construct a Molecule from a InChI representation
from_object(obj[, allow_undefined_stereo, _cls])	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, _cls])	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.
apply_elf_conformer_selection(molecule[, ...])	Applies the `ELF method<https://docs.eyesopen.com/toolkits/python/quacpactk/molchargetheory.html#elf-conformer-selection>`_ to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from a molecules conformers.
assign_partial_charges(molecule[, ...])	Compute partial charges with RDKit, and assign the new values to the partial_charges attribute.
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_rings(molecule)	Find the rings in a given 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, _cls])	Construct a Molecule from a InChI representation
from_object(obj[, allow_undefined_stereo, _cls])	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, _cls])	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

continues on next page

Table 138 – continued from previous page

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	Return the name of the toolkit wrapped by this class as a str
toolkit_version	Return the version of the wrapped toolkit as a str

property toolkit_file_write_formats
List of file formats that this toolkit can write.

classmethod is_available()
Check whether the RDKit toolkit can be imported

Returns

is_installed [bool] True if RDKit is installed, False otherwise.

from_object(*obj*, *allow_undefined_stereo=False*, *_cls=None*)
If given an rdchem.Mol (or rdchem.Mol-derived object), this function will load it into an openforcefield.topology.molecule. Otherwise, it will return False.

Parameters

obj [A rdchem.Mol-derived object] An object to be type-checked and converted into a Molecule, if possible.

allow_undefined_stereo [bool, default=False] Whether to accept molecules with undefined stereocenters. If False, an exception will be raised if a molecule with undefined stereochemistry is passed into this function.

_cls [class] Molecule constructor

Returns

Molecule or False An openforcefield.topology.molecule Molecule.

Raises

NotImplementedError If the object could not be converted into a Molecule.

from_pdb_and_smiles(*file_path*, *smiles*, *allow_undefined_stereo=False*, *_cls=None*)
Create a Molecule from a pdb file and a SMILES string using RDKit.

Requires RDKit to be installed.

The molecule is created and sanitised based on the SMILES string, we then find a mapping between this molecule and one from the PDB based only on atomic number and connections. The SMILES molecule is then reindex to match the PDB, the conformer is attached and the molecule returned.

Parameters

file_path: str PDB file path

smiles [str] a valid smiles string for the pdb, used for seterochemistry and bond order

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

_cls [class] Molecule constructor

Returns

molecule [openforcefield.Molecule (or _cls() type)] An OFFMol instance with ordering the same as used in the PDB file.

Raises

InvalidConformerError [if the SMILES and PDB molecules are not isomorphic.]

from_file(*file_path*, *file_format*, *allow_undefined_stereo=False*, *_cls=None*)
Create an openforcefield.topology.Molecule from a file using this toolkit.

Parameters

file_path [str] The file to read the molecule from

file_format [str] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

_cls [class] Molecule constructor

Returns

———-

molecules [iterable of Molecules] a list of Molecule objects is returned.

from_file_obj(*file_obj*, *file_format*, *allow_undefined_stereo=False*, *_cls=None*)
Return an openforcefield.topology.Molecule from a file-like object (an object with a ".read()" method using this toolkit.

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

Parameters

file_obj [file-like object] The file-like object to read the molecule from

file_format [str] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.

allow_undefined_stereo [bool, default=False] If false, raises an exception if oemol contains undefined stereochemistry.

_cls [class] Molecule constructor

Returns

———-

molecules [Molecule or list of Molecules] a list of Molecule objects is returned.

to_file_obj(*molecule*, *file_obj*, *file_format*)
Writes an OpenFF Molecule to a file-like object

Parameters

molecule [an OpenFF Molecule] The molecule to write

file_obj The file-like object to write to

file_format The format for writing the molecule data

to_file(*molecule*, *file_path*, *file_format*)
Writes an OpenFF Molecule to a file-like object

Parameters

molecule [an OpenFF Molecule] The molecule to write

file_path The file path to write to

file_format The format for writing the molecule data

Returns

——

enumerate_stereoisomers(*molecule*, *undefined_only=False*, *max_isomers=20*, *rationalise=True*)
Enumerate the stereocenters and bonds of the current molecule.

Parameters

molecule: openforcefield.topology.Molecule The molecule whose state we should enumerate

undefined_only: bool optional, default=False If we should enumerate all stereocenters and bonds or only those with undefined stereochemistry

max_isomers: int optional, default=20 The maximum amount of molecules that should be returned

rationalise: bool optional, default=True If we should try to build and rationalise the molecule to ensure it can exist

Returns

molecules: List[openforcefield.topology.Molecule] A list of openforcefield.topology.Molecule instances

enumerate_tautomers(*molecule*, *max_states=20*)
Enumerate the possible tautomers of the current molecule.

Parameters

molecule: openforcefield.topology.Molecule The molecule whose state we should enumerate

max_states: int optional, default=20 The maximum amount of molecules that should be returned

Returns

molecules: List[openforcefield.topology.Molecule] A list of openforcefield.topology.Molecule instances not including the input molecule.

canonical_order_atoms(*molecule*)
Canonical order the atoms in the molecule using the RDKit.

Parameters

molecule: openforcefield.topology.Molecule The input molecule

Returns

molecule [openforcefield.topology.Molecule] The input molecule, with canonically-indexed atoms and bonds.

`to_smiles`(*molecule*, *isomeric=True*, *explicit_hydrogens=True*, *mapped=False*)

Uses the RDKit toolkit to convert a Molecule into a SMILES string. A partially mapped smiles can also be generated for atoms of interest by supplying an *atom_map* to the properties dictionary.

Parameters

molecule [An openforcefield.topology.Molecule] The molecule to convert into a SMILES.

isomeric: bool optional, default= True return an isomeric smiles

explicit_hydrogens: bool optional, default=True return a smiles string containing all hydrogens explicitly

mapped: bool optional, default=False return a explicit hydrogen mapped smiles, the atoms to be mapped can be controlled by supplying an atom map into the properties dictionary. If no mapping is passed all atoms will be mapped in order, else an atom map dictionary from the current atom index to the map id should be supplied with no duplicates. The map ids (values) should start from 0 or 1.

Returns

smiles [str] The SMILES of the input molecule.

`from_smiles`(*smiles*, *hydrogens_are_explicit=False*, *allow_undefined_stereo=False*, *_cls=None*)

Create a Molecule from a SMILES string using the RDKit toolkit.

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

Parameters

smiles [str] The SMILES string to turn into a molecule

hydrogens_are_explicit [bool, default=False] If False, RDKit will perform hydrogen addition using Chem.AddHs

allow_undefined_stereo [bool, default=False] Whether to accept SMILES with undefined stereochemistry. If False, an exception will be raised if a SMILES with undefined stereochemistry is passed into this function.

_cls [class] Molecule constructor

Returns

molecule [openforcefield.topology.Molecule] An openforcefield-style molecule.

`from_inchi`(*inchi*, *allow_undefined_stereo=False*, *_cls=None*)

Construct a Molecule from a InChI representation

Parameters

inchi [str] The InChI representation of the molecule.

allow_undefined_stereo [bool, default=False] Whether to accept InChI with undefined stereochemistry. If False, an exception will be raised if a InChI with undefined stereochemistry is passed into this function.

_cls [class] Molecule constructor

Returns

molecule [openforcefield.topology.Molecule]

generate_conformers(*molecule*, *n_conformers=1*, *rms_cutoff=None*, *clear_existing=True*, *_cls=None*)

Generate molecule conformers using RDKit.

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

molecule [a `Molecule`] The molecule to generate conformers for.

n_conformers [int, default=1] Maximum number of conformers to generate.

rms_cutoff [simtk.Quantity-wrapped float, in units of distance, optional, default=None] The minimum RMS value at which two conformers are considered redundant and one is deleted. If None, the cutoff is set to 1 Angstrom

clear_existing [bool, default=True] Whether to overwrite existing conformers for the molecule.

_cls [class] Molecule constructor

assign_partial_charges(*molecule*, *partial_charge_method=None*, *use_conformers=None*, *strict_n_conformers=False*, *_cls=None*)

Compute partial charges with RDKit, and assign the new values to the partial_charges attribute.

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

Parameters

molecule [openforcefield.topology.Molecule] Molecule for which partial charges are to be computed

partial_charge_method [str, optional, default=None] The charge model to use. One of ['mmff94']. If None, 'mmff94' will be used.

- **'mmff94': Applies partial charges using the Merck Molecular Force Field** (MMFF). This method does not make use of conformers, and hence `use_conformers` and `strict_n_conformers` will not impact the partial charges produced.

use_conformers [iterable of simtk.unit.Quantity-wrapped numpy arrays, each with shape (n_atoms, 3) and dimension of distance. Optional, default = None] Coordinates to use for partial charge calculation. If None, an appropriate number of conformers will be generated.

strict_n_conformers [bool, default=False] Whether to raise an exception if an invalid number of conformers is provided for the given charge method. If this is False and an invalid number of conformers is found, a warning will be raised.

_cls [class] Molecule constructor

Raises

ChargeMethodUnavailableError if the requested charge method can not be handled by this toolkit

ChargeCalculationError if the charge method is supported by this toolkit, but fails

apply_elf_conformer_selection(*molecule: Molecule, percentage: float = 2.0, limit: int = 10, rms_tolerance: simtk.unit.quantity.Quantity = Quantity(value=0.05, unit=angstrom)*)

Applies the **`ELF method<https://docs.eyesopen.com/toolkits/python/quacpactk/molchargetheory.html#elf-conformer-selection>`_** to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from a molecules conformers.

The diverse conformer selection is performed by the `_elf_select_diverse_conformers` function, which attempts to greedily select conformers which are most distinct according to their RMS.

Parameters

molecule The molecule which contains the set of conformers to select from.

percentage The percentage of conformers with the lowest electrostatic interaction energies to greedily select from.

limit The maximum number of conformers to select.

rms_tolerance Conformers whose RMS is within this amount will be treated as identical and the duplicate discarded.

Warning:

- Although this function is inspired by the OpenEye ELF10 method, this implementation may yield slightly different conformers due to potential differences in this and the OE closed source implementation.

See also:

RDKitToolkitWrapper._elf_select_diverse_conformers

Notes

- The input molecule should have a large set of conformers already generated to select the ELF10 conformers from.
- The selected conformers will be retained in the *molecule.conformers* list while unselected conformers will be discarded.
- Only heavy atoms are included when using the RMS to select diverse conformers.

from_rdkit(*rdmol, allow_undefined_stereo=False, _cls=None*)

Create a Molecule from an RDKit molecule.

Requires the RDKit to be installed.

Warning: This API is experimental and subject to change.

Parameters

rdmol [rkit.RDMol] An RDKit molecule

allow_undefined_stereo [bool, default=False] If false, raises an exception if rdmol contains undefined stereochemistry.

_cls [class] Molecule constructor

Returns

molecule [openforcefield.Molecule] An openforcefield molecule

Examples

Create a molecule from an RDKit molecule

```
>>> from rdkit import Chem
>>> from openforcefield.tests.utils import get_data_file_path
>>> rdmol = Chem.MolFromMolFile(get_data_file_path('systems/monomers/ethanol.sdf'))
```

```
>>> toolkit_wrapper = RDKitToolkitWrapper()
>>> molecule = toolkit_wrapper.from_rdkit(rdmol)
```

classmethod to_rdkit(*molecule*, *aromaticity_model='OEAroModel_MDL'*)

Create an RDKit molecule

Requires the RDKit to be installed.

Warning: This API is experimental and subject to change.

Parameters

aromaticity_model [str, optional, default=DEFAULT_AROMATICITY_MODEL] The aromaticity model to use

Returns

rdmol [rkit.RDMol] An RDKit molecule

Examples

Convert a molecule to RDKit

```
>>> from openforcefield.topology import Molecule
>>> ethanol = Molecule.from_smiles('CCO')
>>> rdmol = ethanol.to_rdkit()
```

to_inchi(*molecule*, *fixed_hydrogens=False*)

Create an InChI string for the molecule using the RDKit Toolkit. InChI is a standardised representation that does not capture tautomers unless specified using the fixed hydrogen layer.

For information on InChi see here https://iupac.org/who-we-are/divisions/division-details/inchi/

Parameters

molecule [An openforcefield.topology.Molecule] The molecule to convert into a SMILES.

fixed_hydrogens: bool, default=False If a fixed hydrogen layer should be added to the InChI, if *True* this will produce a non standard specific InChI string of the molecule.

Returns

inchi: str The InChI string of the molecule.

`to_inchikey`(*molecule*, *fixed_hydrogens=False*)

Create an InChIKey for the molecule using the RDKit Toolkit. InChIKey is a standardised representation that does not capture tautomers unless specified using the fixed hydrogen layer.

For information on InChi see here https://iupac.org/who-we-are/divisions/division-details/inchi/

Parameters

molecule [An openforcefield.topology.Molecule] The molecule to convert into a SMILES.

fixed_hydrogens: bool, default=False If a fixed hydrogen layer should be added to the InChI, if *True* this will produce a non standard specific InChI string of the molecule.

Returns

inchi_key: str The InChIKey representation of the molecule.

`get_tagged_smarts_connectivity`(*smarts*)

Returns a tuple of tuples indicating connectivity between tagged atoms in a SMARTS string. Does not return bond order.

Parameters

smarts [str] The tagged SMARTS to analyze

Returns

unique_tags [tuple of int] A sorted tuple of all unique tagged atom map indices.

tagged_atom_connectivity [tuple of tuples of int, shape n_tagged_bonds x 2] A tuple of tuples, where each inner tuple is a pair of tagged atoms (tag_idx_1, tag_idx_2) which are bonded. The inner tuples are ordered smallest-to-largest, and the tuple of tuples is ordered lexically. So the return value for an improper torsion would be ((1, 2), (2, 3), (2, 4)).

Raises

SMIRKSParsingError If RDKit was unable to parse the provided smirks/tagged smarts

`find_smarts_matches`(*molecule*, *smarts*, *aromaticity_model='OEAroModel_MDL'*)

Find all SMARTS matches for the specified molecule, using the specified aromaticity model.

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

Parameters

molecule [openforcefield.topology.Molecule] The molecule for which all specified SMARTS matches are to be located

smarts [str] SMARTS string with optional SMIRKS-style atom tagging

aromaticity_model [str, optional, default='OEAroModel_MDL'] Molecule is prepared with this aromaticity model prior to querying.

.. note :: Currently, the only supported ``aromaticity_model`` is ``OEAroModel_MDL``

find_rings(*molecule*)

Find the rings in a given molecule.

Note: For systems containing some special cases of connected rings, this function may not be well-behaved and may report a different number rings than expected. Some problematic cases include networks of many (5+) rings or bicyclic moieties (i.e. norbornane).

Parameters

molecule [openforcefield.topology.Molecule] The molecule for which rings are to be found

Returns

rings [tuple of tuples of atom indices] Nested tuples, each containing the indices of atoms in each ring

property toolkit_file_read_formats

List of file formats that this toolkit can read.

property toolkit_installation_instructions

classmethod(function) -> method

Convert a function to be a class method.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

class C: @classmethod def f(cls, arg1, arg2, ...):

...

It can be called either on the class (e.g. C.f()) or on an instance (e.g. C().f()). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see the staticmethod builtin.

property toolkit_name

Return the name of the toolkit wrapped by this class as a str

Warning: This API is experimental and subject to change.

Returns

toolkit_name [str] The name of the wrapped toolkit

property toolkit_version
Return the version of the wrapped toolkit as a str

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

Returns

toolkit_version [str] The version of the wrapped toolkit

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 Return the name of the toolkit wrapped by this class as a str

toolkit_version Return the version of the wrapped toolkit as a str

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`	Return the name of the toolkit wrapped by this class as a str
`toolkit_version`	Return the version of the wrapped toolkit as a str

static is_available()

Check whether the AmberTools toolkit is installed

Returns

is_installed [bool] True if AmberTools is installed, False otherwise.

assign_partial_charges(*molecule*, *partial_charge_method=None*, *use_conformers=None*, *strict_n_conformers=False*, *_cls=None*)

Compute partial charges with AmberTools using antechamber/sqm, and assign the new values to the partial_charges attribute.

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

Parameters

molecule [openforcefield.topology.Molecule] Molecule for which partial charges are to be computed

partial_charge_method [str, optional, default=None] The charge model to use. One of ['gasteiger', 'am1bcc', 'am1-mulliken']. If None, 'am1-mulliken' will be used.

use_conformers [iterable of simtk.unit.Quantity-wrapped numpy arrays, each with shape (n_atoms, 3) and dimension of distance. Optional, default = None] List of (n_atoms x 3) simtk.unit.Quantities to use for partial charge calculation. If None, an appropriate number of conformers will be generated.

strict_n_conformers [bool, default=False] Whether to raise an exception if an invalid number of conformers is provided for the given charge method. If this is False and an invalid number of conformers is found, a warning will be raised.

_cls [class] Molecule constructor

Raises

ChargeMethodUnavailableError if the requested charge method can not be handled by this toolki

ChargeCalculationError if the charge method is supported by this toolkit, but fails

compute_partial_charges_am1bcc(*molecule*, *use_conformers=None*, *strict_n_conformers=False*)

Compute partial charges with AmberTools using antechamber/sqm. This will calculate AM1-BCC charges on the first conformer only.

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

Parameters

molecule [Molecule] Molecule for which partial charges are to be computed

use_conformers [iterable of simtk.unit.Quantity-wrapped numpy arrays, each with shape (n_atoms, 3) and dimension of distance. Optional, default = None] Coordinates to use for partial charge calculation. If None, an appropriate number of conformers will be generated.

strict_n_conformers [bool, default=False] Whether to raise an exception if an invalid number of conformers is provided. If this is False and an invalid number of conformers is found, a warning will be raised instead of an Exception.

Returns

charges [numpy.array of shape (natoms) of type float] The partial charges

assign_fractional_bond_orders(*molecule*, *bond_order_model=None*, *use_conformers=None*, *_cls=None*)

Update and store list of bond orders this molecule. Bond orders are stored on each bond, in the *bond.fractional_bond_order* attribute.

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

Parameters

molecule [openforcefield.topology.molecule Molecule] The molecule to assign wiberg bond orders to

bond_order_model [str, optional, default=None] The charge model to use. Only allowed value is 'am1-wiberg'. If None, 'am1-wiberg' will be used.

use_conformers [iterable of simtk.unit.Quantity(np.array) with shape (n_atoms, 3) and dimension of distance, optional, default=None] The conformers to use for fractional bond order calculation. If None, an appropriate number of conformers will be generated by an available ToolkitWrapper.

_cls [class] Molecule constructor

from_file(*file_path*, *file_format*, *allow_undefined_stereo=False*)

Return an openforcefield.topology.Molecule from a file using this toolkit.

Parameters

file_path [str] The file to read the molecule from

file_format [str] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.

allow_undefined_stereo [bool, default=False] If false, raises an exception if any molecules contain undefined stereochemistry.

_cls [class] Molecule constructor

Returns

———-

molecules [Molecule or list of Molecules] a list of Molecule objects is returned.

from_file_obj(*file_obj*, *file_format*, *allow_undefined_stereo=False*, *_cls=None*)

Return an openforcefield.topology.Molecule from a file-like object (an object with a ".read()" method usin

toolkit.

Parameters

file_obj [file-like object] The file-like object to read the molecule from

file_format [str] Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.

allow_undefined_stereo [bool, default=False] If false, raises an exception if any molecules contain undefined stereochemistry. If false, the function skips loading the molecule.

_cls [class] Molecule constructor

Returns

molecules [Molecule or list of Molecules] a list of Molecule objects is returned.

property toolkit_file_read_formats

List of file formats that this toolkit can read.

property toolkit_file_write_formats

List of file formats that this toolkit can write.

property toolkit_installation_instructions

classmethod(function) -> method

Convert a function to be a class method.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

class C: @classmethod def f(cls, arg1, arg2, ...):

...

It can be called either on the class (e.g. C.f()) or on an instance (e.g. C().f()). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see the staticmethod builtin.

property toolkit_name

Return the name of the toolkit wrapped by this class as a str

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

Returns

toolkit_name [str] The name of the wrapped toolkit

property toolkit_version

Return the version of the wrapped toolkit as a str

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

Returns

toolkit_version [str] The version of the wrapped toolkit

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.

continues on next page

Table 145 – 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()	
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.

to_json(*indent=None*)
Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters

indent [int, optional, default=None] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [str] A JSON serialized representation of the object

classmethod from_json(*serialized*)
Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters

serialized [str] A JSON serialized representation of the object

Returns

instance [cls] An instantiated object

to_bson()
Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns

serialized [bytes] A BSON serialized representation of the objecft

classmethod from_bson(*serialized*)
Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters

serialized [bytes] A BSON serialized representation of the object

Returns

instance [cls] An instantiated object

to_toml()
Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns

serialized [str] A TOML serialized representation of the object

classmethod from_toml(*serialized*)
Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters

serlialized [str] A TOML serialized representation of the object

Returns

instance [cls] An instantiated object

to_yaml()
Return a YAML serialized representation.

Specification: http://yaml.org/

Returns

serialized [str] A YAML serialized representation of the object

classmethod from_yaml(*serialized*)
Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters

serialized [str] A YAML serialized representation of the object

Returns

instance [cls] Instantiated object

to_messagepack()
Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation of the object

classmethod from_messagepack(*serialized*)
Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters

serialized [bytes] A MessagePack-encoded bytes serialized representation

Returns

instance [cls] Instantiated object.

to_xml(*indent=2*)
Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters

indent [int, optional, default=2] If not None, will pretty-print with specified number of spaces for indentation

Returns

serialized [bytes] A MessagePack-encoded bytes serialized representation.

classmethod from_xml(*serialized*)
Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters

serialized [bytes] An XML serialized representation

Returns

instance [cls] Instantiated object.

to_pickle()
Return a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns

serialized [str] A pickled representation of the object

classmethod from_pickle(*serialized*)
Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters

serialized [str] A pickled representation of the object

Returns

instance [cls] An instantiated object

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. Parameters ———- dir_path : str The directory path to enter within the context Examples —— >>> dir_path = '/tmp' >>> with temporary_cd(dir_path): ... pass # do something in dir_path.
get_data_file_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).
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.
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.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

openforcefield.utils.utils.get_molecule_parameterIDs

openforcefield.utils.utils.**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

__init__() (*openforcefield.topology.Atom method*), 160
__init__() (*openforcefield.topology.Bond method*), 166
__init__() (*openforcefield.topology.BondChargeVirtualSite method*), 189
__init__() (*openforcefield.topology.DivalentLonePairVirtualSite method*), 203
__init__() (*openforcefield.topology.FrozenMolecule method*), 69
__init__() (*openforcefield.topology.ImproperDict method*), 235
__init__() (*openforcefield.topology.Molecule method*), 100
__init__() (*openforcefield.topology.MonovalentLonePairVirtualSite method*), 196
__init__() (*openforcefield.topology.Particle method*), 155
__init__() (*openforcefield.topology.Topology method*), 134
__init__() (*openforcefield.topology.TopologyAtom method*), 216
__init__() (*openforcefield.topology.TopologyBond method*), 222
__init__() (*openforcefield.topology.TopologyMolecule method*), 149
__init__() (*openforcefield.topology.TopologyVirtualParticle method*), 183
__init__() (*openforcefield.topology.TopologyVirtualSite method*), 227
__init__() (*openforcefield.topology.TrivalentLonePairVirtualSite method*), 210
__init__() (*openforcefield.topology.ValenceDict method*), 233
__init__() (*openforcefield.topology.VirtualParticle method*), 177
__init__() (*openforcefield.topology.VirtualSite method*), 172
__init__() (*openforcefield.typing.chemistry.ChemicalEnvironment method*), 237
__init__() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField method*), 242
__init__() (*openforcefield.typing.engines.smirnoff.io.ParameterIOHandler method*), 454
__init__() (*openforcefield.typing.engines.smirnoff.io.XMLParameterIOHandler method*), 455
__init__() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler method*), 291
__init__() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler method*), 271
__init__() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler method*), 397
__init__() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler method*), 439
__init__() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHand method*), 335
__init__() (*openforcefield.typing.engines.smirnoff.parameters.IndexedMappedParam method*), 464
__init__() (*openforcefield.typing.engines.smirnoff.parameters.IndexedParameterAttr method*), 461
__init__() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandle method*), 405
__init__() (*openforcefield.typing.engines.smirnoff.parameters.MappedParameterAtt method*), 462

__init__() (*openforcefield.typing.engines.smirnoff.parameters.ParameterAttribute method*), 459
__init__() (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler method*), 259
__init__() (*openforcefield.typing.engines.smirnoff.parameters.ParameterList method*), 255
__init__() (*openforcefield.typing.engines.smirnoff.parameters.ParameterType method*), 252
__init__() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler method*), 314
__init__() (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler method*), 422
__init__() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler method*), 367
__init__() (*openforcefield.utils.serialization.Serializable method*), 505
__init__() (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper method*), 499
__init__() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 475
__init__() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 488
__init__() (*openforcefield.utils.toolkits.ToolkitRegistry method*), 467
__init__() (*openforcefield.utils.toolkits.ToolkitWrapper method*), 471

A

add_atom() (*openforcefield.topology.Molecule method*), 127
add_bond() (*openforcefield.topology.Atom method*), 162
add_bond() (*openforcefield.topology.Molecule method*), 129
add_bond_charge_virtual_site() (*openforcefield.topology.Molecule method*), 127
add_conformer() (*openforcefield.topology.Molecule method*), 130
add_constraint() (*openforcefield.topology.Topology method*), 147
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler method*), 302
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler.AngleTy method*), 300
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler method*), 283
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler.BondTyp method*), 282
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler method*), 400
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler method*), 450
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler.GBSATy method*), 449
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHand method*), 347
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHand method*), 346
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandle method*), 416
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandle method*), 414
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler method*), 263
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ParameterType method*), 252
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler method*), 326
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler method*), 325
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandl method*), 424
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler method*), 380
add_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler.vdWType method*), 379
add_divalent_lone_pair_virtual_site() (*open-*

forcefield.topology.Molecule method), 128
add_molecule() (*openforcefield.topology.Topology method*), 147
add_monovalent_lone_pair_virtual_site() (*openforcefield.topology.Molecule method*), 128
add_parameter() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler method*), 302
add_parameter() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler method*), 284
add_parameter() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler method*), 400
add_parameter() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler method*), 451
add_parameter() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler method*), 347
add_parameter() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandler method*), 416
add_parameter() (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler method*), 261
add_parameter() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler method*), 327
add_parameter() (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler method*), 424
add_parameter() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler method*), 380
add_particle() (*openforcefield.topology.Topology method*), 147
add_toolkit() (*openforcefield.utils.toolkits.ToolkitRegistry method*), 469
add_trivalent_lone_pair_virtual_site() (*openforcefield.topology.Molecule method*), 129
add_virtual_site() (*openforcefield.topology.Atom method*), 162
all_subclasses() (*in module openforcefield.utils.utils*), 509
amber_impropers() (*openforcefield.topology.FrozenMolecule property*), 84
amber_impropers() (*openforcefield.topology.Molecule property*), 104
amber_impropers() (*openforcefield.topology.Topology property*), 141
amber_impropers() (*openforcefield.topology.TopologyMolecule property*), 153
AmberToolsToolkitWrapper (*class in openforcefield.utils.toolkits*), 499
AngleHandler (*class in openforcefield.typing.engines.smirnoff.parameters*), 287
AngleHandler.AngleType (*class in openforcefield.typing.engines.smirnoff.parameters*), 292
angles() (*openforcefield.topology.FrozenMolecule property*), 83
angles() (*openforcefield.topology.Molecule property*), 105
angles() (*openforcefield.topology.Topology property*), 140
angles() (*openforcefield.topology.TopologyMolecule property*), 152
append() (*openforcefield.typing.engines.smirnoff.parameters.ParameterList method*), 255
apply_elf_conformer_selection() (*openforcefield.topology.FrozenMolecule method*), 79
apply_elf_conformer_selection() (*openforcefield.topology.Molecule method*), 105
apply_elf_conformer_selection() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 483
apply_elf_conformer_selection() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 495
are_isomorphic() (*openforcefield.topology.FrozenMolecule static method*), 77
are_isomorphic() (*openforcefield.topology.Molecule static method*), 105
aromaticity_model() (*openforcefield.topology.Topology property*), 138
aromaticity_model() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField property*), 244
assert_bonded() (*openforcefield.topology.Topology method*), 138
assign_charge_from_molecules() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler method*), 399
assign_fractional_bond_orders() (*openforcefield.topology.FrozenMolecule method*), 81
assign_fractional_bond_orders() (*openforcefield.topology.Molecule method*), 106
assign_fractional_bond_orders() (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper method*), 501

assign_fractional_bond_orders() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 485
assign_parameters() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler method*), 303
assign_parameters() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler method*), 285
assign_parameters() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler method*), 401
assign_parameters() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler method*), 451
assign_parameters() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler method*), 348
assign_parameters() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandler method*), 417
assign_parameters() (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler method*), 262
assign_parameters() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler method*), 327
assign_parameters() (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler method*), 425
assign_parameters() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler method*), 381
assign_partial_charges() (*openforcefield.topology.FrozenMolecule method*), 80
assign_partial_charges() (*openforcefield.topology.Molecule method*), 107
assign_partial_charges() (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper method*), 500
assign_partial_charges() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 484
assign_partial_charges() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 494
Atom (*class in openforcefield.topology*), 159
atom() (*openforcefield.topology.Topology method*), 144
atom() (*openforcefield.topology.TopologyAtom property*), 217
atom() (*openforcefield.topology.TopologyMolecule method*), 151
atom() (*openforcefield.topology.TopologyVirtualParticle method*), 184
atom() (*openforcefield.topology.TopologyVirtualSite method*), 229
atom_start_topology_index() (*openforcefield.topology.TopologyMolecule property*), 151
atomic_number() (*openforcefield.topology.Atom property*), 162
atomic_number() (*openforcefield.topology.TopologyAtom property*), 217
atoms() (*openforcefield.topology.BondChargeVirtualSite property*), 191
atoms() (*openforcefield.topology.DivalentLonePairVirtualSite property*), 205
atoms() (*openforcefield.topology.FrozenMolecule property*), 83
atoms() (*openforcefield.topology.Molecule property*), 107
atoms() (*openforcefield.topology.MonovalentLonePairVirtualSite property*), 198
atoms() (*openforcefield.topology.TopologyBond property*), 223
atoms() (*openforcefield.topology.TopologyMolecule property*), 151
atoms() (*openforcefield.topology.TopologyVirtualParticle property*), 184
atoms() (*openforcefield.topology.TopologyVirtualSite property*), 229
atoms() (*openforcefield.topology.TrivalentLonePairVirtualSite property*), 212
atoms() (*openforcefield.topology.VirtualSite property*), 174
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler method*), 303
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler.AngleTy method*), 301
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler method*), 285
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler.BondTyp method*), 282
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler method*), 401
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler method*), 452
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler.GBSATy*

method), 449
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler method*), 348
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler method*), 346
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandler method*), 417
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandler.LibraryChargeType method*), 415
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler method*), 263
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.ParameterType method*), 253
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler method*), 328
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler.ProperTorsionType method*), 325
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler method*), 425
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler method*), 381
attribute_is_cosmetic() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler.vdWType method*), 379
author() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField property*), 244

B

Bond (*class in openforcefield.topology*), 166
bond() (*openforcefield.topology.Topology method*), 144
bond() (*openforcefield.topology.TopologyBond property*), 223
bond() (*openforcefield.topology.TopologyMolecule method*), 151
bond_order() (*openforcefield.topology.TopologyBond property*), 223
bond_start_topology_index() (*openforcefield.topology.TopologyMolecule property*), 151
BondChargeVirtualSite (*class in openforcefield.topology*), 188
bonded_atoms() (*openforcefield.topology.Atom property*), 162
BondHandler (*class in openforcefield.typing.engines.smirnoff.parameters*), 344
BondHandler.BondType (*class in openforcefield.typing.engines.smirnoff.parameters*), 272
bonds() (*openforcefield.topology.Atom property*), 162
bonds() (*openforcefield.topology.FrozenMolecule property*), 83
bonds() (*openforcefield.topology.Molecule property*), 107
bonds() (*openforcefield.topology.TopologyMolecule property*), 151
box_vectors() (*openforcefield.topology.Topology property*), 138

C

call() (*openforcefield.utils.toolkits.ToolkitRegistry method*), 470
canonical_order_atoms() (*openforcefield.topology.FrozenMolecule method*), 92
canonical_order_atoms() (*openforcefield.topology.Molecule method*), 107
canonical_order_atoms() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 482
canonical_order_atoms() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 492
charge_increments() (*openforcefield.topology.BondChargeVirtualSite property*), 191
charge_increments() (*openforcefield.topology.DivalentLonePairVirtualSite property*), 205
charge_increments() (*openforcefield.topology.MonovalentLonePairVirtualSite property*), 198
charge_increments() (*openforcefield.topology.TrivalentLonePairVirtualSite property*), 212
charge_increments() (*openforcefield.topology.VirtualSite property*), 174
charge_model() (*openforcefield.topology.Topology property*), 138
check_charges_assigned() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler static method*), 401
check_charges_assigned() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandler static method*), 417
check_charges_assigned() (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler*

static method), 425
check_charges_assigned() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler static method*), 381
check_handler_compatibility() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler method*), 301
check_handler_compatibility() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler method*), 283
check_handler_compatibility() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler method*), 399
check_handler_compatibility() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler method*), 450
check_handler_compatibility() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler method*), 347
check_handler_compatibility() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandler method*), 418
check_handler_compatibility() (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler method*), 260
check_handler_compatibility() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler method*), 326
check_handler_compatibility() (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler method*), 423
check_handler_compatibility() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler method*), 380
chemical_environment_matches() (*openforcefield.topology.FrozenMolecule method*), 85
chemical_environment_matches() (*openforcefield.topology.Molecule method*), 108
chemical_environment_matches() (*openforcefield.topology.Topology method*), 141
ChemicalEnvironment (*class in openforcefield.typing.chemistry*), 237
clear() (*openforcefield.topology.ImproperDict method*), 236
clear() (*openforcefield.topology.ValenceDict method*), 234
clear() (*openforcefield.typing.engines.smirnoff.parameters.ParameterList method*), 256
compute_partial_charges_am1bcc() (*openforcefield.topology.FrozenMolecule method*), 79
compute_partial_charges_am1bcc() (*openforcefield.topology.Molecule method*), 108
compute_partial_charges_am1bcc() (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper method*), 501
compute_partial_charges_am1bcc() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 485
conformers() (*openforcefield.topology.FrozenMolecule property*), 83
conformers() (*openforcefield.topology.Molecule property*), 109
constrained_atom_pairs() (*openforcefield.topology.Topology property*), 138
convert_0_1_smirnoff_to_0_2() (*in module openforcefield.utils.utils*), 510
convert_0_2_smirnoff_to_0_3() (*in module openforcefield.utils.utils*), 510
convert() (*openforcefield.typing.engines.smirnoff.parameters.IndexedMappedParameterHandler method*), 465
convert() (*openforcefield.typing.engines.smirnoff.parameters.IndexedParameterAttributeHandler method*), 461
convert() (*openforcefield.typing.engines.smirnoff.parameters.MappedParameterAttributeHandler method*), 463
convert() (*openforcefield.typing.engines.smirnoff.parameters.ParameterAttributeHandler method*), 459
copy() (*openforcefield.typing.engines.smirnoff.parameters.ParameterList method*), 256
count() (*openforcefield.typing.engines.smirnoff.parameters.ParameterList method*), 256
create_openmm_system() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField method*), 247
create_parmed_structure() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField method*), 248

D

date() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField property*), 245
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler method*), 303
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler.AngleType method*), 301
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler method*), 285
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler.BondType*

method), 283
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler method*), 401
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler method*), 452
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler.GBSAType method*), 450
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler method*), 349
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler.ImproperTorsionType method*), 346
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandler method*), 418
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandler.LibraryChargeType method*), 415
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler method*), 263
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ParameterType method*), 253
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler method*), 328
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler.ProperTorsionType method*), 326
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler method*), 426
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler method*), 382
delete_cosmetic_attribute() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler.vdWType method*), 379
deregister_parameter_handler() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField method*), 246
deregister_toolkit() (*openforcefield.utils.toolkits.ToolkitRegistry method*), 469
distance() (*openforcefield.topology.BondChargeVirtualSite property*), 191
distance() (*openforcefield.topology.DivalentLonePairVirtualSite property*), 205
distance() (*openforcefield.topology.MonovalentLonePairVirtualSite property*), 198
distance() (*openforcefield.topology.TrivalentLonePairVirtualSite property*), 212
DivalentLonePairVirtualSite (*class in openforcefield.topology*), 202

E

ElectrostaticsHandler (*class in openforcefield.typing.engines.smirnoff.parameters*), 384
element() (*openforcefield.topology.Atom property*), 162
enumerate_protomers() (*openforcefield.topology.FrozenMolecule method*), 88
enumerate_protomers() (*openforcefield.topology.Molecule method*), 109
enumerate_protomers() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 478
enumerate_stereoisomers() (*openforcefield.topology.FrozenMolecule method*), 88
enumerate_stereoisomers() (*openforcefield.topology.Molecule method*), 109
enumerate_stereoisomers() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 479
enumerate_stereoisomers() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 492
enumerate_tautomers() (*openforcefield.topology.FrozenMolecule method*), 88
enumerate_tautomers() (*openforcefield.topology.Molecule method*), 110
enumerate_tautomers() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 479
enumerate_tautomers() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 492
epsilon() (*openforcefield.topology.BondChargeVirtualSite property*), 191
epsilon() (*openforcefield.topology.DivalentLonePairVirtualSite property*), 205
epsilon() (*openforcefield.topology.MonovalentLonePairVirtualSite*

property), 198
epsilon() (*openforcefield.topology.TrivalentLonePairVirtualSite property*), 212
epsilon() (*openforcefield.topology.VirtualSite property*), 174
extend() (*openforcefield.typing.engines.smirnoff.parameters.ParameterList method*), 256

F

find_matches() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler method*), 303
find_matches() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler method*), 285
find_matches() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler method*), 402
find_matches() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler method*), 452
find_matches() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler method*), 347
find_matches() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandler method*), 415
find_matches() (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler method*), 262
find_matches() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler method*), 328
find_matches() (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler method*), 426
find_matches() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler method*), 382
find_rings() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 498
find_rotatable_bonds() (*openforcefield.topology.FrozenMolecule method*), 82
find_rotatable_bonds() (*openforcefield.topology.Molecule method*), 110
find_smarts_matches() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 486
find_smarts_matches() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 497
ForceField (*class in openforcefield.typing.engines.smirnoff.forcefield*), 240
formal_charge() (*openforcefield.topology.Atom property*), 162
fractional_bond_order_model() (*openforcefield.topology.Topology property*), 139
from_bson() (*openforcefield.topology.Atom class method*), 163
from_bson() (*openforcefield.topology.Bond class method*), 168
from_bson() (*openforcefield.topology.BondChargeVirtualSite class method*), 191
from_bson() (*openforcefield.topology.DivalentLonePairVirtualSite class method*), 205
from_bson() (*openforcefield.topology.FrozenMolecule class method*), 93
from_bson() (*openforcefield.topology.Molecule class method*), 110
from_bson() (*openforcefield.topology.MonovalentLonePairVirtualSite class method*), 198
from_bson() (*openforcefield.topology.Particle class method*), 156
from_bson() (*openforcefield.topology.Topology class method*), 144
from_bson() (*openforcefield.topology.TopologyAtom class method*), 218
from_bson() (*openforcefield.topology.TopologyBond class method*), 223
from_bson() (*openforcefield.topology.TopologyVirtualParticle class method*), 184
from_bson() (*openforcefield.topology.TopologyVirtualSite class method*), 230
from_bson() (*openforcefield.topology.TrivalentLonePairVirtualSite class method*), 212
from_bson() (*openforcefield.topology.VirtualParticle class method*), 179
from_bson() (*openforcefield.topology.VirtualSite class method*), 174
from_bson() (*openforcefield.utils.serialization.Serializable class method*), 506
from_dict() (*openforcefield.topology.Atom class method*), 162
from_dict() (*openforcefield.topology.Bond class method*), 168
from_dict() (*openforcefield.topology.BondChargeVirtualSite class*

method), 191
from_dict() (*openforcefield.topology.DivalentLonePairVirtualSite class method*), 205
from_dict() (*openforcefield.topology.FrozenMolecule class method*), 74
from_dict() (*openforcefield.topology.Molecule class method*), 110
from_dict() (*openforcefield.topology.MonovalentLonePairVirtualSite class method*), 198
from_dict() (*openforcefield.topology.Particle class method*), 156
from_dict() (*openforcefield.topology.Topology class method*), 141
from_dict() (*openforcefield.topology.TopologyAtom class method*), 218
from_dict() (*openforcefield.topology.TopologyBond class method*), 223
from_dict() (*openforcefield.topology.TopologyMolecule class method*), 153
from_dict() (*openforcefield.topology.TopologyVirtualParticle class method*), 184
from_dict() (*openforcefield.topology.TopologyVirtualSite class method*), 230
from_dict() (*openforcefield.topology.TrivalentLonePairVirtualSite class method*), 212
from_dict() (*openforcefield.topology.VirtualParticle class method*), 179
from_dict() (*openforcefield.topology.VirtualSite class method*), 173
from_file() (*openforcefield.topology.FrozenMolecule class method*), 87
from_file() (*openforcefield.topology.Molecule class method*), 110
from_file() (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper method*), 502
from_file() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 477
from_file() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 491
from_file() (*openforcefield.utils.toolkits.ToolkitWrapper method*), 473
from_file_obj() (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper method*), 502
from_file_obj() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 478
from_file_obj() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 491
from_file_obj() (*openforcefield.utils.toolkits.ToolkitWrapper method*), 473
from_inchi() (*openforcefield.topology.FrozenMolecule class method*), 75
from_inchi() (*openforcefield.topology.Molecule class method*), 111
from_inchi() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 483
from_inchi() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 493
from_iupac() (*openforcefield.topology.FrozenMolecule class method*), 85
from_iupac() (*openforcefield.topology.Molecule class method*), 112
from_iupac() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 483
from_json() (*openforcefield.topology.Atom class method*), 163
from_json() (*openforcefield.topology.Bond class method*), 168
from_json() (*openforcefield.topology.BondChargeVirtualSite class method*), 192
from_json() (*openforcefield.topology.DivalentLonePairVirtualSite class method*), 206
from_json() (*openforcefield.topology.FrozenMolecule class method*), 93
from_json() (*openforcefield.topology.Molecule class method*), 112
from_json() (*openforcefield.topology.MonovalentLonePairVirtualSite class method*), 199
from_json() (*openforcefield.topology.Particle class method*), 156
from_json() (*openforcefield.topology.Topology class method*), 144
from_json() (*openforcefield.topology.TopologyAtom class method*), 218
from_json() (*openforcefield.topology.TopologyBond class method*), 224
from_json() (*openforce-*

field.topology.TopologyVirtualParticle class method), 185
from_json() (*openforcefield.topology.TopologyVirtualSite class method*), 230
from_json() (*openforcefield.topology.TrivalentLonePairVirtualSite class method*), 212
from_json() (*openforcefield.topology.VirtualParticle class method*), 179
from_json() (*openforcefield.topology.VirtualSite class method*), 174
from_json() (*openforcefield.utils.serialization.Serializable class method*), 506
from_mapped_smiles() (*openforcefield.topology.FrozenMolecule class method*), 90
from_mapped_smiles() (*openforcefield.topology.Molecule class method*), 112
from_mdtraj() (*openforcefield.topology.Topology static method*), 142
from_messagepack() (*openforcefield.topology.Atom class method*), 163
from_messagepack() (*openforcefield.topology.Bond class method*), 168
from_messagepack() (*openforcefield.topology.BondChargeVirtualSite class method*), 192
from_messagepack() (*openforcefield.topology.DivalentLonePairVirtualSite class method*), 206
from_messagepack() (*openforcefield.topology.FrozenMolecule class method*), 93
from_messagepack() (*openforcefield.topology.Molecule class method*), 113
from_messagepack() (*openforcefield.topology.MonovalentLonePairVirtualSite class method*), 199
from_messagepack() (*openforcefield.topology.Particle class method*), 156
from_messagepack() (*openforcefield.topology.Topology class method*), 144
from_messagepack() (*openforcefield.topology.TopologyAtom class method*), 219
from_messagepack() (*openforcefield.topology.TopologyBond class method*), 224
from_messagepack() (*openforcefield.topology.TopologyVirtualParticle class method*), 185
from_messagepack() (*openforcefield.topology.TopologyVirtualSite class method*), 230
from_messagepack() (*openforcefield.topology.TrivalentLonePairVirtualSite class method*), 213
from_messagepack() (*openforcefield.topology.VirtualParticle class method*), 179
from_messagepack() (*openforcefield.topology.VirtualSite class method*), 174
from_messagepack() (*openforcefield.utils.serialization.Serializable class method*), 507
from_molecules() (*openforcefield.topology.Topology class method*), 138
from_object() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 477
from_object() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 490
from_openeye() (*openforcefield.topology.FrozenMolecule class method*), 89
from_openeye() (*openforcefield.topology.Molecule class method*), 113
from_openeye() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper static method*), 479
from_openmm() (*openforcefield.topology.Topology class method*), 141
from_parmed() (*openforcefield.topology.Topology static method*), 143
from_pdb_and_smiles() (*openforcefield.topology.FrozenMolecule class method*), 91
from_pdb_and_smiles() (*openforcefield.topology.Molecule class method*), 113
from_pdb_and_smiles() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 490
from_pickle() (*openforcefield.topology.Atom class method*), 163
from_pickle() (*openforcefield.topology.Bond class method*), 168
from_pickle() (*openforcefield.topology.BondChargeVirtualSite class method*), 192
from_pickle() (*openforce-*

field.topology.DivalentLonePairVirtualSite class method), 206
from_pickle() (*openforcefield.topology.FrozenMolecule class method*), 94
from_pickle() (*openforcefield.topology.Molecule class method*), 114
from_pickle() (*openforcefield.topology.MonovalentLonePairVirtualSite class method*), 199
from_pickle() (*openforcefield.topology.Particle class method*), 157
from_pickle() (*openforcefield.topology.Topology class method*), 145
from_pickle() (*openforcefield.topology.TopologyAtom class method*), 219
from_pickle() (*openforcefield.topology.TopologyBond class method*), 224
from_pickle() (*openforcefield.topology.TopologyVirtualParticle class method*), 185
from_pickle() (*openforcefield.topology.TopologyVirtualSite class method*), 230
from_pickle() (*openforcefield.topology.TrivalentLonePairVirtualSite class method*), 213
from_pickle() (*openforcefield.topology.VirtualParticle class method*), 179
from_pickle() (*openforcefield.topology.VirtualSite class method*), 174
from_pickle() (*openforcefield.utils.serialization.Serializable class method*), 508
from_qcschema() (*openforcefield.topology.FrozenMolecule class method*), 91
from_qcschema() (*openforcefield.topology.Molecule class method*), 114
from_rdkit() (*openforcefield.topology.FrozenMolecule class method*), 89
from_rdkit() (*openforcefield.topology.Molecule class method*), 115
from_rdkit() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 495
from_smiles() (*openforcefield.topology.FrozenMolecule class method*), 76
from_smiles() (*openforcefield.topology.Molecule class method*), 115
from_smiles() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 482
from_smiles() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 493
from_toml() (*openforcefield.topology.Atom class method*), 164
from_toml() (*openforcefield.topology.Bond class method*), 169
from_toml() (*openforcefield.topology.BondChargeVirtualSite class method*), 192
from_toml() (*openforcefield.topology.DivalentLonePairVirtualSite class method*), 206
from_toml() (*openforcefield.topology.FrozenMolecule class method*), 94
from_toml() (*openforcefield.topology.Molecule class method*), 116
from_toml() (*openforcefield.topology.MonovalentLonePairVirtualSite class method*), 199
from_toml() (*openforcefield.topology.Particle class method*), 157
from_toml() (*openforcefield.topology.Topology class method*), 145
from_toml() (*openforcefield.topology.TopologyAtom class method*), 219
from_toml() (*openforcefield.topology.TopologyBond class method*), 224
from_toml() (*openforcefield.topology.TopologyVirtualParticle class method*), 185
from_toml() (*openforcefield.topology.TopologyVirtualSite class method*), 231
from_toml() (*openforcefield.topology.TrivalentLonePairVirtualSite class method*), 213
from_toml() (*openforcefield.topology.VirtualParticle class method*), 179
from_toml() (*openforcefield.topology.VirtualSite class method*), 175
from_toml() (*openforcefield.utils.serialization.Serializable class method*), 507
from_topology() (*openforcefield.topology.FrozenMolecule class method*), 86
from_topology() (*openforcefield.topology.Molecule class method*), 116
from_xml() (*openforcefield.topology.Atom class method*), 164
from_xml() (*openforcefield.topology.Bond class*

method), 169
from_xml() (*openforcefield.topology.BondChargeVirtualSite class method*), 192
from_xml() (*openforcefield.topology.DivalentLonePairVirtualSite class method*), 206
from_xml() (*openforcefield.topology.FrozenMolecule class method*), 94
from_xml() (*openforcefield.topology.Molecule class method*), 116
from_xml() (*openforcefield.topology.MonovalentLonePairVirtualSite class method*), 199
from_xml() (*openforcefield.topology.Particle class method*), 157
from_xml() (*openforcefield.topology.Topology class method*), 145
from_xml() (*openforcefield.topology.TopologyAtom class method*), 219
from_xml() (*openforcefield.topology.TopologyBond class method*), 224
from_xml() (*openforcefield.topology.TopologyVirtualParticle class method*), 185
from_xml() (*openforcefield.topology.TopologyVirtualSite class method*), 231
from_xml() (*openforcefield.topology.TrivalentLonePairVirtualSite class method*), 213
from_xml() (*openforcefield.topology.VirtualParticle class method*), 180
from_xml() (*openforcefield.topology.VirtualSite class method*), 175
from_xml() (*openforcefield.utils.serialization.Serializable class method*), 508
from_yaml() (*openforcefield.topology.Atom class method*), 164
from_yaml() (*openforcefield.topology.Bond class method*), 169
from_yaml() (*openforcefield.topology.BondChargeVirtualSite class method*), 193
from_yaml() (*openforcefield.topology.DivalentLonePairVirtualSite class method*), 207
from_yaml() (*openforcefield.topology.FrozenMolecule class method*), 94
from_yaml() (*openforcefield.topology.Molecule class method*), 117
from_yaml() (*openforcefield.topology.MonovalentLonePairVirtualSite class method*), 200
from_yaml() (*openforcefield.topology.Particle class method*), 157
from_yaml() (*openforcefield.topology.Topology class method*), 145
from_yaml() (*openforcefield.topology.TopologyAtom class method*), 219
from_yaml() (*openforcefield.topology.TopologyBond class method*), 225
from_yaml() (*openforcefield.topology.TopologyVirtualParticle class method*), 186
from_yaml() (*openforcefield.topology.TopologyVirtualSite class method*), 231
from_yaml() (*openforcefield.topology.TrivalentLonePairVirtualSite class method*), 213
from_yaml() (*openforcefield.topology.VirtualParticle class method*), 180
from_yaml() (*openforcefield.topology.VirtualSite class method*), 175
from_yaml() (*openforcefield.utils.serialization.Serializable class method*), 507
FrozenMolecule (*class in openforcefield.topology*), 65

G

GBSAHandler (*class in openforcefield.typing.engines.smirnoff.parameters*), 427
GBSAHandler.GBSAType (*class in openforcefield.typing.engines.smirnoff.parameters*), 441
generate_conformers() (*openforcefield.topology.FrozenMolecule method*), 78
generate_conformers() (*openforcefield.topology.Molecule method*), 117
generate_conformers() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 483
generate_conformers() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 494
generate_unique_atom_names() (*openforcefield.topology.FrozenMolecule method*), 74
generate_unique_atom_names() (*openforcefield.topology.Molecule method*), 117
get() (*openforcefield.topology.ImproperDict method*), 237
get() (*openforcefield.topology.ValenceDict method*), 234

get_available_force_fields() (*in module openforcefield.typing.engines.smirnoff.forcefield*), 249
get_bond_between() (*openforcefield.topology.FrozenMolecule method*), 93
get_bond_between() (*openforcefield.topology.Molecule method*), 117
get_bond_between() (*openforcefield.topology.Topology method*), 143
get_data_file_path() (*in module openforcefield.utils.utils*), 510
get_molecule_parameterIDs() (*in module openforcefield.utils.utils*), 510
get_openmm_virtual_site() (*openforcefield.topology.BondChargeVirtualSite method*), 191
get_openmm_virtual_site() (*openforcefield.topology.DivalentLonePairVirtualSite method*), 205
get_openmm_virtual_site() (*openforcefield.topology.MonovalentLonePairVirtualSite method*), 198
get_openmm_virtual_site() (*openforcefield.topology.TrivalentLonePairVirtualSite method*), 212
get_parameter() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler method*), 304
get_parameter() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler method*), 285
get_parameter() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler method*), 402
get_parameter() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler method*), 452
get_parameter() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler method*), 349
get_parameter() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandler method*), 418
get_parameter() (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler method*), 261
get_parameter() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler method*), 328
get_parameter() (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler method*), 426
get_parameter() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler method*), 382
get_parameter_handler() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField method*), 245
get_parameter_io_handler() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField method*), 246
get_tagged_smarts_connectivity() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 485
get_tagged_smarts_connectivity() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 497
get_type() (*openforcefield.typing.chemistry.ChemicalEnvironment method*), 239

H

has_unique_atom_names() (*openforcefield.topology.FrozenMolecule property*), 74
has_unique_atom_names() (*openforcefield.topology.Molecule property*), 117
hill_formula() (*openforcefield.topology.FrozenMolecule property*), 84
hill_formula() (*openforcefield.topology.Molecule property*), 118

I

ImproperDict (*class in openforcefield.topology*), 235
impropers() (*openforcefield.topology.FrozenMolecule property*), 83
impropers() (*openforcefield.topology.Molecule property*), 118
impropers() (*openforcefield.topology.Topology property*), 140
impropers() (*openforcefield.topology.TopologyMolecule property*), 152
ImproperTorsionHandler (*class in openforcefield.typing.engines.smirnoff.parameters*), 330
ImproperTorsionHandler.ImproperTorsionType (*class in openforcefield.typing.engines.smirnoff.parameters*), 336
in_plane_angle() (*openforcefield.topology.MonovalentLonePairVirtualSite property*), 198
index() (*openforcefield.typing.engines.smirnoff.parameters.ParameterList method*), 256

index_of() (*openforcefield.topology.ImproperDict static method*), 236
index_of() (*openforcefield.topology.ValenceDict static method*), 234
IndexedMappedParameterAttribute (*class in openforcefield.typing.engines.smirnoff.parameters*), 463
IndexedMappedParameterAttribute.UNDEFINED (*class in openforcefield.typing.engines.smirnoff.parameters*), 465
IndexedParameterAttribute (*class in openforcefield.typing.engines.smirnoff.parameters*), 459
IndexedParameterAttribute.UNDEFINED (*class in openforcefield.typing.engines.smirnoff.parameters*), 461
inherit_docstrings() (*in module openforcefield.utils.utils*), 509
insert() (*openforcefield.typing.engines.smirnoff.parameters.ParameterList method*), 256
is_aromatic() (*openforcefield.topology.Atom property*), 162
is_available() (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper static method*), 500
is_available() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper class method*), 477
is_available() (*openforcefield.utils.toolkits.RDKitToolkitWrapper class method*), 490
is_available() (*openforcefield.utils.toolkits.ToolkitWrapper class method*), 473
is_bonded() (*openforcefield.topology.Topology method*), 143
is_bonded_to() (*openforcefield.topology.Atom method*), 163
is_constrained() (*openforcefield.topology.Topology method*), 147
is_in_ring() (*openforcefield.topology.Atom property*), 163
is_in_ring() (*openforcefield.topology.Bond property*), 168
is_isomorphic_with() (*openforcefield.topology.FrozenMolecule method*), 78
is_isomorphic_with() (*openforcefield.topology.Molecule method*), 118
is_periodic() (*openforcefield.topology.Topology property*), 138
items() (*openforcefield.topology.ImproperDict method*), 237
items() (*openforcefield.topology.ValenceDict method*), 234

K

key_transform() (*openforcefield.topology.ImproperDict static method*), 236
key_transform() (*openforcefield.topology.ValenceDict static method*), 234
keys() (*openforcefield.topology.ImproperDict method*), 237
keys() (*openforcefield.topology.ValenceDict method*), 234
known_kwargs() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler property*), 304
known_kwargs() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler property*), 286
known_kwargs() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler property*), 402
known_kwargs() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler property*), 453
known_kwargs() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHand property*), 349
known_kwargs() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandle property*), 419
known_kwargs() (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler property*), 260
known_kwargs() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler property*), 329
known_kwargs() (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandl property*), 427
known_kwargs() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler property*), 383

L

label_molecules() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField method*), 248
LibraryChargeHandler (*class in openforcefield.typing.engines.smirnoff.parameters*), 403

LibraryChargeHandler.LibraryChargeType (*class in openforcefield.typing.engines.smirnoff.parameters*), 407

M

MappedParameterAttribute (*class in openforcefield.typing.engines.smirnoff.parameters*), 461
MappedParameterAttribute.UNDEFINED (*class in openforcefield.typing.engines.smirnoff.parameters*), 463
mark_charges_assigned() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler method*), 402
mark_charges_assigned() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandler method*), 419
mark_charges_assigned() (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler method*), 427
mark_charges_assigned() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler method*), 383
mass() (*openforcefield.topology.Atom property*), 162
Molecule (*class in openforcefield.topology*), 96
molecule() (*openforcefield.topology.Atom property*), 164
molecule() (*openforcefield.topology.BondChargeVirtualSite property*), 193
molecule() (*openforcefield.topology.DivalentLonePairVirtualSite property*), 207
molecule() (*openforcefield.topology.MonovalentLonePairVirtualSite property*), 200
molecule() (*openforcefield.topology.Particle property*), 156
molecule() (*openforcefield.topology.TopologyAtom property*), 218
molecule() (*openforcefield.topology.TopologyBond property*), 223
molecule() (*openforcefield.topology.TopologyVirtualParticle property*), 186
molecule() (*openforcefield.topology.TopologyVirtualSite property*), 229
molecule() (*openforcefield.topology.TrivalentLonePairVirtualSite property*), 214
molecule() (*openforcefield.topology.VirtualParticle property*), 180
molecule() (*openforcefield.topology.VirtualSite property*), 175
molecule_atom_index() (*openforcefield.topology.Atom property*), 163
molecule_bond_index() (*openforcefield.topology.Bond property*), 168
molecule_particle_index() (*openforcefield.topology.Atom property*), 163
molecule_particle_index() (*openforcefield.topology.BondChargeVirtualSite property*), 193
molecule_particle_index() (*openforcefield.topology.DivalentLonePairVirtualSite property*), 207
molecule_particle_index() (*openforcefield.topology.MonovalentLonePairVirtualSite property*), 200
molecule_particle_index() (*openforcefield.topology.Particle property*), 156
molecule_particle_index() (*openforcefield.topology.TrivalentLonePairVirtualSite property*), 214
molecule_particle_index() (*openforcefield.topology.VirtualParticle property*), 180
molecule_particle_index() (*openforcefield.topology.VirtualSite property*), 175
molecule_virtual_site_index() (*openforcefield.topology.BondChargeVirtualSite property*), 193
molecule_virtual_site_index() (*openforcefield.topology.DivalentLonePairVirtualSite property*), 207
molecule_virtual_site_index() (*openforcefield.topology.MonovalentLonePairVirtualSite property*), 200
molecule_virtual_site_index() (*openforcefield.topology.TrivalentLonePairVirtualSite property*), 214
molecule_virtual_site_index() (*openforcefield.topology.VirtualSite property*), 174
MonovalentLonePairVirtualSite (*class in openforcefield.topology*), 195

N

n_angles() (*openforcefield.topology.FrozenMolecule property*), 82
n_angles() (*openforcefield.topology.Molecule property*), 118
n_angles() (*openforcefield.topology.Topology property*), 140
n_angles() (*openforcefield.topology.TopologyMolecule property*), 152

n_atoms() (*openforcefield.topology.FrozenMolecule property*), 82
n_atoms() (*openforcefield.topology.Molecule property*), 119
n_atoms() (*openforcefield.topology.TopologyMolecule property*), 151
n_bonds() (*openforcefield.topology.FrozenMolecule property*), 82
n_bonds() (*openforcefield.topology.Molecule property*), 119
n_bonds() (*openforcefield.topology.TopologyMolecule property*), 151
n_conformers() (*openforcefield.topology.FrozenMolecule property*), 83
n_conformers() (*openforcefield.topology.Molecule property*), 119
n_impropers() (*openforcefield.topology.FrozenMolecule property*), 82
n_impropers() (*openforcefield.topology.Molecule property*), 119
n_impropers() (*openforcefield.topology.Topology property*), 140
n_impropers() (*openforcefield.topology.TopologyMolecule property*), 152
n_particles() (*openforcefield.topology.BondChargeVirtualSite property*), 193
n_particles() (*openforcefield.topology.DivalentLonePairVirtualSite property*), 207
n_particles() (*openforcefield.topology.FrozenMolecule property*), 82
n_particles() (*openforcefield.topology.Molecule property*), 119
n_particles() (*openforcefield.topology.MonovalentLonePairVirtualSite property*), 200
n_particles() (*openforcefield.topology.TopologyMolecule property*), 152
n_particles() (*openforcefield.topology.TopologyVirtualParticle property*), 186
n_particles() (*openforcefield.topology.TopologyVirtualSite property*), 229
n_particles() (*openforcefield.topology.TrivalentLonePairVirtualSite property*), 214
n_particles() (*openforcefield.topology.VirtualSite property*), 173
n_propers() (*openforcefield.topology.FrozenMolecule property*), 82
n_propers() (*openforcefield.topology.Molecule property*), 119
n_propers() (*openforcefield.topology.Topology property*), 140
n_propers() (*openforcefield.topology.TopologyMolecule property*), 152
n_reference_molecules() (*openforcefield.topology.Topology property*), 139
n_rings() (*openforcefield.topology.FrozenMolecule property*), 82
n_rings() (*openforcefield.topology.Molecule property*), 119
n_topology_atoms() (*openforcefield.topology.Topology property*), 139
n_topology_bonds() (*openforcefield.topology.Topology property*), 139
n_topology_molecules() (*openforcefield.topology.Topology property*), 139
n_topology_particles() (*openforcefield.topology.Topology property*), 139
n_topology_virtual_sites() (*openforcefield.topology.Topology property*), 140
n_virtual_particles() (*openforcefield.topology.FrozenMolecule property*), 82
n_virtual_particles() (*openforcefield.topology.Molecule property*), 119
n_virtual_sites() (*openforcefield.topology.FrozenMolecule property*), 82
n_virtual_sites() (*openforcefield.topology.Molecule property*), 119
n_virtual_sites() (*openforcefield.topology.TopologyMolecule property*), 152
name() (*openforcefield.topology.Atom property*), 162
name() (*openforcefield.topology.BondChargeVirtualSite property*), 193
name() (*openforcefield.topology.DivalentLonePairVirtualSite property*), 207
name() (*openforcefield.topology.FrozenMolecule property*), 84
name() (*openforcefield.topology.Molecule property*), 119
name() (*openforcefield.topology.MonovalentLonePairVirtualSite property*), 200
name() (*openforcefield.topology.Particle property*), 156
name() (*openforcefield.topology.TrivalentLonePairVirtualSite property*), 214

name() (*openforcefield.topology.VirtualParticle property*), 180
name() (*openforcefield.topology.VirtualSite property*), 174

O

OpenEyeToolkitWrapper (*class in openforcefield.utils.toolkits*), 474
out_of_plane_angle() (*openforcefield.topology.DivalentLonePairVirtualSite property*), 205
out_of_plane_angle() (*openforcefield.topology.MonovalentLonePairVirtualSite property*), 198

P

ParameterAttribute (*class in openforcefield.typing.engines.smirnoff.parameters*), 457
ParameterAttribute.UNDEFINED (*class in openforcefield.typing.engines.smirnoff.parameters*), 459
ParameterHandler (*class in openforcefield.typing.engines.smirnoff.parameters*), 257
ParameterIOHandler (*class in openforcefield.typing.engines.smirnoff.io*), 454
ParameterList (*class in openforcefield.typing.engines.smirnoff.parameters*), 254
parameters() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler property*), 304
parameters() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler property*), 286
parameters() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler property*), 403
parameters() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler property*), 453
parameters() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler property*), 349
parameters() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandler property*), 419
parameters() (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler property*), 260
parameters() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler property*), 329
parameters() (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandl property*), 427
parameters() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler property*), 383
ParameterType (*class in openforcefield.typing.engines.smirnoff.parameters*), 249
parse_file() (*openforcefield.typing.engines.smirnoff.io.ParameterIOHandler method*), 454
parse_file() (*openforcefield.typing.engines.smirnoff.io.XMLParameterIOHandler method*), 455
parse_smirnoff_from_source() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField method*), 247
parse_sources() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField method*), 246
parse_string() (*openforcefield.typing.engines.smirnoff.io.ParameterIOHandler method*), 454
parse_string() (*openforcefield.typing.engines.smirnoff.io.XMLParameterIOHandler method*), 456
partial_charge() (*openforcefield.topology.Atom property*), 162
partial_charges() (*openforcefield.topology.FrozenMolecule property*), 82
partial_charges() (*openforcefield.topology.Molecule property*), 119
Particle (*class in openforcefield.topology*), 154
particle() (*openforcefield.topology.TopologyMolecule method*), 151
particles() (*openforcefield.topology.BondChargeVirtualSite property*), 193
particles() (*openforcefield.topology.DivalentLonePairVirtualSite property*), 207
particles() (*openforcefield.topology.FrozenMolecule property*), 83
particles() (*openforcefield.topology.Molecule property*), 119
particles() (*openforcefield.topology.MonovalentLonePairVirtualSite property*), 200
particles() (*openforcefield.topology.TopologyMolecule property*), 158
particles() (*openforce-*

field.topology.TopologyVirtualParticle property), 186
particles() (*openforcefield.topology.TopologyVirtualSite property*), 229
particles() (*openforcefield.topology.TrivalentLonePairVirtualSite property*), 214
particles() (*openforcefield.topology.VirtualSite property*), 173
pop() (*openforcefield.topology.ImproperDict method*), 237
pop() (*openforcefield.topology.ValenceDict method*), 234
pop() (*openforcefield.typing.engines.smirnoff.parameters.ParameterList method*), 256
popitem() (*openforcefield.topology.ImproperDict method*), 237
popitem() (*openforcefield.topology.ValenceDict method*), 234
postprocess_system() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler method*), 304
postprocess_system() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler method*), 286
postprocess_system() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler method*), 399
postprocess_system() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler method*), 453
postprocess_system() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler method*), 349
postprocess_system() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandler method*), 419
postprocess_system() (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler method*), 262
postprocess_system() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler method*), 329
postprocess_system() (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler method*), 423
postprocess_system() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler method*), 383
propers() (*openforcefield.topology.FrozenMolecule property*), 83
propers() (*openforcefield.topology.Molecule property*), 119
propers() (*openforcefield.topology.Topology property*), 140
propers() (*openforcefield.topology.TopologyMolecule property*), 152
properties() (*openforcefield.topology.FrozenMolecule property*), 84
properties() (*openforcefield.topology.Molecule property*), 119
ProperTorsionHandler (*class in openforcefield.typing.engines.smirnoff.parameters*), 305
ProperTorsionHandler.ProperTorsionType (*class in openforcefield.typing.engines.smirnoff.parameters*), 315

R

RDKitToolkitWrapper (*class in openforcefield.utils.toolkits*), 487
reference_molecule() (*openforcefield.topology.TopologyMolecule property*), 150
reference_molecules() (*openforcefield.topology.Topology property*), 138
register_parameter_handler() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField method*), 245
register_parameter_io_handler() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField method*), 245
register_toolkit() (*openforcefield.utils.toolkits.ToolkitRegistry method*), 469
registered_parameter_handlers() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField property*), 245
registered_toolkit_versions() (*openforcefield.utils.toolkits.ToolkitRegistry property*), 468
registered_toolkits() (*openforcefield.utils.toolkits.ToolkitRegistry property*), 468
remap() (*openforcefield.topology.FrozenMolecule method*), 92
remap() (*openforcefield.topology.Molecule method*), 119
remove() (*openforcefield.typing.engines.smirnoff.parameters.ParameterList method*), 256
resolve() (*openforcefield.utils.toolkits.ToolkitRegistry method*), 469
reverse() (*openforcefield.typing.engines.smirnoff.parameters.ParameterList method*), 257

rings() (*openforcefield.topology.FrozenMolecule property*), 94
rings() (*openforcefield.topology.Molecule property*), 120
rmin_half() (*openforcefield.topology.BondChargeVirtualSite property*), 193
rmin_half() (*openforcefield.topology.DivalentLonePairVirtualSite property*), 207
rmin_half() (*openforcefield.topology.MonovalentLonePairVirtualSite property*), 200
rmin_half() (*openforcefield.topology.TrivalentLonePairVirtualSite property*), 214
rmin_half() (*openforcefield.topology.VirtualSite property*), 174

S

Serializable (*class in openforcefield.utils.serialization*), 503
setdefault() (*openforcefield.topology.ImproperDict method*), 237
setdefault() (*openforcefield.topology.ValenceDict method*), 234
sigma() (*openforcefield.topology.BondChargeVirtualSite property*), 193
sigma() (*openforcefield.topology.DivalentLonePairVirtualSite property*), 207
sigma() (*openforcefield.topology.MonovalentLonePairVirtualSite property*), 200
sigma() (*openforcefield.topology.TrivalentLonePairVirtualSite property*), 214
sigma() (*openforcefield.topology.VirtualSite property*), 174
smirnoff_impropers() (*openforcefield.topology.FrozenMolecule property*), 83
smirnoff_impropers() (*openforcefield.topology.Molecule property*), 120
smirnoff_impropers() (*openforcefield.topology.Topology property*), 140
smirnoff_impropers() (*openforcefield.topology.TopologyMolecule property*), 152
sort() (*openforcefield.typing.engines.smirnoff.parameters.ParameterList method*), 257
stereochemistry() (*openforcefield.topology.Atom property*), 162
strip_atom_stereochemistry() (*openforcefield.topology.FrozenMolecule method*), 74
strip_atom_stereochemistry() (*openforcefield.topology.Molecule method*), 120

T

TAGNAME() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler property*), 301
TAGNAME() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler property*), 283
TAGNAME() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler property*), 399
TAGNAME() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler property*), 450
TAGNAME() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHand property*), 347
TAGNAME() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandle property*), 416
TAGNAME() (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler property*), 260
TAGNAME() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler property*), 326
TAGNAME() (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandl property*), 424
TAGNAME() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler property*), 380
temporary_cd() (*in module openforcefield.utils.utils*), 509
to_bson() (*openforcefield.topology.Atom method*), 164
to_bson() (*openforcefield.topology.Bond method*), 169
to_bson() (*openforcefield.topology.BondChargeVirtualSite method*), 193
to_bson() (*openforcefield.topology.DivalentLonePairVirtualSite method*), 207
to_bson() (*openforcefield.topology.FrozenMolecule method*), 95
to_bson() (*openforcefield.topology.Molecule method*), 121
to_bson() (*openforcefield.topology.MonovalentLonePairVirtualSite method*), 200

to_bson() (*openforcefield.topology.Particle method*), 157
to_bson() (*openforcefield.topology.Topology method*), 145
to_bson() (*openforcefield.topology.TopologyAtom method*), 220
to_bson() (*openforcefield.topology.TopologyBond method*), 225
to_bson() (*openforcefield.topology.TopologyVirtualParticle method*), 186
to_bson() (*openforcefield.topology.TopologyVirtualSite method*), 231
to_bson() (*openforcefield.topology.TrivalentLonePairVirtualSite method*), 214
to_bson() (*openforcefield.topology.VirtualParticle method*), 180
to_bson() (*openforcefield.topology.VirtualSite method*), 175
to_bson() (*openforcefield.utils.serialization.Serializable method*), 506
to_dict() (*openforcefield.topology.Atom method*), 162
to_dict() (*openforcefield.topology.Bond method*), 168
to_dict() (*openforcefield.topology.BondChargeVirtualSite method*), 191
to_dict() (*openforcefield.topology.DivalentLonePairVirtualSite method*), 205
to_dict() (*openforcefield.topology.FrozenMolecule method*), 74
to_dict() (*openforcefield.topology.Molecule method*), 121
to_dict() (*openforcefield.topology.MonovalentLonePairVirtualSite method*), 198
to_dict() (*openforcefield.topology.Particle method*), 156
to_dict() (*openforcefield.topology.Topology method*), 141
to_dict() (*openforcefield.topology.TopologyAtom method*), 218
to_dict() (*openforcefield.topology.TopologyBond method*), 223
to_dict() (*openforcefield.topology.TopologyMolecule method*), 153
to_dict() (*openforcefield.topology.TopologyVirtualParticle method*), 186
to_dict() (*openforcefield.topology.TopologyVirtualSite method*), 230
to_dict() (*openforcefield.topology.TrivalentLonePairVirtualSite method*), 212
to_dict() (*openforcefield.topology.VirtualParticle method*), 180
to_dict() (*openforcefield.topology.VirtualSite method*), 173
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler method*), 304
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.AngleHandler.AngleTy method*), 301
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler method*), 286
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.BondHandler.BondTyp method*), 283
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.ElectrostaticsHandler method*), 403
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler method*), 453
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.GBSAHandler.GBSATy method*), 450
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHand method*), 350
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHand method*), 346
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandle method*), 419
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.LibraryChargeHandle method*), 415
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.ParameterHandler method*), 263
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.ParameterType method*), 253
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler method*), 329
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler

method), 326
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler method*), 427
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler method*), 383
to_dict() (*openforcefield.typing.engines.smirnoff.parameters.vdWHandler.vdWType method*), 379
to_file() (*openforcefield.topology.FrozenMolecule method*), 87
to_file() (*openforcefield.topology.Molecule method*), 121
to_file() (*openforcefield.topology.Topology method*), 142
to_file() (*openforcefield.typing.engines.smirnoff.forcefield.ForceField method*), 247
to_file() (*openforcefield.typing.engines.smirnoff.io.ParameterIOHandler method*), 455
to_file() (*openforcefield.typing.engines.smirnoff.io.XMLParameterIOHandler method*), 456
to_file() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 478
to_file() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 492
to_file_obj() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 478
to_file_obj() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 491
to_hill_formula() (*openforcefield.topology.FrozenMolecule static method*), 84
to_hill_formula() (*openforcefield.topology.Molecule static method*), 121
to_inchi() (*openforcefield.topology.FrozenMolecule method*), 75
to_inchi() (*openforcefield.topology.Molecule method*), 122
to_inchi() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 481
to_inchi() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 496
to_inchikey() (*openforcefield.topology.FrozenMolecule method*), 76
to_inchikey() (*openforcefield.topology.Molecule method*), 122
to_inchikey() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 481
to_inchikey() (*openforcefield.utils.toolkits.RDKitToolkitWrapper method*), 497
to_iupac() (*openforcefield.topology.FrozenMolecule method*), 86
to_iupac() (*openforcefield.topology.Molecule method*), 123
to_iupac() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper method*), 482
to_json() (*openforcefield.topology.Atom method*), 164
to_json() (*openforcefield.topology.Bond method*), 169
to_json() (*openforcefield.topology.BondChargeVirtualSite method*), 193
to_json() (*openforcefield.topology.DivalentLonePairVirtualSite method*), 207
to_json() (*openforcefield.topology.FrozenMolecule method*), 95
to_json() (*openforcefield.topology.Molecule method*), 123
to_json() (*openforcefield.topology.MonovalentLonePairVirtualSite method*), 200
to_json() (*openforcefield.topology.Particle method*), 158
to_json() (*openforcefield.topology.Topology method*), 145
to_json() (*openforcefield.topology.TopologyAtom method*), 220
to_json() (*openforcefield.topology.TopologyBond method*), 225
to_json() (*openforcefield.topology.TopologyVirtualParticle method*), 186
to_json() (*openforcefield.topology.TopologyVirtualSite method*), 231
to_json() (*openforcefield.topology.TrivalentLonePairVirtualSite method*), 214
to_json() (*openforcefield.topology.VirtualParticle method*), 180
to_json() (*openforcefield.topology.VirtualSite method*), 176

to_json() (*openforcefield.utils.serialization.Serializable method*), 506
to_list() (*openforcefield.typing.engines.smirnoff.parameters.ParameterList method*), 256
to_messagepack() (*openforcefield.topology.Atom method*), 165
to_messagepack() (*openforcefield.topology.Bond method*), 170
to_messagepack() (*openforcefield.topology.BondChargeVirtualSite method*), 194
to_messagepack() (*openforcefield.topology.DivalentLonePairVirtualSite method*), 207
to_messagepack() (*openforcefield.topology.FrozenMolecule method*), 95
to_messagepack() (*openforcefield.topology.Molecule method*), 123
to_messagepack() (*openforcefield.topology.MonovalentLonePairVirtualSite method*), 201
to_messagepack() (*openforcefield.topology.Particle method*), 158
to_messagepack() (*openforcefield.topology.Topology method*), 146
to_messagepack() (*openforcefield.topology.TopologyAtom method*), 220
to_messagepack() (*openforcefield.topology.TopologyBond method*), 225
to_messagepack() (*openforcefield.topology.TopologyVirtualParticle method*), 186
to_messagepack() (*openforcefield.topology.TopologyVirtualSite method*), 231
to_messagepack() (*openforcefield.topology.TrivalentLonePairVirtualSite method*), 214
to_messagepack() (*openforcefield.topology.VirtualParticle method*), 180
to_messagepack() (*openforcefield.topology.VirtualSite method*), 176
to_messagepack() (*openforcefield.utils.serialization.Serializable method*), 507
to_networkx() (*openforcefield.topology.FrozenMolecule method*), 81
to_networkx() (*openforcefield.topology.Molecule method*), 123
to_openeye() (*openforcefield.topology.FrozenMolecule method*), 93
to_openeye() (*openforcefield.topology.Molecule method*), 124
to_openeye() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper static method*), 480
to_openmm() (*openforcefield.topology.Topology method*), 142
to_parmed() (*openforcefield.topology.Topology method*), 143
to_pickle() (*openforcefield.topology.Atom method*), 165
to_pickle() (*openforcefield.topology.Bond method*), 170
to_pickle() (*openforcefield.topology.BondChargeVirtualSite method*), 194
to_pickle() (*openforcefield.topology.DivalentLonePairVirtualSite method*), 208
to_pickle() (*openforcefield.topology.FrozenMolecule method*), 95
to_pickle() (*openforcefield.topology.Molecule method*), 124
to_pickle() (*openforcefield.topology.MonovalentLonePairVirtualSite method*), 201
to_pickle() (*openforcefield.topology.Particle method*), 158
to_pickle() (*openforcefield.topology.Topology method*), 146
to_pickle() (*openforcefield.topology.TopologyAtom method*), 220
to_pickle() (*openforcefield.topology.TopologyBond method*), 225
to_pickle() (*openforcefield.topology.TopologyVirtualParticle method*), 187
to_pickle() (*openforcefield.topology.TopologyVirtualSite method*), 232
to_pickle() (*openforcefield.topology.TrivalentLonePairVirtualSite method*), 214
to_pickle() (*openforcefield.topology.VirtualParticle method*), 181
to_pickle() (*openforcefield.topology.VirtualSite method*), 176
to_pickle() (*openforcefield.utils.serialization.Serializable method*),

508
to_qcschema() (openforcefield.topology.FrozenMolecule method), 90
to_qcschema() (openforcefield.topology.Molecule method), 124
to_rdkit() (openforcefield.topology.FrozenMolecule method), 89
to_rdkit() (openforcefield.topology.Molecule method), 125
to_rdkit() (openforcefield.utils.toolkits.RDKitToolkitWrapper class method), 496
to_smiles() (openforcefield.topology.FrozenMolecule method), 74
to_smiles() (openforcefield.topology.Molecule method), 125
to_smiles() (openforcefield.utils.toolkits.OpenEyeToolkitWrapper method), 481
to_smiles() (openforcefield.utils.toolkits.RDKitToolkitWrapper method), 493
to_string() (openforcefield.typing.engines.smirnoff.forcefield.ForceField method), 247
to_string() (openforcefield.typing.engines.smirnoff.io.ParameterIOHandler method), 455
to_string() (openforcefield.typing.engines.smirnoff.io.XMLParameterIOHandler method), 456
to_toml() (openforcefield.topology.Atom method), 165
to_toml() (openforcefield.topology.Bond method), 170
to_toml() (openforcefield.topology.BondChargeVirtualSite method), 194
to_toml() (openforcefield.topology.DivalentLonePairVirtualSite method), 208
to_toml() (openforcefield.topology.FrozenMolecule method), 95
to_toml() (openforcefield.topology.Molecule method), 126
to_toml() (openforcefield.topology.MonovalentLonePairVirtualSite method), 201
to_toml() (openforcefield.topology.Particle method), 158
to_toml() (openforcefield.topology.Topology method), 146
to_toml() (openforcefield.topology.TopologyAtom method), 220
to_toml() (openforcefield.topology.TopologyBond method), 226
to_toml() (openforcefield.topology.TopologyVirtualParticle method), 187
to_toml() (openforcefield.topology.TopologyVirtualSite method), 232
to_toml() (openforcefield.topology.TrivalentLonePairVirtualSite method), 215
to_toml() (openforcefield.topology.VirtualParticle method), 181
to_toml() (openforcefield.topology.VirtualSite method), 176
to_toml() (openforcefield.utils.serialization.Serializable method), 506
to_topology() (openforcefield.topology.FrozenMolecule method), 86
to_topology() (openforcefield.topology.Molecule method), 126
to_xml() (openforcefield.topology.Atom method), 165
to_xml() (openforcefield.topology.Bond method), 170
to_xml() (openforcefield.topology.BondChargeVirtualSite method), 194
to_xml() (openforcefield.topology.DivalentLonePairVirtualSite method), 208
to_xml() (openforcefield.topology.FrozenMolecule method), 95
to_xml() (openforcefield.topology.Molecule method), 126
to_xml() (openforcefield.topology.MonovalentLonePairVirtualSite method), 201
to_xml() (openforcefield.topology.Particle method), 158
to_xml() (openforcefield.topology.Topology method), 146
to_xml() (openforcefield.topology.TopologyAtom method), 220
to_xml() (openforcefield.topology.TopologyBond method), 226
to_xml() (openforcefield.topology.TopologyVirtualParticle method), 187
to_xml() (openforcefield.topology.TopologyVirtualSite method), 232
to_xml() (openforcefield.topology.TrivalentLonePairVirtualSite method), 215
to_xml() (openforcefield.topology.VirtualParticle method), 181
to_xml() (openforcefield.topology.VirtualSite method), 176

to_xml() (*openforcefield.utils.serialization.Serializable method*), 507
to_yaml() (*openforcefield.topology.Atom method*), 165
to_yaml() (*openforcefield.topology.Bond method*), 170
to_yaml() (*openforcefield.topology.BondChargeVirtualSite method*), 194
to_yaml() (*openforcefield.topology.DivalentLonePairVirtualSite method*), 208
to_yaml() (*openforcefield.topology.FrozenMolecule method*), 96
to_yaml() (*openforcefield.topology.Molecule method*), 126
to_yaml() (*openforcefield.topology.MonovalentLonePairVirtualSite method*), 201
to_yaml() (*openforcefield.topology.Particle method*), 158
to_yaml() (*openforcefield.topology.Topology method*), 146
to_yaml() (*openforcefield.topology.TopologyAtom method*), 221
to_yaml() (*openforcefield.topology.TopologyBond method*), 226
to_yaml() (*openforcefield.topology.TopologyVirtualParticle method*), 187
to_yaml() (*openforcefield.topology.TopologyVirtualSite method*), 232
to_yaml() (*openforcefield.topology.TrivalentLonePairVirtualSite method*), 215
to_yaml() (*openforcefield.topology.VirtualParticle method*), 181
to_yaml() (*openforcefield.topology.VirtualSite method*), 176
to_yaml() (*openforcefield.utils.serialization.Serializable method*), 507
toolkit_file_read_formats() (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper property*), 502
toolkit_file_read_formats() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper property*), 486
toolkit_file_read_formats() (*openforcefield.utils.toolkits.RDKitToolkitWrapper property*), 498
toolkit_file_read_formats() (*openforcefield.utils.toolkits.ToolkitWrapper property*), 472
toolkit_file_write_formats() (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper property*), 502
toolkit_file_write_formats() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper property*), 486
toolkit_file_write_formats() (*openforcefield.utils.toolkits.RDKitToolkitWrapper property*), 490
toolkit_file_write_formats() (*openforcefield.utils.toolkits.ToolkitWrapper property*), 472
toolkit_installation_instructions() (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper property*), 502
toolkit_installation_instructions() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper property*), 486
toolkit_installation_instructions() (*openforcefield.utils.toolkits.RDKitToolkitWrapper property*), 498
toolkit_installation_instructions() (*openforcefield.utils.toolkits.ToolkitWrapper property*), 472
toolkit_name() (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper property*), 503
toolkit_name() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper property*), 487
toolkit_name() (*openforcefield.utils.toolkits.RDKitToolkitWrapper property*), 498
toolkit_name() (*openforcefield.utils.toolkits.ToolkitWrapper property*), 472
toolkit_version() (*openforcefield.utils.toolkits.AmberToolsToolkitWrapper property*), 503
toolkit_version() (*openforcefield.utils.toolkits.OpenEyeToolkitWrapper property*), 487
toolkit_version() (*openforcefield.utils.toolkits.RDKitToolkitWrapper property*), 498
toolkit_version() (*openforcefield.utils.toolkits.ToolkitWrapper property*), 473
ToolkitAM1BCCHandler (*class in openforcefield.typing.engines.smirnoff.parameters*), 419
ToolkitRegistry (*class in openforcefield.utils.toolkits*), 466

ToolkitWrapper (*class in openforcefield.utils.toolkits*), 471
Topology (*class in openforcefield.topology*), 130
topology() (*openforcefield.topology.TopologyMolecule property*), 150
topology_atom_index() (*openforcefield.topology.TopologyAtom property*), 218
topology_atoms() (*openforcefield.topology.Topology property*), 139
topology_bond_index() (*openforcefield.topology.TopologyBond property*), 223
topology_bonds() (*openforcefield.topology.Topology property*), 139
topology_bonds() (*openforcefield.topology.TopologyAtom property*), 218
topology_molecule() (*openforcefield.topology.TopologyAtom property*), 218
topology_molecule() (*openforcefield.topology.TopologyBond property*), 223
topology_molecule() (*openforcefield.topology.TopologyVirtualParticle property*), 187
topology_molecule() (*openforcefield.topology.TopologyVirtualSite property*), 229
topology_molecules() (*openforcefield.topology.Topology property*), 139
topology_particle_index() (*openforcefield.topology.TopologyAtom property*), 218
topology_particle_index() (*openforcefield.topology.TopologyVirtualParticle property*), 184
topology_particles() (*openforcefield.topology.Topology property*), 140
topology_virtual_particle_start_index() (*openforcefield.topology.TopologyVirtualParticle property*), 187
topology_virtual_particle_start_index() (*openforcefield.topology.TopologyVirtualSite property*), 229
topology_virtual_site_index() (*openforcefield.topology.TopologyVirtualParticle property*), 187
topology_virtual_site_index() (*openforcefield.topology.TopologyVirtualSite property*), 229
topology_virtual_sites() (*openforcefield.topology.Topology property*), 140
TopologyAtom (*class in openforcefield.topology*), 215
TopologyBond (*class in openforcefield.topology*), 221
TopologyMolecule (*class in openforcefield.topology*), 147
TopologyVirtualParticle (*class in openforcefield.topology*), 182
TopologyVirtualSite (*class in openforcefield.topology*), 226
torsions() (*openforcefield.topology.FrozenMolecule property*), 83
torsions() (*openforcefield.topology.Molecule property*), 127
total_charge() (*openforcefield.topology.FrozenMolecule property*), 84
total_charge() (*openforcefield.topology.Molecule property*), 127
TrivalentLonePairVirtualSite (*class in openforcefield.topology*), 209
type() (*openforcefield.topology.BondChargeVirtualSite property*), 194
type() (*openforcefield.topology.DivalentLonePairVirtualSite property*), 208
type() (*openforcefield.topology.MonovalentLonePairVirtualSite property*), 201
type() (*openforcefield.topology.TopologyVirtualParticle property*), 188
type() (*openforcefield.topology.TopologyVirtualSite property*), 230
type() (*openforcefield.topology.TrivalentLonePairVirtualSite property*), 215
type() (*openforcefield.topology.VirtualSite property*), 174

U

update() (*openforcefield.topology.ImproperDict method*), 237
update() (*openforcefield.topology.ValenceDict method*), 234

V

ValenceDict (*class in openforcefield.topology*), 233
validate() (*openforcefield.typing.chemistry.ChemicalEnvironment method*), 238
validate_smirks() (*openforcefield.typing.chemistry.ChemicalEnvironment class method*), 238
values() (*openforcefield.topology.ImproperDict method*), 237
values() (*openforcefield.topology.ValenceDict method*), 235
vdWHandler (*class in openforcefield.typing.engines.smirnoff.parameters*),

350
vdWHandler.vdWType (*class in openforcefield.typing.engines.smirnoff.parameters*), 369
virtual_particle_start_topology_index() (*openforcefield.topology.TopologyMolecule property*), 151
virtual_site() (*openforcefield.topology.Topology method*), 144
virtual_site() (*openforcefield.topology.TopologyMolecule method*), 152
virtual_site() (*openforcefield.topology.TopologyVirtualParticle property*), 188
virtual_site() (*openforcefield.topology.TopologyVirtualSite property*), 229
virtual_site_particle_index() (*openforcefield.topology.VirtualParticle property*), 178
virtual_site_start_topology_index() (*openforcefield.topology.TopologyMolecule property*), 153
virtual_sites() (*openforcefield.topology.Atom property*), 163
virtual_sites() (*openforcefield.topology.FrozenMolecule property*), 83
virtual_sites() (*openforcefield.topology.Molecule property*), 127
virtual_sites() (*openforcefield.topology.TopologyMolecule property*), 152
VirtualParticle (*class in openforcefield.topology*), 177
VirtualSite (*class in openforcefield.topology*), 171
visualize() (*openforcefield.topology.Molecule method*), 130

X

XMLParameterIOHandler (*class in openforcefield.typing.engines.smirnoff.io*), 455