openforcefield Documentation

Release 0.2.0

Open Force Field Consortium

Apr 07, 2019

Contents

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

CHAPTER 1

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.continuum.io/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

```
$ wget https://repo.continuum.io/miniconda/Miniconda2-latest-MacOSX-x86_64.sh
$ bash ./Miniconda2-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.

1.2 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 0.2.0

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 0.2 SMIRNOFF 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

1.2.2 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 v0.2

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

1.3.1 Authors and acknowledgments

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

Primary contributors include:

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

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

1.3.2 Representations and encodings

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

XML representation

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

- By convention, XML-encoded SMIRNOFF force fields use an `.offxml` extension if written to a file to prevent confusion with other file formats.
- In XML, numeric quantities appear as strings, like `"1"` or `"2.3"`.
- Integers should always be written without a decimal point, such as `"1"`, `"9"`.
- Non-integral numbers, such as parameter values, should be written with a decimal point, such as `"1.23"`, `"2."`.
- In XML, certain special characters that occur in valid SMARTS/SMIRKS patterns (such as ampersand symbols `&`) must be specially encoded. See this list of XML and HTML character entity references for more details.

Future representations: JSON, MessagePack, YAML, and TOML

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

1.3.3 Reference implementation

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

1.3.4 Support for molecular simulation packages

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

1.3.5 Basic structure

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

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

The enclosing <SMIRNOFF> tag

A SMIRNOFF forcefield XML specification always is enclosed in a `<SMIRNOFF>` tag, with certain required attributes provided.

```
<SMIRNOFF version="0.2" 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="0.2" 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.

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).

Todo: Should we have a separate *<Metadata>* or *<Provenance>* section that users can add whatever they want to? This would minimize the potential for accidentally colliding with other tags we add in the future.

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`, `TorsionForce`, etc., as discussed in more detail below); these specify units used for the different constants provided for individual force terms.

```
<Angles angle_unit="degrees" k_unit="kilocalories_per_mole/radian**2">
   ...
</Angles>
```

which introduces following `Angle` terms which will use units of degrees for the angle and kilocalories per mole per square radian for the force constant.

Specifying parameters

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

```
<Angles angle_unit="degrees" k_unit="kilocalories_per_mole/radian**2">
   <Angle smirks="[a,A:1]-[#6X4:2]-[a,A:3]" angle="109.50" k="100.0"/>
   <Angle smirks="[#1:1]-[#6X4:2]-[#1:3]" angle="109.50" k="70.0"/>
</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 as given above by `angle_unit` and `k_unit`, respectively).

Note: The XML parser ignores attributes in the XML that it does not know how to process. For example, providing an *<Angle>* tag that also specifies a second force constant *k2* will simply result in *k2* being silently ignored.

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 `.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. For example, for the `angle` (equilibrium angle) and `k` (force constant) parameters in the `<Angles>` example block above, the `angle_unit` and `k_unit` top-level attributes specify the corresponding units:

```
<Angles angle_unit="degrees" k_unit="kilocalories_per_mole/radian**2">
...
</Angles>
```

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

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

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

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 charge_unit="elementary_charge">
   <!-- 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" charge2="0.2719" charge3="0.0337" charge4="0.0823"␣
↪charge5="-0.1825" charge6="0.0603" charge7="0.0603" charge8="0.0603" charge9="0.5973" charge10="-0.
↪5679">
   ...
</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 charge_unit="elementary_charge">
   <!-- TIP3P water oxygen with charge override -->
   <LibraryCharge name="TIP3P" smirks="[#1:1]-[#8X2H2+0:2]-[#1:3]" charge1="+0.417" charge2="-0.834"␣
↪charge3="+0.417"/>
</LibraryCharges>
```

`<ChargeIncrementModel>`: Small molecule and fragment charges

Warning: This functionality is not yet implemented and will appear in a future version of the toolkit. This area of the SMIRNOFF spec is under further consideration. Please see [Issue 208 on the Open Force Field Toolkit issue tracker](https://github.com/openforcefield/openforcefield/issues/208).

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 number_of_conformers="10" quantum_chemical_method="AM1" partial_charge_method="CM2
↪" increment_unit="elementary_charge">
  <!-- A fractional charge can be moved along a single bond -->
  <ChargeIncrement smirks="[#6X4:1]-[#6X3a:2]" chargeincrement1="-0.0073" chargeincrement2="+0.0073"/>
  <ChargeIncrement smirks="[#6X4:1]-[#6X3a:2]-[#7]" chargeincrement1="+0.0943" chargeincrement2="-0.0943
↪"/>
  <ChargeIncrement smirks="[#6X4:1]-[#8:2]" chargeincrement1="-0.0718" chargeincrement2="+0.0718"/>
  <!--- Alternatively, factional charges can be redistributed among any number of bonded atoms -->
```

(continues on next page)

(continued from previous page)

```
  <ChargeIncrement smirks="[N:1](H:2)(H:3)" chargeincrement1="+0.02" chargeincrement2="-0.01"␣
↪chargeincrement3="-0.01"/>
</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: `"10"`) is used to specify how many conformers will be generated for the molecule (or capped fragment) prior to charging.
- The `quantum_chemical_method` attribute (default: `"AM1"`) is used to specify the quantum chemical method applied to the molecule or capped fragment.
- The `partial_charge_method` attribute (default: `"CM2"`) is used to specify how uncorrected partial charges are to be generated from the quantum chemical wavefunction. 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` attribute specify how much the charge on the associated tagged atom index (replacing #) should be modified. The sum of charge increments should equal zero.

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

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

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. This decoupling of how parameters are assigned for each term provides a great deal of flexibility in composing new force fields while allowing a minimal number of parameters to be used to achieve accurate modeling of intramolecular forces.

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

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

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

<vdW>

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

```
<vdW potential="Lennard-Jones-12-6" combining_rules="Lorentz-Berthelot" scale12="0.0" scale13="0.0"␣
↪scale14="0.5" scale15="1.0" sigma_unit="angstroms" epsilon_unit="kilocalories_per_mole" switch_width=
↪"8.0" switch_width_unit="angstrom" cutoff="9.0" cutoff_unit="angstroms" long_range_dispersion=
↪"isotropic">
   <Atom smirks="[#1:1]" sigma="1.4870" epsilon="0.0157"/>
   <Atom smirks="[#1:1]-[#6]" sigma="1.4870" epsilon="0.0157"/>
   ...
</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", 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. Note that, if rmin_half is specified instead of sigma, rmin_half_unit should be specified; both can be used in the same block if desired.

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

(continues on next page)

(continued from previous page)

```
   ...
</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 potential="Lennard-Jones-12-6" combining_rules="Lorentz-Berthelot" scale12="0.0" scale13="0.0"␣
↪scale14="0.5" sigma_unit="angstroms" epsilon_unit="kilocalories_per_mole">
   <AtomPair smirks1="[#1:1]" smirks2="[#6:2]" sigma="1.4870" epsilon="0.0157"/>
   ...
</vdW>
```

<Electrostatics>

Electrostatic interactions are specified via the <Electrostatics> tag.

```
<Electrostatics 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.

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

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

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

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

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

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

Fractional bond orders (EXPERIMENTAL)

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

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

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

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 potential="harmonic" length_unit="angstroms" k_unit="kilocalories_per_mole/angstrom**2"␣
↪fractional_bondorder_method="Wiberg" fractional_bondorder_interpolation="linear">
    <Bond smirks="[#6X3:1]!#[#6X3:2]" k_bondorder1="820.0" k_bondorder2="1098" length_bondorder1="1.45"␣
↪length_bondorder2="1.35"/>
    ...
```

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

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

<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 potential="harmonic" angle_unit="degrees" k_unit="kilocalories_per_mole/radian**2">
   <Angle smirks="[a,A:1]-[#6X4:2]-[a,A:3]" angle="109.50" k="100.0"/>
   <Angle smirks="[#1:1]-[#6X4:2]-[#1:3]" angle="109.50" k="70.0"/>
   ...
</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.

<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 potential="charmm" phase_unit="degrees" k_unit="kilocalories_per_mole">
   <Proper smirks="[a,A:1]-[#6X4:2]-[#6X4:3]-[a,A:4]" idivf1="9" periodicity1="3" phase1="0.0" k1="1.40
↪"/>
   <Proper smirks="[#6X4:1]-[#6X4:2]-[#8X2:3]-[#6X4:4]" idivf1="1" periodicity1="3" phase1="0.0" k1="0.
↪383" idivf2="1" periodicity2="2" phase2="180.0" k2="0.1"/>
   ...
</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 potential="charmm" phase_unit="degrees" k_unit="kilocalories_per_mole" default_idivf=
↪"auto">
   <Proper smirks="[a,A:1]-[#6X4:2]-[#6X4:3]-[a,A:4]" periodicity1="3" phase1="0.0" k1="1.40"/>
   ...
</ProperTorsions>
```

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`.

<ImproperTorsions>

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

```
<ImproperTorsions potential="charmm" phase_unit="degrees" k_unit="kilocalories_per_mole">
    <Improper smirks="[*:1]~[#6X3:2](=[#7X2,#7X3+1:3])~[#7:4]" k1="10.5" periodicity1="2" phase1="180."/
↪>
    ...
</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.

<GBSA>

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

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

```
<GBSA gb_model="OBC1" solvent_dielectric="78.5" solute_dielectric="1" radius_unit="nanometers" sa_model=
↪"ACE" surface_area_penalty="5.4" surface_area_penalty_unit="calories/mole/angstroms**2" solvent_
↪radius="1.4" solvent_radius_unit="angstroms">
  <Atom smirks="[#1:1]" radius="0.12" scale="0.85"/>
  <Atom smirks="[#1:1]~[#6]" radius="0.13" scale="0.85"/>
  <Atom smirks="[#1:1]~[#8]" radius="0.08" scale="0.85"/>
  <Atom smirks="[#1:1]~[#16]" radius="0.08" scale="0.85"/>
  <Atom smirks="[#6:1]" radius="0.22" scale="0.72"/>
  <Atom smirks="[#7:1]" radius="0.155" scale="0.79"/>
  <Atom smirks="[#8:1]" radius="0.15" scale="0.85"/>
  <Atom smirks="[#9:1]" radius="0.15" scale="0.88"/>
  <Atom smirks="[#14:1]" radius="0.21" scale="0.8"/>
  <Atom smirks="[#15:1]" radius="0.185" scale="0.86"/>
  <Atom smirks="[#16:1]" radius="0.18" scale="0.96"/>
  <Atom smirks="[#17:1]" radius="0.17" 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's "simtk.openmm.app"* <http://docs.openmm.org/latest/userguide/application.html#amber-implicit-solvent>'_:

- *HCT*: [Hawkins-Cramer-Truhlar](http://docs.openmm.org/latest/userguide/zbibliography.html#hawkins1995) (corresponding to `igb=1` in AMBER): requires `[radius, scale]`
- *OBC1*: [Onufriev-Bashford-Case](http://docs.openmm.org/latest/userguide/zbibliography.html#onufriev2004) using the GB(OBC)I parameters (corresponding to `igb=2` in AMBER): requires `[radius, scale]`
- *OBC2*: [Onufriev-Bashford-Case](http://docs.openmm.org/latest/userguide/zbibliography.html#onufriev2004) using the GB(OBC)II parameters (corresponding to `igb=5` in AMBER): requires `[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. Units are for these per-particle parameters (such as `radius_units`) specified in the `<GBSA>` tag.

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`)

<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>
  <!-- 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 distance_unit="angstroms">
  <!-- constrain water O-H bond to equilibrium bond length (overrides earlier constraint) -->
  <Constraint smirks="[#1:1]-[#8X2H2:2]-[#1]" distance="0.9572"/>
  <!-- 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"/>
</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 distance_unit="angstroms">
  <!-- 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"/>
  <Constraint smirks="[#1:1]-[#8X2H2]-[#1:2]" distance="1.8532"/>
</Constraints>
```

1.3.10 Advanced features

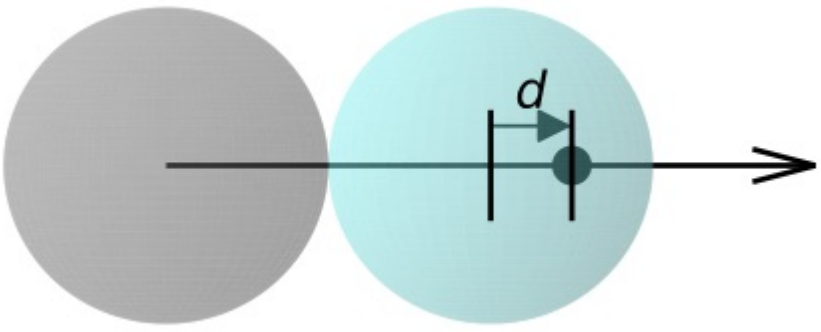
Standard usage is expected to rely primarily on the features documented above and potentially new features. However, some advanced features are also currently supported.

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

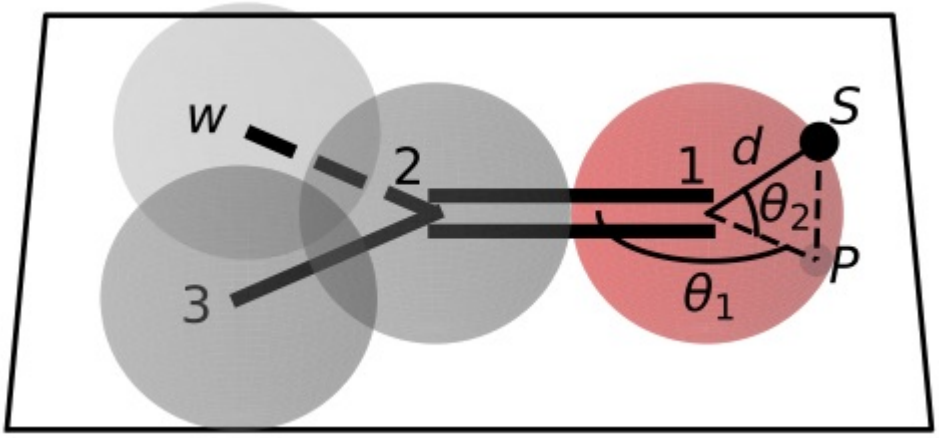
Warning: This functionality is not yet implemented and will appear in a future version of the toolkit

We have implemented 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. Currently we support the following different types or geometries of off-center charges (as diagrammed below):

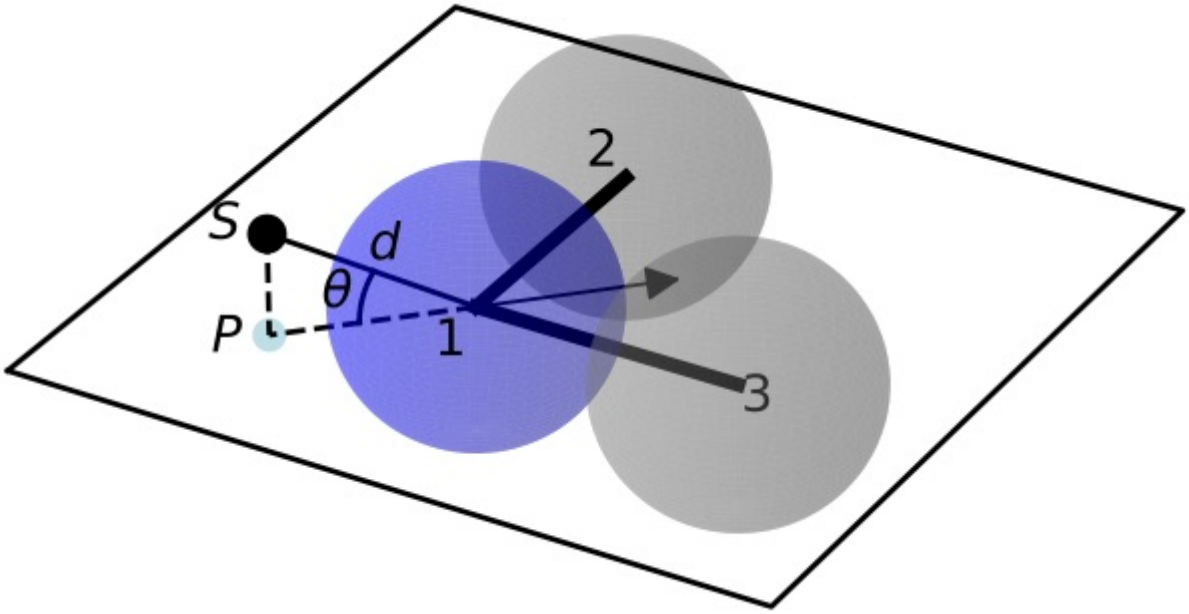
- `BondCharge`: This supports placement of a virtual site `S` along a vector between two specified atoms, e.g. to allow for a sigma hole for halogens or similar contexts. With positive values of the distance, the virtual site lies outside the first indexed atom (green in this image).



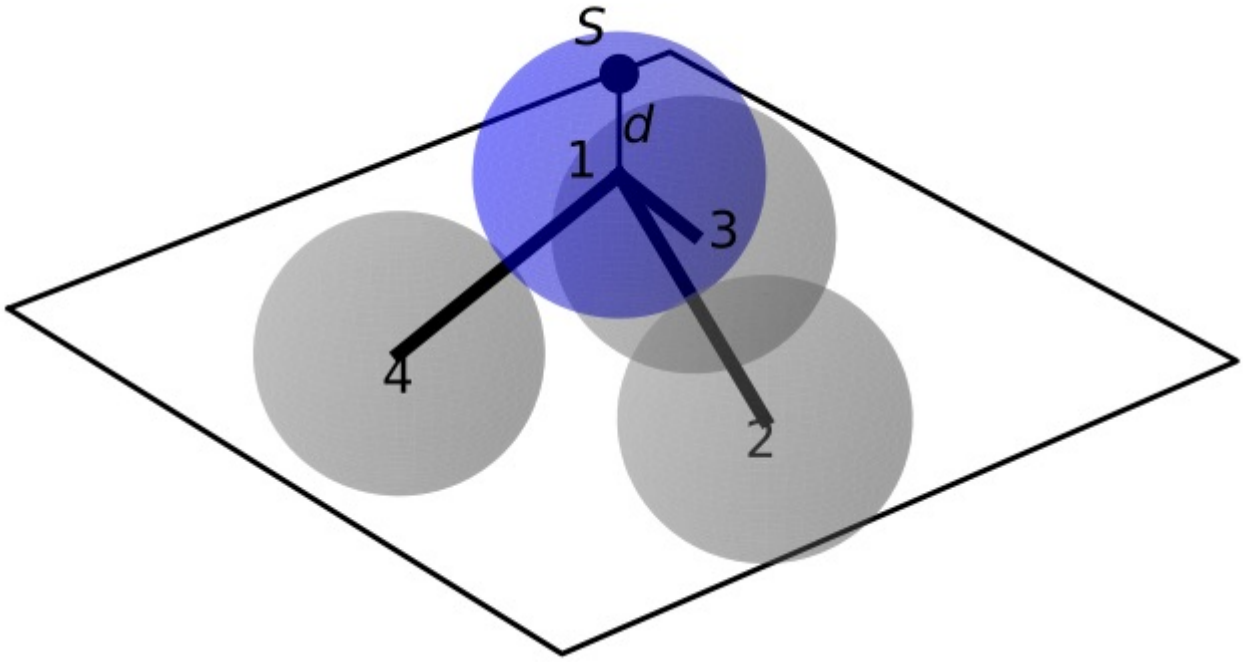
- `MonovalentLonePair`: This is originally intended for situations like a carbonyl, and allows placement of a virtual site `S` at a specified distance `d`, `inPlaneAngle` (theta 1 in the diagram), and `outOfPlaneAngle` (theta 2 in the diagram) relative to a central atom and two connected atoms.



- `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 `chargeincrement#` parameter, e.g., if `chargeincrement1=+0.1` then the virtual site will receive a charge of -0.1 and the atom labeled `1` will have its charge adjusted upwards by +0.1. N may index any indexed atom. Increments which are left unspecified default to zero. Additionally, each virtual site can bear Lennard-Jones parameters, specified by `sigma` and `epsilon` or `rmin_half` and `epsilon`. If unspecified these also default to zero.

In the SMIRNOFF format, these are encoded as:

```
<VirtualSites distanceUnits="angstroms" angleUnits="degrees" sigma_unit="angstroms" epsilon_unit=
↪"kilocalories_per_mole">
    <!-- 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" chargeincrement1="+0.2"␣
↪chargeincrement2="+0.1" sigma="0.1" epsilon="0.05" />
    <!-- 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" chargeincrement1="+0.1"␣
↪chargeincrement2="+0.1" chargeincrement3="+0.05" sigma="0.1" epsilon="0.05" />
    <!-- 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" outOfPlaneAngle="0
↪" inPlaneAngle="120" chargeincrement1="+0.2" />
    <!-- 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 lies along the angle
↪'s bisector. -->
```

(continues on next page)

(continued from previous page)

```
    <VirtualSite type="DivalentLonePair" smirks="[*:2]~[#7X2:1]~[*:3]" distance="0.30" OfPlaneAngle="0.0
↪" chargeincrement1="+0.1" />
    <!-- trivalent nitrogen lone pair -->
    <!-- charge sites lie above and below the nitrogen at specified distances from the nitrogen, along␣
↪the vector perpendicular to the plane of (2,3,4) that passes through the nitrogen. If the nitrogen is␣
↪co-planar with the connected atom, charge sites are simply above and below the plane-->
    <!-- Positive and negative values refer to above or below the nitrogen as measured relative to the␣
↪plane of (2,3,4), i.e. below the nitrogen means nearer the 2,3,4 plane unless they are co-planar -->
    <VirtualSite type="TrivalentLonePair" smirks="[*:2]~[#7X3:1](~[*:4])~[*:3]" distance="0.30"␣
↪chargeincrement1="+0.1"/>
    <VirtualSite type="TrivalentLonePair" smirks="[*:2]~[#7X3:1](~[*:4])~[*:3]" distance="-0.30"␣
↪chargeincrement1="+0.1"/>
</VirtualSites>
```

Aromaticity models

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

Additional plans for future development

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

1.3.11 The `openforcefield` reference implementation

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

Examples

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

Parameterizing a system

Consider parameterizing a simple system containing a the drug imatinib.

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

Using SMIRNOFF small molecule forcefields with traditional biopolymer force fields

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

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

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

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

```
<Bond smirks="[#6X3:1]-[#6X3:2]" id="b5" k="820.0" length="1.45"/>
```

Some functionality in `ForceField`, such as `ForceField.labelMolecules`, 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

0.2

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

- Aromaticity model now defaults to `OEAroModel_MDL`, and aromaticity model names drop OpenEye-specific prefixes
- Top-level tags are now required to specify units for any unit-bearing quantities to avoid the potential for mistakes from implied units.
- Potential energy component definitions were renamed to be more general:
  - `<NonbondedForce>` was renamed to `<vdW>`
  - `<HarmonicBondForce>` was renamed to `<Bonds>`
  - `<HarmonicAngleForce>` was renamed to `<Angles>`
  - `<BondChargeCorrections>` was renamed to `<ChargeIncrementModel>` and generalized to accommodate an arbitrary number of tagged atoms
  - `<GBSAForce>` was renamed to `<GBSA>`
- `<PeriodicTorsionForce>` was split into `<ProperTorsions>` and `<ImproperTorsions>`
- `<vdW>` now specifies 1-2, 1-3, 1-4, and 1-5 scaling factors via `scale12` (default: 0), `scale13` (default: 0), `scale14` (default: 0.5), and `scale15` (default 1.0) attributes. It also specifies the long-range vdW method to use, currently supporting `cutoff` (default) and `PME`. Coulomb scaling parameters have been removed from `StericsForce`.
- Added the `<Electrostatics>` tag to separately specify 1-2, 1-3, 1-4, and 1-5 scaling factors for electrostatics, as well as the method used to compute electrostatics (`PME`, `reaction-field`, `Coulomb`) since this has a huge effect on the energetics of the system.
- Made it clear that `<Constraint>` entries do not have to be between bonded atoms.
- `<VirtualSites>` has been added, and the specification of charge increments harmonized with `<ChargeIncrementModel>`
- The `potential` attribute was added to most forces to allow flexibility in extending forces to additional functional forms (or algebraic expressions) in the future. `potential` defaults to the current recommended scheme if omitted.
- `<GBSA>` now has defaults specified for `gb_method` and `sa_method`
- Changes to how fractional bond orders are handled:
  - Use of fractional bond order is now are specified at the force tag level, rather than the root level
  - The fractional bond order method is specified via the `fractional_bondorder_method` attribute
  - The fractional bond order interpolation scheme is specified via the `fractional_bondorder_interpolation`
- Section heading names were cleaned up.
- Example was updated to reflect use of the new `openforcefield.topology.Topology` class
- Eliminated "Requirements" section, since it specified requirements for the software, rather than described an aspect of the SMIRNOFF specification
- Fractional bond orders are described in `<Bonds>`, since they currently only apply to this term.

0.1

Initial draft specification.

1.4 Examples using SMIRNOFF with the toolkit

The following examples are available in the openforcefield toolkit repository:

1.4.1 Index of provided examples

- SMIRNOFF_simulation - simulation of a molecule in the gas phase with the SMIRNOFF forcefield 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 Forcefield Toolkit, which can be simulated natively with OpenMM, into AMBER prmtop/inpcrd and GROMACS top/gro input files through the ParmEd library.
- 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)

1.5 Developing for the toolkit

1.5.1 Style guide

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

1.5.2 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.

1.5.3 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.

1.6 Frequently asked questions (FAQ)

1.6.1 Input files for applying SMIRNOFF parameters

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

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

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

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

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

In a word, “no”.

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

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

1.6.3 What about starting from a PDB file?

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

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

- Use a cheminformatics toolkit (see above) to infer bond orders

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

1.6.4 What do you recommend as a starting point?

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

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

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

CHAPTER 2

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.

openforcefield.topology.FrozenMolecule

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

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

Todo: What other API calls would be useful for supporting biopolymers as small molecules? Perhaps iterating over chains and residues?

Examples

Create a molecule from a sdf file

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

Convert to OpenEye OEMol object

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

Create a molecule from an OpenEye molecule

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

Convert to RDKit Mol object

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

Create a molecule from an RDKit molecule

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

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

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

Create a molecule from SMILES

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

Warning: This API is experimental and subject to change.

Attributes

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

atoms Iterate over all Atom objects.

bonds Iterate over all Bond objects.

conformers Iterate over all conformers in this molecule.

impropers Iterate over all proper torsions in the molecule

n_angles int: number of angles in the Molecule.

n_atoms The number of Atom objects.

n_bonds The number of Bond objects.

n_conformers Iterate over all Atom objects.

n_impropers int: number of improper torsions in the Molecule.

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

n_propers int: number of proper torsions in the Molecule.

n_virtual_sites The number of VirtualSite objects.

name The name (or title) of the molecule

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

particles Iterate over all Particle objects.

propers Iterate over all proper torsions in the molecule

properties The properties dictionary of the molecule

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

total_charge Return the total charge on the molecule

virtual_sites Iterate over all VirtualSite objects.

Methods

chemical_environment_matches(query[, ...])	Retrieve all matches for a given chemical environment query.
compute_partial_charges([toolkit_registry])	**Warning! Not Implemented!** Calculate partial atomic charges for this molecule using an underlying toolkit
compute_partial_charges_am1bcc([...])	Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit
compute_wiberg_bond_orders([charge_model, ...])	Calculate wiberg bond orders for this molecule using an underlying toolkit
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(filename[, file_format, ...])	Create one or more molecules from a file
from_iupac(iupac_name, **kwargs)	Generate a molecule from IUPAC or common name
from_json(serialized)	Instantiate an object from a JSON serialized representation.
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_pickle(serialized)	Instantiate an object from a pickle serialized representation.
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 openforcefield Topology containing a single Molecule object.
from_xml(serialized)	Instantiate an object from an XML serialized representation.

Continued on next page

Table 2 – continued from previous page

from_yaml(serialized)	Instantiate from a YAML serialized representation.
generate_conformers([toolkit_registry, ...])	Generate conformers for this molecule using an underlying toolkit
get_fractional_bond_orders([method, ...])	Get fractional bond orders.
is_isomorphic(other[, ...])	Determines whether the molecules are isomorphic by comparing their graphs.
to_bson()	Return a BSON serialized representation.
to_dict()	Return a dictionary representation of the molecule.
to_file(outfile, outfile_format[, ...])	Write the current molecule to a file or file-like object
to_iupac()	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_rdkit([aromaticity_model])	Create an RDKit molecule
to_smiles([toolkit_registry])	Return a canonical isomeric SMILES representation of the current molecule
to_toml()	Return a TOML serialized representation.
to_topology()	Return an openforcefield Topology representation containing one copy of this molecule
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

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

Create a new FrozenMolecule object

Todo:

- If a filename or file-like object is specified but the file contains more than one molecule, what is the proper behavior?

Read just the first molecule, or raise an exception if more than one molecule is found?

- Should we also support SMILES strings or IUPAC names for `other`?

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_filename
>>> sdf_filepath = get_data_filename('molecules/ethanol.sdf')
>>> molecule = FrozenMolecule(sdf_filepath)
>>> molecule = FrozenMolecule(open(sdf_filepath, 'r'), file_format='sdf')
```

```
>>> import gzip
>>> mol2_gz_filepath = get_data_filename('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
chemical_environment_matches(query[, ...])	Retrieve all matches for a given chemical environment query.
compute_partial_charges([toolkit_registry])	**Warning! Not Implemented!** Calculate partial atomic charges for this molecule using an underlying toolkit
compute_partial_charges_am1bcc([...])	Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit
compute_wiberg_bond_orders([charge_model, ...])	Calculate wiberg bond orders for this molecule using an underlying toolkit
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(filename[, file_format, ...])	Create one or more molecules from a file
from_iupac(iupac_name, **kwargs)	Generate a molecule from IUPAC or common name
from_json(serialized)	Instantiate an object from a JSON serialized representation.
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_pickle(serialized)	Instantiate an object from a pickle serialized representation.
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 openforcefield Topology containing a single Molecule object.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
generate_conformers([toolkit_registry, ...])	Generate conformers for this molecule using an underlying toolkit
get_fractional_bond_orders([method, ...])	Get fractional bond orders.
is_isomorphic(other[, ...])	Determines whether the molecules are isomorphic by comparing their graphs.
to_bson()	Return a BSON serialized representation.
to_dict()	Return a dictionary representation of the molecule.
to_file(outfile, outfile_format[, ...])	Write the current molecule to a file or file-like object
to_iupac()	Generate IUPAC name from Molecule

Continued on next page

Table 3 – continued from previous page

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_rdkit([aromaticity_model])	Create an RDKit molecule
to_smiles([toolkit_registry])	Return a canonical isomeric SMILES representation of the current molecule
to_toml()	Return a TOML serialized representation.
to_topology()	Return an openforcefield Topology representation containing one copy of this molecule
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

Attributes

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

to_dict()

Return a dictionary representation of the molecule.

Todo:

- Document the representation standard.
- How do we do version control with this standard?

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(*toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)
Return a canonical isomeric SMILES representation of the current molecule

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

Parameters

toolkit_registry [openforcefield.utils.toolkits.ToolRegistry 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_filename
>>> sdf_filepath = get_data_filename('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
>>> smiles = molecule.to_smiles()
```

static from_smiles(*smiles*, *hydrogens_are_explicit=False*, *toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)
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.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=None] `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion

Returns

molecule [openforcefield.topology.Molecule]

Examples

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

is_isomorphic(*other*, *compare_atom_stereochemistry=True*, *compare_bond_stereochemistry=True*)

Determines whether the molecules are isomorphic by comparing their graphs.

Parameters

other [an openforcefield.topology.molecule.FrozenMolecule] The molecule to test for isomorphism.

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

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

Returns

molecules_are_isomorphic [bool]

generate_conformers(*toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*, *num_conformers=10*, *clear_existing=True*)

Generate conformers for this molecule using an underlying toolkit

Parameters

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

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

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

Raises

InvalidToolkitError If an invalid object is passed as the toolkit_registry parameter

Examples

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

compute_partial_charges_am1bcc(*toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)

Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit

Parameters

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

Raises

InvalidToolkitError 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()
```

compute_partial_charges(*toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)

Warning! Not Implemented! Calculate partial atomic charges for this molecule using an underlying toolkit

Parameters

quantum_chemical_method [string, default='AM1-BCC'] The quantum chemical method to use for partial charge calculation.

partial_charge_method [string, default='None'] The partial charge calculation method to use for partial charge calculation.

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

Raises

InvalidToolkitError If an invalid object is passed as the toolkit_registry parameter

Examples

molecule = Molecule.from_smiles('CCCCCC') molecule.generate_conformers() molecule.compute_partial_charges()

compute_wiberg_bond_orders(*charge_model=None*, *toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)

Calculate wiberg bond orders for this molecule using an underlying toolkit

Parameters

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

charge_model [string, optional] The charge model to use for partial charge calculation

Examples

———

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

>>> molecule.generate_conformers()

>>> molecule.compute_wiberg_bond_orders()

Raises

InvalidToolkitError If an invalid object is passed as the toolkit_registry parameter

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.

Todo:

- Do we need a `from_networkx()` method? If so, what would the Graph be required to provide?
- Should edges be labeled with discrete bond types in some aromaticity model?
- Should edges be labeled with fractional bond order if a method is specified?
- Should we add other per-atom and per-bond properties (e.g. partial charges) if present?
- Can this encode bond/atom chirality?

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()
```

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]] The partial charges on this Molecule's atoms.

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

n_atoms
The number of Atom objects.

n_virtual_sites
The number of VirtualSite objects.

n_bonds
The number of Bond objects.

n_angles
int: number of angles in the Molecule.

n_propers
int: number of proper torsions in the Molecule.

n_impropers
int: number of improper torsions in the Molecule.

particles
Iterate over all Particle objects.

atoms
Iterate over all Atom objects.

conformers
Iterate over all conformers in this molecule.

n_conformers
Iterate over all Atom objects.

virtual_sites
Iterate over all VirtualSite objects.

bonds
Iterate over all Bond objects.

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

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]

propers
Iterate over all proper torsions in the molecule

Todo:

- Do we need to return a `Torsion` object that collects information about fractional bond orders?

---

impropers
Iterate over all proper torsions in the molecule

Todo:

- Do we need to return a `Torsion` object that collects information about fractional bond orders?

---

total_charge
Return the total charge on the molecule

name
The name (or title) of the molecule

properties
The properties dictionary of the molecule

chemical_environment_matches(*query*, *toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)
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.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=GLOBAL_TOOLKIT_REGISTRY] `ToolkitRegistry` or `ToolkitWrapper` to use for chemical environment matches

Returns

matches [list of Atom tuples] A list of all matching Atom tuples

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]')
```

Todo:

- Do we want to generalize `query` to allow other kinds of queries, such as mdtraj DSL, pymol selections, atom index slices, etc? We could call it `topology.matches(query)` instead of `chemical_environment_matches`

---

classmethod from_iupac(*iupac_name*, ***kwargs*)

Generate a molecule from IUPAC or common name

Parameters

iupac_name [str] IUPAC name of molecule to be generated

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 a common 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()

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_filename
>>> sdf_filepath = get_data_filename('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
>>> iupac_name = molecule.to_iupac()
```

static from_topology(*topology*)

Return a Molecule representation of an openforcefield 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)  # doctest: +SKIP
```

to_topology()

Return an openforcefield 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()
```

static from_file(*filename*, *file_format=None*, *toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*, *allow_undefined_stereo=False*)

Create one or more molecules from a file

Todo:

- Extend this to also include some form of .offmol Open Force Field Molecule format?
- Generalize this to also include file-like objects?

Parameters

filename [str or file-like object] The name of 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.ToolRegistry 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_mol2file
>>> mol2_filename = get_monomer_mol2file('cyclohexane')
>>> molecule = Molecule.from_file(mol2_filename)
```

to_file(*outfile*, *outfile_format*, *toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)

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

Parameters

outfile [str or file-like object] A file-like object or the filename of the file to be written to

outfile_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.ToolRegistry 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 outfile_format is not supported by one of the installed cheminformatics toolkits

Examples

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> molecule.to_file('imatinib.mol2', outfile_format='mol2')  # doctest: +SKIP
>>> molecule.to_file('imatinib.sdf', outfile_format='sdf')  # doctest: +SKIP
>>> molecule.to_file('imatinib.pdb', outfile_format='pdb')  # doctest: +SKIP
```

static 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.Molecule] An openforcefield molecule

Examples

Create a molecule from an RDKit molecule

```
>>> from rdkit import Chem
>>> from openforcefield.tests.utils import get_data_filename
>>> rdmol = Chem.MolFromMolFile(get_data_filename('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_filename
>>> sdf_filepath = get_data_filename('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
>>> rdmol = molecule.to_rdkit()
```

static 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 openforcefield molecule

Examples

Create a Molecule from an OpenEye OEMol

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

to_openeye(*aromaticity_model='OEAroModel_MDL'*)

Create an OpenEye molecule

Requires the OpenEye toolkit to be installed.

Todo:

- Use stored conformer positions instead of an argument.
- Should the aromaticity model be specified in some other way?

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_fractional_bond_orders(*method='Wiberg'*, *toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)

Get fractional bond orders.

method [str, optional, default='Wiberg'] The name of the charge method to use. Options are: * 'Wiberg' : Wiberg bond order

toolkit_registry [openforcefield.utils.toolkits ToolkitRegistry] The toolkit registry to use for molecule operations

Examples

Get fractional Wiberg bond orders

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> molecule.generate_conformers()
>>> fractional_bond_orders = molecule.get_fractional_bond_orders(method='Wiberg')
```

Todo:

- Is it OK that the `Molecule` object does not store geometry, but will create it using `openeye.omega` or `rdkit`?
- Should this method assign fractional bond orders to the `Bond``s in the molecule, a separate ``bond_orders` molecule property, or just return the array of bond orders?
- How do we add enough flexibility to specify the toolkit and optional parameters, such as: `oequacpac.OEAssignPartialCharges(charged_copy, getattr(oequacpac, 'OECharges_AM1BCCSym'), False, False)`
- Generalize to allow user to specify both QM method and bond order computation approach (e.g. `AM1` and `Wiberg`)

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.Molecule

class openforcefield.topology.**Molecule**(**args*, ***kwargs*)
Mutable chemical representation of a molecule, such as a small molecule or biopolymer.

> **Todo:** What other API calls would be useful for supporting biopolymers as small molecules? Perhaps iterating over chains and residues?

Examples

Create a molecule from an sdf file

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

Convert to OpenEye OEMol object

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

Create a molecule from an OpenEye molecule

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

Convert to RDKit Mol object

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

Create a molecule from an RDKit molecule

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

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

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

Create a molecule from SMILES

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

Warning: This API is experimental and subject to change.

Attributes

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

atoms Iterate over all Atom objects.

bonds Iterate over all Bond objects.

conformers Iterate over all conformers in this molecule.

impropers Iterate over all proper torsions in the molecule

n_angles int: number of angles in the Molecule.

n_atoms The number of Atom objects.

n_bonds The number of Bond objects.

n_conformers Iterate over all Atom objects.

n_impropers int: number of improper torsions in the Molecule.

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

n_propers int: number of proper torsions in the Molecule.

n_virtual_sites The number of VirtualSite objects.

name The name (or title) of the molecule

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

particles Iterate over all Particle objects.

propers Iterate over all proper torsions in the molecule

properties The properties dictionary of the molecule

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

total_charge Return the total charge on the molecule

virtual_sites Iterate over all VirtualSite objects.

Methods

add_atom(atomic_number, formal_charge, ...)	Add an atom
add_bond(atom1, atom2, bond_order, is_aromatic)	Add a bond between two specified atom indices
add_bond_charge_virtual_site(atoms, distance)	Create a bond charge-type virtual site, in which the location of the charge is specified by the position of two atoms.
add_conformer(coordinates)	# TODO: Should this not be public? Adds a conformer 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.
chemical_environment_matches(query[, ...])	Retrieve all matches for a given chemical environment query.
compute_partial_charges([toolkit_registry])	**Warning! Not Implemented!** Calculate partial atomic charges for this molecule using an underlying toolkit
compute_partial_charges_am1bcc([...])	Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit
compute_wiberg_bond_orders([charge_model, ...])	Calculate wiberg bond orders for this molecule using an underlying toolkit
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(filename[, file_format, ...])	Create one or more molecules from a file
from_iupac(iupac_name, **kwargs)	Generate a molecule from IUPAC or common name

Continued on next page

Table 5 – 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_openeye(oemol[, allow_undefined_stereo])	Create a Molecule from an OpenEye molecule.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
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 openforcefield Topology containing a single Molecule object.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
generate_conformers([toolkit_registry, ...])	Generate conformers for this molecule using an underlying toolkit
get_fractional_bond_orders([method, ...])	Get fractional bond orders.
is_isomorphic(other[, ...])	Determines whether the molecules are isomorphic by comparing their graphs.
to_bson()	Return a BSON serialized representation.
to_dict()	Return a dictionary representation of the molecule.
to_file(outfile, outfile_format[, ...])	Write the current molecule to a file or file-like object
to_iupac()	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_rdkit([aromaticity_model])	Create an RDKit molecule
to_smiles([toolkit_registry])	Return a canonical isomeric SMILES representation of the current molecule
to_toml()	Return a TOML serialized representation.
to_topology()	Return an openforcefield Topology representation containing one copy of this molecule
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

__init__(**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_filename
>>> sdf_filepath = get_data_filename('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
>>> molecule = Molecule(open(sdf_filepath, 'r'), file_format='sdf')
```

```
>>> import gzip
>>> mol2_gz_filepath = get_data_filename('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)
```

Todo:

- If a filename or file-like object is specified but the file contains more than one molecule, what is the

proper behavior? Read just the first molecule, or raise an exception if more than one molecule is found?

- Should we also support SMILES strings or IUPAC names for `other`?

Methods

`__init__`(*args, **kwargs)	Create a new Molecule object
`add_atom`(atomic_number, formal_charge, ...)	Add an atom
`add_bond`(atom1, atom2, bond_order, is_aromatic)	Add a bond between two specified atom indices
`add_bond_charge_virtual_site`(atoms, distance)	Create a bond charge-type virtual site, in which the location of the charge is specified by the position of two atoms.
`add_conformer`(coordinates)	# TODO: Should this not be public? Adds a conformer 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.
`chemical_environment_matches`(query[, ...])	Retrieve all matches for a given chemical environment query.
`compute_partial_charges`([toolkit_registry])	**Warning! Not Implemented!** Calculate partial atomic charges for this molecule using an underlying toolkit
`compute_partial_charges_am1bcc`([...])	Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit
`compute_wiberg_bond_orders`([charge_model, ...])	Calculate wiberg bond orders for this molecule using an underlying toolkit
`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`(filename[, file_format, ...])	Create one or more molecules from a file
`from_iupac`(iupac_name, **kwargs)	Generate a molecule from IUPAC or common name
`from_json`(serialized)	Instantiate an object from a JSON serialized representation.
`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_pickle`(serialized)	Instantiate an object from a pickle serialized representation.

Continued on next page

Table 6 – continued from previous page

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 openforcefield Topology containing a single Molecule object.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
generate_conformers([toolkit_registry, ...])	Generate conformers for this molecule using an underlying toolkit
get_fractional_bond_orders([method, ...])	Get fractional bond orders.
is_isomorphic(other[, ...])	Determines whether the molecules are isomorphic by comparing their graphs.
to_bson()	Return a BSON serialized representation.
to_dict()	Return a dictionary representation of the molecule.
to_file(outfile, outfile_format[, ...])	Write the current molecule to a file or file-like object
to_iupac()	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_rdkit([aromaticity_model])	Create an RDKit molecule
to_smiles([toolkit_registry])	Return a canonical isomeric SMILES representation of the current molecule
to_toml()	Return a TOML serialized representation.
to_topology()	Return an openforcefield Topology representation containing one copy of this molecule
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

Attributes

angles	Get an iterator over all i-j-k angles.
atoms	Iterate over all Atom objects.
bonds	Iterate over all Bond objects.
conformers	Iterate over all conformers in this molecule.
impropers	Iterate over all proper torsions in the molecule
n_angles	int: number of angles in the Molecule.
n_atoms	The number of Atom objects.
n_bonds	The number of Bond objects.
n_conformers	Iterate over all Atom objects.

Continued on next page

Table 7 – continued from previous page

n_impropers	int: number of improper torsions in the Molecule.
n_particles	The number of Particle objects, which corresponds to how many positions must be used.
n_propers	int: number of proper torsions in the Molecule.
n_virtual_sites	The number of VirtualSite objects.
name	The name (or title) of the molecule
partial_charges	Returns the partial charges (if present) on the molecule
particles	Iterate over all Particle objects.
propers	Iterate over all proper torsions in the molecule
properties	The properties dictionary of the molecule
torsions	Get an iterator over all i-j-k-l torsions.
total_charge	Return the total charge on the molecule
virtual_sites	Iterate over all VirtualSite objects.

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

atoms
Iterate over all Atom objects.

bonds
Iterate over all Bond objects.

chemical_environment_matches(*query*, *toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)
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.ToolRegistry or openforcefield.utils.toolkits.ToolkitWrapper, optional, default=GLOBAL_TOOLKIT_REGISTRY] `ToolkitRegistry` or `ToolkitWrapper` to use for chemical environment matches

Returns

matches [list of Atom tuples] A list of all matching Atom tuples

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]')
```

Todo:

- Do we want to generalize `query` to allow other kinds of queries, such as mdtraj DSL, pymol selections, atom index slices, etc? We could call it `topology.matches(query)` instead of `chemical_environment_matches`

compute_partial_charges(*toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)

Warning! Not Implemented! Calculate partial atomic charges for this molecule using an underlying toolkit

Parameters

quantum_chemical_method [string, default='AM1-BCC'] The quantum chemical method to use for partial charge calculation.

partial_charge_method [string, default='None'] The partial charge calculation method to use for partial charge calculation.

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

Raises

InvalidToolkitError If an invalid object is passed as the toolkit_registry parameter

Examples

molecule = Molecule.from_smiles('CCCCCC') molecule.generate_conformers() molecule.compute_partial_charges()

compute_partial_charges_am1bcc(*toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)

Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit

Parameters

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

Raises

InvalidToolkitError 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()
```

compute_wiberg_bond_orders(*charge_model=None*, *toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)

Calculate wiberg bond orders for this molecule using an underlying toolkit

Parameters

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

charge_model [string, optional] The charge model to use for partial charge calculation

Examples

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

>>> molecule.generate_conformers()

>>> molecule.compute_wiberg_bond_orders()

Raises

InvalidToolkitError If an invalid object is passed as the toolkit_registry parameter

conformers
Iterate over all conformers in this molecule.

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

static from_file(*filename*, *file_format=None*, *toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*, *allow_undefined_stereo=False*)
Create one or more molecules from a file

Todo:

- Extend this to also include some form of .offmol Open Force Field Molecule format?
- Generalize this to also include file-like objects?

Parameters

filename [str or file-like object] The name of 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.ToolRegistry 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_mol2file
>>> mol2_filename = get_monomer_mol2file('cyclohexane')
>>> molecule = Molecule.from_file(mol2_filename)
```

classmethod from_iupac(*iupac_name*, ***kwargs*)

Generate a molecule from IUPAC or common name

Parameters

iupac_name [str] IUPAC name of molecule to be generated

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 a common 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_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.

static 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 openforcefield molecule

Examples

Create a Molecule from an OpenEye OEMol

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

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

static 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.Molecule] An openforcefield molecule

Examples

Create a molecule from an RDKit molecule

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

static from_smiles(*smiles*, *hydrogens_are_explicit=False*, *toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)

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

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

static from_topology(*topology*)

Return a Molecule representation of an openforcefield 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)  # doctest: +SKIP
```

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=<openforcefield.utils.toolkits.ToolkitRegistry object>*, *num_conformers=10*, *clear_existing=True*)
Generate conformers for this molecule using an underlying toolkit

Parameters

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

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

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

Raises

InvalidToolkitError If an invalid object is passed as the toolkit_registry parameter

Examples

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

get_fractional_bond_orders(*method='Wiberg'*, *toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)
Get fractional bond orders.

method [str, optional, default='Wiberg'] The name of the charge method to use. Options are: * 'Wiberg' : Wiberg bond order

toolkit_registry [openforcefield.utils.toolkits ToolkitRegistry] The toolkit registry to use for molecule operations

Examples

Get fractional Wiberg bond orders

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> molecule.generate_conformers()
>>> fractional_bond_orders = molecule.get_fractional_bond_orders(method='Wiberg')
```

Todo:

- Is it OK that the `Molecule` object does not store geometry, but will create it using `openeye.omega` or `rdkit`?
- Should this method assign fractional bond orders to the `Bond``s in the molecule, a separate ``bond_orders` molecule property, or just return the array of bond orders?
- How do we add enough flexibility to specify the toolkit and optional parameters, such as: `oequacpac.OEAssignPartialCharges(charged_copy, getattr(oequacpac, 'OECharges_AM1BCCSym'), False, False)`
- Generalize to allow user to specify both QM method and bond order computation approach (e.g. `AM1` and `Wiberg`)

---

impropers
Iterate over all proper torsions in the molecule

Todo:

- Do we need to return a `Torsion` object that collects information about fractional bond orders?

---

is_isomorphic(*other*, *compare_atom_stereochemistry=True*, *compare_bond_stereochemistry=True*)
Determines whether the molecules are isomorphic by comparing their graphs.

Parameters

other [an openforcefield.topology.molecule.FrozenMolecule] The molecule to test for isomorphism.

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

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

Returns

molecules_are_isomorphic [bool]

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
Iterate over all Atom objects.

n_impropers
int: number of improper torsions in the Molecule.

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

n_propers
int: number of proper torsions in the Molecule.

n_virtual_sites
The number of VirtualSite objects.

name
The name (or title) of the molecule

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

Returns

partial_charges [a simtk.unit.Quantity - wrapped numpy array [1 x n_atoms]] The partial charges on this Molecule's atoms.

particles
Iterate over all Particle objects.

propers
Iterate over all proper torsions in the molecule

Todo:

- Do we need to return a `Torsion` object that collects information about fractional bond orders?

---

properties
The properties dictionary of the molecule

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.

Todo:

- Document the representation standard.
- How do we do version control with this standard?

Returns

molecule_dict [OrderedDict] A dictionary representation of the molecule.

to_file(*outfile*, *outfile_format*, *toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)

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

Parameters

outfile [str or file-like object] A file-like object or the filename of the file to be written to

outfile_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.ToolRegistry 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 outfile_format is not supported by one of the installed cheminformatics toolkits

Examples

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> molecule.to_file('imatinib.mol2', outfile_format='mol2')  # doctest: +SKIP
>>> molecule.to_file('imatinib.sdf', outfile_format='sdf')  # doctest: +SKIP
>>> molecule.to_file('imatinib.pdb', outfile_format='pdb')  # doctest: +SKIP
```

to_iupac()

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_filename
>>> sdf_filepath = get_data_filename('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.

Todo:

- Do we need a `from_networkx()` method? If so, what would the Graph be required to provide?
- Should edges be labeled with discrete bond types in some aromaticity model?
- Should edges be labeled with fractional bond order if a method is specified?
- Should we add other per-atom and per-bond properties (e.g. partial charges) if present?
- Can this encode bond/atom chirality?

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.

Todo:

- Use stored conformer positions instead of an argument.
- Should the aromaticity model be specified in some other way?

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_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_filename
>>> sdf_filepath = get_data_filename('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
>>> rdmol = molecule.to_rdkit()
```

to_smiles(*toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)
Return a canonical isomeric SMILES representation of the current molecule

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

Parameters

toolkit_registry [openforcefield.utils.toolkits.ToolRegistry 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_filename
>>> sdf_filepath = get_data_filename('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 openforcefield 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

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]

total_charge
Return the total charge on the molecule

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*, *charge_increments=None*, *weights=None*, *epsilon=None*, *sigma=None*, *rmin_half=None*, *name=''*)
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 ''.

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.

TODO : Do "weights" have any meaning here?

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 ''.

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*, *in_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.

TODO : Do "weights" have any meaning here?

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]

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 ''.

Returns

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

add_trivalent_lone_pair_virtual_site(*atoms*, *distance*, *out_of_plane_angle*, *in_plane_angle*, ***kwargs*)

Create a trivalent lone pair-type virtual site, in which the location of the charge is specified by the position of four atoms.

TODO : Do "weights" have any meaning here?

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]

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 ".

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*)

TODO: Should this not be public? Adds a conformer of the molecule

Parameters

coordinates: simtk.unit.Quantity(np.array) with shape (n_atoms, 3)
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 Index of the conformer in the Molecule

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.

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_filename, get_packmol_pdbfile
>>> pdb_filepath = get_packmol_pdbfile('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_filename(f'systems/monomers/{name}.sdf') for name in monomer_names]
>>> unique_molecules = [Molecule.from_file(sdf_filepath) for sdf_filepath in sdf_filepaths]
>>> topology = Topology.from_openmm(pdbfile.topology, unique_molecules=unique_molecules)
```

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

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

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

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

Attributes

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

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

box_vectors Return the box vectors of the topology, if specified

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

constrained_atom_pairs Returns the constrained atom pairs of the Topology

fractional_bond_order_model Get the fractional bond order model for the Topology.

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

n_angles int: number of angles in this Topology.

n_impropers int: number of improper torsions in this Topology.

n_propers int: number of proper torsions in this Topology.

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

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

n_topology_bonds Returns the number of TopologyBonds in in this Topology.

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

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

n_topology_virtual_sites Returns the number of TopologyVirtualSites in in this Topology.

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

reference_molecules Get an iterator of reference molecules in this Topology.

topology_atoms Returns an iterator over the atoms in this Topology.

topology_bonds Returns an iterator over the bonds in this Topology

topology_molecules Returns an iterator over all the TopologyMolecules in this Topology

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

topology_virtual_sites Get an iterator over the virtual sites in this Topology

Methods

add_constraint(iatom, jatom[, distance])	Mark a pair of atoms as constrained.
add_molecule(molecule[, ...])	Add a Molecule to the Topology.
add_particle(particle)	Add a Particle to the Topology.
assert_bonded(atom1, atom2)	Raise an exception if the specified atoms are not bonded in the topology.
atom(atom_topology_index)	Get the TopologyAtom at a given Topology atom index.
bond(bond_topology_index)	Get the TopologyBond at a given Topology bond index.
chemical_environment_matches(query[, ...])	Retrieve all matches for a given chemical environment query.
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(d)	Static constructor from dictionary representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.

Continued on next page

Table 8 – continued from previous page

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
get_fractional_bond_order(iatom, jatom)	Retrieve the fractional bond order for a bond.
is_bonded(i, j)	Returns True if the two atoms are bonded
is_constrained(iatom, jatom)	Check if a pair of atoms are marked as constrained.
to_bson()	Return a BSON serialized representation.
to_dict()	Convert to dictionary representation.
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.

Continued 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.

Continued 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
get_fractional_bond_order(iatom, jatom)	Retrieve the fractional bond order for a bond.
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.

Continued on next page

Table 9 – 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_parmed()	**Warning:** This functionality will be implemented in a future toolkit release.
to_pickle()	Return a pickle serialized representation.
to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.
virtual_site(vsite_topology_index)	Get the TopologyAtom at a given Topology atom index.

Attributes

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

Continued on next page

Table 10 – continued from previous page

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

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

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

Returns

aromaticity_model [str] Aromaticity model in use.

`box_vectors`
Return the box vectors of the topology, if specified Returns ——- box_vectors : simtk.unit.Quantity wrapped numpy array

The unit-wrapped box vectors of this topology

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

Returns

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

`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)

`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.

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

Returns

n_reference_molecules [int]

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

Returns

n_topology_molecules [int]

`topology_molecules`
Returns an iterator over all the TopologyMolecules in this Topology

Returns

topology_molecules [Iterable of TopologyMolecule]

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

Returns

n_topology_atoms [int]

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]

n_topology_bonds
Returns the number of TopologyBonds in in this Topology.

Returns

n_bonds [int]

topology_bonds
Returns an iterator over the bonds in this Topology

Returns

topology_bonds [Iterable of TopologyBond]

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

Returns

n_topology_particles [int]

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]

n_topology_virtual_sites
Returns the number of TopologyVirtualSites in in this Topology.

Returns

n_virtual_sites [iterable of TopologyVirtualSites]

topology_virtual_sites
Get an iterator over the virtual sites in this Topology

Returns

topology_virtual_sites [Iterable of TopologyVirtualSite]

n_angles
int: number of angles in this Topology.

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

n_propers
int: number of proper torsions in this Topology.

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

n_impropers
int: number of improper torsions in this Topology.

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

chemical_environment_matches(*query*, *aromaticity_model='MDL'*, *toolkit_registry=<openforcefield.utils.toolkits.ToolkitRegistry object>*)
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 TopologyAtom tuples] A list of all matching Atom tuples

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. If all bonds have bond orders assigned in `mdtraj_topology`, these bond orders will be used to attempt to construct the list of unique Molecules if the `unique_molecules` argument is omitted.

Returns

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

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 mult 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. If all bonds have bond orders assigned in `mdtraj_topology`, these bond orders will be used to attempt to construct the list of unique Molecules if the `unique_molecules` argument is omitted.

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 mult 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. If all bonds have bond orders assigned in the structure's `topology` object, these bond orders will be used to attempt to construct the list of unique Molecules if the `unique_molecules` argument is omitted.

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

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.

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

Returns

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

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_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

`get_fractional_bond_order`(*iatom*, *jatom*)
Retrieve the fractional bond order for a bond.

An Exception is raised if it cannot be determined.

Parameters

iatom, jatom [Atom] Atoms for which a fractional bond order is to be retrieved.

Returns

order [float] Fractional bond order between the two specified atoms.

2.1.2 Secondary objects

Particle	Base class for all particles in a molecule.
Atom	A particle representing a chemical atom.
Bond	Chemical bond representation.
VirtualSite	A particle representing a virtual site whose position is defined in terms of Atom positions.

openforcefield.topology.Particle

class openforcefield.topology.**Particle**

Base class for all particles in a molecule.

A particle object could be an Atom or a VirtualSite.

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

Attributes

molecule The Molecule this atom is part of.

molecule_particle_index Returns the index of this particle in its molecule

name The name of the particle

Methods

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

Continued on next page

Table 12 – continued from previous page

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__(*$self, /, *args, **kwargs*)
Initialize self. See help(type(self)) for accurate signature.

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.

Attributes

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

molecule
The Molecule this atom is part of.

Todo:

- Should we have a single unique Molecule for each molecule type in the system,

or if we have multiple copies of the same molecule, should we have multiple "Molecule"s?

molecule_particle_index
Returns the index of this particle in its molecule

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.

Todo:

- Should `Atom` objects be immutable or mutable?

- Do we want to support the addition of arbitrary additional properties, such as floating point quantities (e.g. `charge`), integral quantities (such as `id` or `serial` index in a PDB file), or string labels (such as Lennard-Jones types)?

Todo: Allow atoms to have associated properties.

Warning: This API is experimental and subject to change.

Attributes

atomic_number The integer atomic number of the atom.

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

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

element The element name

formal_charge The atom's formal charge

is_aromatic The atom's is_aromatic flag

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

molecule The `Molecule` this atom is part of.

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

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

name The name of this atom, if any

partial_charge The partial charge of the atom, if any.

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

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

Methods

add_bond(bond)	Adds a bond that this atom is involved in ..
add_virtual_site(vsite)	Adds a bond that this atom is involved in ..
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(atom_dict)	Create an Atom from a dict representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.

Continued on next page

Table 15 – 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.
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.

Todo: Use attrs to validate?

Todo: We can add setters if we need to.

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 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
`mass`	The standard atomic weight (abundance-weighted isotopic mass) of the atomic site.
`molecule`	The `Molecule` this atom is part of.
`molecule_atom_index`	The index of this Atom within the the list of atoms in `Molecules`.
`molecule_particle_index`	The index of this Atom within the the list of particles in the parent `Molecule`.
`name`	The name of this atom, if any
`partial_charge`	The partial charge of the atom, if any.

Continued on next page

Table 17 – continued from previous page

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.

formal_charge
The atom's formal charge

partial_charge
The partial charge of the atom, if any.

Returns

float or None

is_aromatic
The atom's is_aromatic flag

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

element
The element name

atomic_number
The integer atomic number of the atom.

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

Todo: Should we discriminate between standard atomic weight and most abundant isotopic mass?

TODO (from jeff): Are there atoms that have different chemical properties based on their isotopes?

name
The name of this atom, if any

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

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

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

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`.

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

molecule
The `Molecule` this atom is part of.

Todo:

- Should we have a single unique `Molecule` for each molecule type in the system,

or if we have multiple copies of the same molecule, should we have multiple "Molecule"s?

---

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.

Todo: Allow bonds to have associated properties.

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.

Continued on next page

Table 18 – continued from previous page

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

Continued on next page

Table 20 – continued from previous page

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

to_dict()
Return a dict representation of the bond.

classmethod from_dict(*molecule*, *d*)
Create a Bond from a dict representation.

molecule_bond_index
The index of this Bond within the the list of bonds in Molecules.

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*)

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

Note that chemical atoms are represented by the `Atom`.

Warning: This API is experimental and subject to change.

Todo:

- Should a virtual site be able to belong to more than one Topology?
- Should virtual sites be immutable or mutable?

Attributes

atoms Atoms on whose position this VirtualSite depends.

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

epsilon The VdW epsilon term of this VirtualSite

molecule The `Molecule` this atom is part of.

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

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

name The name of this VirtualSite

rmin_half The VdW rmin_half term of this VirtualSite

sigma The VdW sigma term of this VirtualSite

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

Methods

from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(vsite_dict)	Create a virtual site from a dict representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.

Continued on next page

Table 21 – continued from previous page

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

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

Base class for VirtualSites

Todo:

- This will need to be generalized for virtual sites to allow out-of-plane sites, which are not simply a linear combination of atomic positions
- Add checking for allowed virtual site types
- How do we want users to specify virtual site types?

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.

Continued on next page

Table 22 – continued from previous page

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

Attributes

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

to_dict()
Return a dict representation of the virtual site.

static from_dict(*vsite_dict*)
Create a virtual site from a dict representation.

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.

molecule_particle_index

The index of this VirtualSite within the the list of particles in the parent `Molecule`. Note that this can be different from `molecule_virtual_site_index`.

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

sigma
The VdW sigma term of this VirtualSite

rmin_half
The VdW rmin_half term of this VirtualSite

name
The name of this VirtualSite

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

molecule
The `Molecule` this atom is part of.

Todo:

- Should we have a single unique `Molecule` for each molecule type in the system,

or if we have multiple copies of the same molecule, should we have multiple "Molecule"s?

---

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

2.2 Forcefield typing tools

2.2.1 Chemical environments

Tools for representing and operating on chemical environments

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

ChemicalEnvironment	Chemical environment abstract base class that matches an atom, bond, angle, etc.

openforcefield.typing.chemistry.ChemicalEnvironment

class openforcefield.typing.chemistry.**ChemicalEnvironment**(*smirks=None*, *label=None*, *replacements=None*, *toolkit='openeye'*)

Chemical environment abstract base class that matches an atom, bond, angle, etc.

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

Methods

Atom([ORtypes, ANDtypes, index, ring])	Atom representation, which may have some ORtypes and ANDtypes properties.
Bond([ORtypes, ANDtypes])	Bond representation, which may have ORtype and ANDtype descriptors.
addAtom(bondToAtom[, bondORtypes, ...])	Add an atom to the specified target atom.
asSMIRKS([smarts])	Returns a SMIRKS representation of the chemical environment
getAlphaAtoms()	Returns a list of atoms alpha to any indexed atom
getAlphaBonds()	Returns a list of Bond objects that connect
getAtoms() **Returns**	
getBetaAtoms()	Returns a list of atoms beta to any indexed atom
getBetaBonds()	Returns a list of Bond objects that connect
getBond(atom1, atom2)	Get bond betwen two atoms

Continued on next page

Table 25 – continued from previous page

getBondOrder(atom)	Returns minimum bond order around a given atom 0 if atom has no neighbors aromatic bonds count as 1.5 any bond counts as 1.0
getBonds([atom])	**Parameters**
getComponentList(component_type[, descriptor])	Returns a list of atoms or bonds matching the descriptor
getIndexedAtoms()	returns the list of Atom objects with an index
getIndexedBonds()	Returns a list of Bond objects that connect two indexed atoms
getNeighbors(atom)	Returns atoms that are bound to the given atom in the form of a list of Atom objects
getType()	Uses number of indexed atoms and bond connectivity to determine the type of chemical environment
getUnindexedAtoms()	returns a list of Atom objects that are not indexed
getUnindexedBonds()	Returns a list of Bond objects that connect
getValence(atom)	Returns the valence (number of neighboring atoms) around the given atom
isAlpha(component)	Takes an atom or bond are returns True if it is alpha to an indexed atom
isBeta(component)	Takes an atom or bond are returns True if it is beta to an indexed atom
isIndexed(component)	returns True if the atom or bond is indexed
isUnindexed(component)	returns True if the atom or bond is not indexed
isValid([smirks])	Returns if the environment is valid, that is if it creates a parseable SMIRKS string.
removeAtom(atom[, onlyEmpty])	Remove the specified atom from the chemical environment.
selectAtom([descriptor])	Select a random atom fitting the descriptor.
selectBond([descriptor])	Select a random bond fitting the descriptor.
validate(smirks[, ensure_valence_type, toolkit])	Validate the provided SMIRKS string is valid, and if requested, tags atoms appropriate to the specified valence type.

__init__(*smirks=None*, *label=None*, *replacements=None*, *toolkit='openeye'*)
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

replacements = list of lists, optional, [substitution, smarts] form for parsing SMIRKS

Methods

__init__([smirks, label, replacements, toolkit])	Initialize a chemical environment abstract base class.
addAtom(bondToAtom[, bondORtypes, ...])	Add an atom to the specified target atom.
asSMIRKS([smarts])	Returns a SMIRKS representation of the chemical environment
getAlphaAtoms()	Returns a list of atoms alpha to any indexed atom
getAlphaBonds()	Returns a list of Bond objects that connect
getAtoms()	**Returns**
getBetaAtoms()	Returns a list of atoms beta to any indexed atom
getBetaBonds()	Returns a list of Bond objects that connect
getBond(atom1, atom2)	Get bond betwen two atoms
getBondOrder(atom)	Returns minimum bond order around a given atom 0 if atom has no neighbors aromatic bonds count as 1.5 any bond counts as 1.0
getBonds([atom])	**Parameters**
getComponentList(component_type[, descriptor])	Returns a list of atoms or bonds matching the descriptor
getIndexedAtoms()	returns the list of Atom objects with an index
getIndexedBonds()	Returns a list of Bond objects that connect two indexed atoms
getNeighbors(atom)	Returns atoms that are bound to the given atom in the form of a list of Atom objects
getType()	Uses number of indexed atoms and bond connectivity to determine the type of chemical environment
getUnindexedAtoms()	returns a list of Atom objects that are not indexed
getUnindexedBonds()	Returns a list of Bond objects that connect
getValence(atom)	Returns the valence (number of neighboring atoms) around the given atom
isAlpha(component)	Takes an atom or bond are returns True if it is alpha to an indexed atom
isBeta(component)	Takes an atom or bond are returns True if it is beta to an indexed atom
isIndexed(component)	returns True if the atom or bond is indexed
isUnindexed(component)	returns True if the atom or bond is not indexed
isValid([smirks])	Returns if the environment is valid, that is if it creates a parseable SMIRKS string.
removeAtom(atom[, onlyEmpty])	Remove the specified atom from the chemical environment.
selectAtom([descriptor])	Select a random atom fitting the descriptor.
selectBond([descriptor])	Select a random bond fitting the descriptor.
validate(smirks[, ensure_valence_type, toolkit])	Validate the provided SMIRKS string is valid, and if requested, tags atoms appropriate to the specified valence type.

class Atom(*ORtypes=None*, *ANDtypes=None*, *index=None*, *ring=None*)

Atom representation, which may have some ORtypes and ANDtypes properties.

Attributes

ORtypes [list of tuples in the form (base, [list of decorators])] where bases and decorators are both strings The descriptor types that will be combined with logical OR

ANDtypes [list of string] The descriptor types that will be combined with logical AND

Methods

addANDtype(ANDtype)	Adds ANDtype to the set for this atom.
addORtype(ORbase, ORdecorators)	Adds ORtype to the set for this atom.
asSMARTS()	Return the atom representation as SMARTS.
asSMIRKS()	Return the atom representation as SMIRKS.
getANDtypes()	returns a copy of the list of ANDtypes for this atom
getORtypes()	returns a copy of the dictionary of ORtypes for this atom
setANDtypes(newANDtypes)	sets new ANDtypes for this atom
setORtypes(newORtypes)	sets new ORtypes for this atom

asSMARTS()

Return the atom representation as SMARTS.

Returns

smarts [str]

The SMARTS string for this atom

asSMIRKS()

Return the atom representation as SMIRKS.

Returns

smirks [str]

The SMIRKS string for this atom

addORtype(*ORbase*, *ORdecorators*)

Adds ORtype to the set for this atom.

Parameters

ORbase: string, such as '#6'

ORdecorators: list of strings, such as ['X4','+0']

addANDtype(*ANDtype*)

Adds ANDtype to the set for this atom.

Parameters

ANDtype: string added to the list of ANDtypes for this atom

getORtypes()

returns a copy of the dictionary of ORtypes for this atom

setORtypes(*newORtypes*)

sets new ORtypes for this atom

Parameters

newORtypes: list of tuples in the form (base, [ORdecorators]) for example: ('#6', ['X4','H0','+0']) –> '#6X4H0+0'

getANDtypes()
returns a copy of the list of ANDtypes for this atom

setANDtypes(*newANDtypes*)
sets new ANDtypes for this atom
Parameters
newANDtypes: list of strings strings that will be AND'd together in a SMARTS

class Bond(*ORtypes=None*, *ANDtypes=None*)
Bond representation, which may have ORtype and ANDtype descriptors.

Attributes

ORtypes [list of tuples of ORbases and ORdecorators] in form (base: [list of decorators]) The ORtype types that will be combined with logical OR

ANDtypes [list of string] The ANDtypes that will be combined with logical AND

Methods

addANDtype(ANDtype)	Adds ANDtype to the set for this atom.
addORtype(ORbase, ORdecorators)	Adds ORtype to the set for this atom.
asSMARTS()	Return the atom representation as SMARTS.
asSMIRKS()	

Returns

getANDtypes()	returns a copy of the list of ANDtypes for this atom
getORtypes()	returns a copy of the dictionary of ORtypes for this atom
getOrder()	Returns a float for the order of this bond for multiple ORtypes or ~ it returns the minimum possible order the intended application is for checking valence around a given atom
setANDtypes(newANDtypes)	sets new ANDtypes for this atom
setORtypes(newORtypes)	sets new ORtypes for this atom

asSMARTS()
Return the atom representation as SMARTS.
Returns
smarts [str] The SMARTS string for just this atom

asSMIRKS()
Returns
smarts [str] The SMIRKS string for just this bond

getOrder()
Returns a float for the order of this bond for multiple ORtypes or ~ it returns the minimum possible order the intended application is for checking valence around a given atom

addANDtype(*ANDtype*)
Adds ANDtype to the set for this atom.
Parameters
ANDtype: string added to the list of ANDtypes for this atom

addORtype(*ORbase*, *ORdecorators*)

Adds ORtype to the set for this atom.

Parameters

ORbase: string, such as '#6'

ORdecorators: list of strings, such as ['X4','+0']

`getANDtypes`()

returns a copy of the list of ANDtypes for this atom

`getORtypes`()

returns a copy of the dictionary of ORtypes for this atom

`setANDtypes`(*newANDtypes*)

sets new ANDtypes for this atom

Parameters

newANDtypes: list of strings strings that will be AND'd together in a SMARTS

`setORtypes`(*newORtypes*)

sets new ORtypes for this atom

Parameters

newORtypes: list of tuples in the form (base, [ORdecorators]) for example: ('#6', ['X4','H0','+0']) –> '#6X4H0+0'

`static validate`(*smirks*, *ensure_valence_type=None*, *toolkit='openeye'*)

Validate 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

ensure_valence_type [str, optional, default=None] If specified, ensure the tagged atoms are appropriate to the specified valence type

This method will raise a :class:'SMIRKSParsingError' if the provided SMIRKS string is not valid.

`isValid`(*smirks=None*)

Returns if the environment is valid, that is if it creates a parseable SMIRKS string.

`asSMIRKS`(*smarts=False*)

Returns a SMIRKS representation of the chemical environment

Parameters

smarts: optional, boolean if True, returns a SMARTS instead of SMIRKS without index labels

`selectAtom`(*descriptor=None*)

Select a random atom fitting the descriptor.

Parameters

descriptor: optional, None None - returns any atom with equal probability int - will return an atom with that index 'Indexed' - returns a random indexed atom 'Unindexed' - returns a random unindexed atom 'Alpha' - returns a random alpha atom 'Beta' - returns a random beta atom

Returns

a single Atom object fitting the description

or None if no such atom exists

getComponentList(*component_type*, *descriptor=None*)
Returns a list of atoms or bonds matching the descriptor

Parameters

component_type: string: 'atom' or 'bond'

descriptor: string, optional 'all', 'Indexed', 'Unindexed', 'Alpha', 'Beta'

selectBond(*descriptor=None*)
Select a random bond fitting the descriptor.

Parameters

descriptor: optional, None None - returns any bond with equal probability int - will return an bond with that index 'Indexed' - returns a random indexed bond 'Unindexed' - returns a random unindexed bond 'Alpha' - returns a random alpha bond 'Beta' - returns a random beta bond

Returns

a single Bond object fitting the description

or None if no such atom exists

addAtom(*bondToAtom*, *bondORtypes=None*, *bondANDtypes=None*, *newORtypes=None*, *newANDtypes=None*, *newAtomIndex=None*, *newAtomRing=None*, *beyondBeta=False*)
Add an atom to the specified target atom.

Parameters

bondToAtom: atom object, required atom the new atom will be bound to

bondORtypes: list of tuples, optional strings that will be used for the ORtypes for the new bond

bondANDtypes: list of strings, optional strings that will be used for the ANDtypes for the new bond

newORtypes: list of strings, optional strings that will be used for the ORtypes for the new atom

newANDtypes: list of strings, optional strings that will be used for the ANDtypes for the new atom

newAtomIndex: int, optional integer label that could be used to index the atom in a SMIRKS string

beyondBeta: boolean, optional if True, allows bonding beyond beta position

Returns

newAtom: atom object for the newly created atom

removeAtom(*atom*, *onlyEmpty=False*)
Remove the specified atom from the chemical environment. if the atom is not indexed for the SMIRKS string or used to connect two other atoms.

Parameters

atom: atom object, required atom to be removed if it meets the conditions.

onlyEmpty: boolean, optional True only an atom with no ANDtypes and 1 ORtype can be removed

Returns

Boolean True: atom was removed, False: atom was not removed

getAtoms()

Returns

list of atoms in the environment

getBonds(*atom=None*)

Parameters

atom: Atom object, optional, returns bonds connected to atom

returns all bonds in fragment if atom is None

Returns

a complete list of bonds in the fragment

getBond(*atom1*, *atom2*)
Get bond betwen two atoms

Parameters

atom1 and atom2: atom objects

Returns

bond object between the atoms or None if no bond there

getIndexedAtoms()
returns the list of Atom objects with an index

getUnindexedAtoms()
returns a list of Atom objects that are not indexed

getAlphaAtoms()

Returns a list of atoms alpha to any indexed atom that are not also indexed

getBetaAtoms()

Returns a list of atoms beta to any indexed atom that are not alpha or indexed atoms

getIndexedBonds()
Returns a list of Bond objects that connect two indexed atoms

getUnindexedBonds()

Returns a list of Bond objects that connect an indexed atom to an unindexed atom two unindexed atoms

getAlphaBonds()

Returns a list of Bond objects that connect an indexed atom to alpha atoms

getBetaBonds()

Returns a list of Bond objects that connect alpha atoms to beta atoms

isAlpha(*component*)
Takes an atom or bond are returns True if it is alpha to an indexed atom

isUnindexed(*component*)
returns True if the atom or bond is not indexed

isIndexed(*component*)
returns True if the atom or bond is indexed

isBeta(*component*)
Takes an atom or bond are returns True if it is beta to an indexed atom

getType()
Uses number of indexed atoms and bond connectivity to determine the type of chemical environment

Returns

chemical environemnt type: 'Atom', 'Bond', 'Angle', 'ProperTorsion', 'ImproperTorsion' None if number of indexed atoms is 0 or > 4

getNeighbors(*atom*)
Returns atoms that are bound to the given atom in the form of a list of Atom objects

getValence(*atom*)
Returns the valence (number of neighboring atoms) around the given atom

getBondOrder(*atom*)
Returns minimum bond order around a given atom 0 if atom has no neighbors aromatic bonds count as 1.5 any bond counts as 1.0

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`

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.

openforcefield.typing.engines.smirnoff.parameters.ParameterType

class openforcefield.typing.engines.smirnoff.parameters.**ParameterType**(*smirks=None*, *permit_cosmetic_attributes=False*, ***kwargs*)

Base class for SMIRNOFF parameter types.

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

Attributes

smirks

Methods

to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

__init__(*smirks=None*, *permit_cosmetic_attributes=False*, ***kwargs*)

Create a ParameterType

Parameters

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

permit_cosmetic_attributes [bool optional. Default = False] Whether to store non-spec kwargs as "cosmetic attributes", which can be accessed and written out.

Methods

__init__([smirks, permit_cosmetic_attributes])	Create a ParameterType
to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

Attributes

smirks

to_dict(*return_cosmetic_attributes=False*)

Convert this ParameterType-derived object to dict. A unit-bearing attribute ('X') will be converted to two dict entries, one (['X'] containing the unitless value, and another (['X_unit']) containing a string representation of its unit.

Parameters

return_cosmetic_attributes [bool, optional. default = False] Whether to return non-spec attributes of this ParameterType

Returns

smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this

ParameterType-derived object.

output_units [dict[str: simtk.unit.Unit]] A mapping from each simtk.unit.Quanitity-valued ParameterType attribute to the unit it was converted to during serialization.

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

class openforcefield.typing.engines.smirnoff.parameters.BondHandler.**BondType**(***kwargs*)

A SMIRNOFF bond type

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

Attributes

smirks

Methods

to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

__init__(***kwargs*)

Create a ParameterType

Parameters

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

permit_cosmetic_attributes [bool optional. Default = False] Whether to store non-spec kwargs as "cosmetic attributes", which can be accessed and written out.

Methods

__init__(**kwargs)	Create a ParameterType
to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

Attributes

smirks

to_dict(*return_cosmetic_attributes=False*)

Convert this ParameterType-derived object to dict. A unit-bearing attribute ('X') will be converted to two dict entries, one (['X'] containing the unitless value, and another (['X_unit']) containing a string representation of its unit.

Parameters

return_cosmetic_attributes [bool, optional. default = False] Whether to return non-spec attributes of this ParameterType

Returns

smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this ParameterType-derived object.

output_units [dict[str: simtk.unit.Unit]] A mapping from each simtk.unit.Quanitity-valued ParameterType attribute to the unit it was converted to during serialization.

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

class openforcefield.typing.engines.smirnoff.parameters.AngleHandler.**AngleType**(***kwargs*)

A SMIRNOFF angle type.

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

Attributes

smirks

Methods

to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

__init__(***kwargs*)

Create a ParameterType

Parameters

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

permit_cosmetic_attributes [bool optional. Default = False] Whether to store non-spec kwargs as "cosmetic attributes", which can be accessed and written out.

Methods

__init__(**kwargs)	Create a ParameterType
to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

Attributes

smirks

to_dict(*return_cosmetic_attributes=False*)

Convert this ParameterType-derived object to dict. A unit-bearing attribute ('X') will be converted to two dict entries, one (['X'] containing the unitless value, and another (['X_unit']) containing a string representation of its unit.

Parameters

return_cosmetic_attributes [bool, optional. default = False] Whether to return non-spec attributes of this ParameterType

Returns

smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this ParameterType-derived object.

output_units [dict[str: simtk.unit.Unit]] A mapping from each simtk.unit.Quanitity-valued ParameterType attribute to the unit it was converted to during serialization.

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

class openforcefield.typing.engines.smirnoff.parameters.ProperTorsionHandler.**ProperTorsionType**(***kwargs*)

A SMIRNOFF torsion type for proper torsions.

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

Attributes

smirks

Methods

to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

__init__(***kwargs*)

Create a ParameterType

Parameters

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

permit_cosmetic_attributes [bool optional. Default = False] Whether to store non-spec kwargs as "cosmetic attributes", which can be accessed and written out.

Methods

__init__(**kwargs)	Create a ParameterType
to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

Attributes

smirks

to_dict(*return_cosmetic_attributes=False*)

Convert this ParameterType-derived object to dict. A unit-bearing attribute ('X') will be converted to two dict entries, one (['X'] containing the unitless value, and another (['X_unit']) containing a

string representation of its unit.

Parameters

return_cosmetic_attributes [bool, optional. default = False] Whether to return non-spec attributes of this ParameterType

Returns

smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this ParameterType-derived object.

output_units [dict[str: simtk.unit.Unit]] A mapping from each simtk.unit.Quanitity-valued ParameterType attribute to the unit it was converted to during serialization.

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

class openforcefield.typing.engines.smirnoff.parameters.ImproperTorsionHandler.**ImproperTorsionType**(***kwargs*)

A SMIRNOFF torsion type for improper torsions.

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

Attributes

smirks

Methods

to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

__init__(***kwargs*)

Create a ParameterType

Parameters

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

permit_cosmetic_attributes [bool optional. Default = False] Whether to store non-spec kwargs as "cosmetic attributes", which can be accessed and written out.

Methods

__init__(**kwargs)	Create a ParameterType
to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

Attributes

smirks

to_dict(*return_cosmetic_attributes=False*)
Convert this ParameterType-derived object to dict. A unit-bearing attribute ('X') will be converted to two dict entries, one (['X'] containing the unitless value, and another (['X_unit']) containing a string representation of its unit.

Parameters

return_cosmetic_attributes [bool, optional. default = False] Whether to return non-spec attributes of this ParameterType

Returns

smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this ParameterType-derived object.

output_units [dict[str: simtk.unit.Unit]] A mapping from each simtk.unit.Quanitity-valued ParameterType attribute to the unit it was converted to during serialization.

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

class openforcefield.typing.engines.smirnoff.parameters.vdWHandler.**vdWType**(***kwargs*)
A SMIRNOFF vdWForce type.

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

Attributes

smirks

Methods

to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

__init__(***kwargs*)
Create a ParameterType

Parameters

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

permit_cosmetic_attributes [bool optional. Default = False] Whether to store non-spec kwargs as "cosmetic attributes", which can be accessed and written out.

Methods

__init__(**kwargs)	Create a ParameterType
to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

Attributes

smirks

to_dict(*return_cosmetic_attributes=False*)
Convert this ParameterType-derived object to dict. A unit-bearing attribute ('X') will be converted to two dict entries, one (['X'] containing the unitless value, and another (['X_unit']) containing a string representation of its unit.

Parameters

return_cosmetic_attributes [bool, optional. default = False] Whether to return non-spec attributes of this ParameterType

Returns

smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this ParameterType-derived object.

output_units [dict[str: simtk.unit.Unit]] A mapping from each simtk.unit.Quanitity-valued ParameterType attribute to the unit it was converted to during serialization.

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.
`ToolkitAM1BCCHandler`	Handle SMIRNOFF `<ToolkitAM1BCC>` 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()	
copy()	
count(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])	Raises IndexError if list is empty or index is out of range.
remove(value)	Raises ValueError if the value is not present.
reverse	L.reverse() – reverse *IN PLACE*
sort([key, reverse])	
to_list([return_cosmetic_attributes])	Render this ParameterList to a normal list, serializing each ParameterType-derived 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()	
copy()	
count(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])	Raises IndexError if list is empty or index is out of range.
remove(value)	Raises ValueError if the value is not present.
reverse	L.reverse() – reverse *IN PLACE*
sort([key, reverse])	
to_list([return_cosmetic_attributes])	Render this ParameterList to a normal list, serializing each ParameterType-derived object in it to dict.

append(*parameter*)
Add a ParameterType object to the end of the ParameterList

Parameters

parameter [a ParameterType-derived 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 ValueError if the item is not found.

Parameters

item [ParameterType-derived object or str] The parameter or SMIRKS to look up in this ParameterList

Returns

index [int] The index of the found item

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-derived object] The parameter to insert

to_list(*return_cosmetic_attributes=False*)
Render this ParameterList to a normal list, serializing each ParameterType-derived object in it to dict.

Parameters

return_cosmetic_attributes [bool, optional. default = False] Whether to return non-spec attributes of each ParameterType-derived object.

Returns

parameter_list [List[dict]] A serialized representation of a ParameterList, with each ParameterType it contains converted to dict.

clear() → None – remove all items from L

copy() → list – a shallow copy of L

count(*value*) → integer – return number of occurrences of value

pop([*index*]) → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

remove(*value*) → None – remove first occurrence of value.
Raises ValueError if the value is not present.

reverse()
L.reverse() – reverse *IN PLACE*

sort(*key=None*, *reverse=False*) → None – stable sort *IN PLACE*

openforcefield.typing.engines.smirnoff.parameters.ParameterHandler

class openforcefield.typing.engines.smirnoff.parameters.**ParameterHandler**(*permit_cosmetic_attributes=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 responsile for assigning SMIRKS-based parameters.

Warning: This API is experimental and subject to change.

Attributes

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

parameters The ParameterList that holds this ParameterHandler's parameter objects

Methods

add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
check_parameter_compatibility(parameter_kwargs)	Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler
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([output_units, ...])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

__init__(*permit_cosmetic_attributes=False*, ***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

permit_cosmetic_attributes [bool] Whether to accept non-spec kwargs

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

Methods

__init__([permit_cosmetic_attributes])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
check_parameter_compatibility(parameter_kwargs)	Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler
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([output_units, ...])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects

parameters
The ParameterList that holds this ParameterHandler's parameter objects

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

check_parameter_compatibility(*parameter_kwargs*)
Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler

Parameters

parameter_kwargs: dict The dict that will be used to construct the ParameterType

Raises

Raises a ValueError if the parameters are incompatible.

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*)
Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs [dict] The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

get_parameter(*parameter_attrs*)
Return the parameters in this ParameterHandler that match the parameter_attrs argument

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

list of ParameterType-derived objects A list of matching ParameterType-derived objects

find_matches(*entity*)
Find the elements of the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology or openforcefield.topology.Molecule] Topology or molecule to search.

Returns

matches [ValenceDict[Tuple[int], ParameterType]] `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(*output_units=None*, *return_cosmetic_attributes=False*)
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

output_units [dict[str] A mapping from the ParameterType attribute name to the output unit its value should be converted to.

return_cosmetic_attributes [bool, optional. Default = False.] Whether to return 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.BondHandler

class openforcefield.typing.engines.smirnoff.parameters.**BondHandler**(***kwargs*)

Handle SMIRNOFF <Bonds> tags

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

Attributes

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

parameters The ParameterList that holds this ParameterHandler's parameter objects

Methods

BondType(**kwargs)	A SMIRNOFF bond type
add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
check_parameter_compatibility(parameter_kwargs)	Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler
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([output_units, ...])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

create_force	

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

Parameters

permit_cosmetic_attributes [bool] Whether to accept non-spec kwargs

****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_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
check_parameter_compatibility(parameter_kwargs)	Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler
create_force(system, topology, **kwargs)	
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([output_units, ...])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects

class BondType(***kwargs*)
A SMIRNOFF bond type

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

Attributes

smirks

Methods

to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

to_dict(*return_cosmetic_attributes=False*)
Convert this ParameterType-derived object to dict. A unit-bearing attribute ('X') will be converted to two dict entries, one (['X'] containing the unitless value, and another (['X_unit']) containing a string representation of its unit.

Parameters
return_cosmetic_attributes [bool, optional. default = False] Whether to return non-spec attributes of this ParameterType

Returns
smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this ParameterType-derived object.
output_units [dict[str: simtk.unit.Unit]] A mapping from each simtk.unit.Quanitity-valued ParameterType attribute to the unit it was converted to during serialization.

add_parameter(*parameter_kwargs*)
Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs [dict] The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

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.

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.

check_parameter_compatibility(*parameter_kwargs*)
Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler

Parameters

parameter_kwargs: dict The dict that will be used to construct the ParameterType

Raises

Raises a ValueError if the parameters are incompatible.

find_matches(*entity*)
Find the elements of the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology or openforcefield.topology.Molecule] Topology or molecule to search.

Returns

matches [ValenceDict[Tuple[int], ParameterType]] `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

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

list of ParameterType-derived objects A list of matching ParameterType-derived objects

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

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(*output_units=None*, *return_cosmetic_attributes=False*)
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

output_units [dict[str] A mapping from the ParameterType attribute name to the output unit its value should be converted to.

return_cosmetic_attributes [bool, optional. Default = False.] Whether to return 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**(***kwargs*)

Handle SMIRNOFF <AngleForce> tags

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

Attributes

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

parameters The ParameterList that holds this ParameterHandler's parameter objects

Methods

AngleType(**kwargs)	A SMIRNOFF angle type.
add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
check_parameter_compatibility(parameter_kwargs)	Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler
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([output_units, ...])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

create_force	

__init__(***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

permit_cosmetic_attributes [bool] Whether to accept non-spec kwargs

****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_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
check_parameter_compatibility(parameter_kwargs)	Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler
create_force(system, topology, **kwargs)	
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([output_units, ...])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects

class AngleType(***kwargs*)

A SMIRNOFF angle type.

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

Attributes

smirks

Methods

to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

to_dict(*return_cosmetic_attributes=False*)

Convert this ParameterType-derived object to dict. A unit-bearing attribute ('X') will be converted to two dict entries, one (['X'] containing the unitless value, and another (['X_unit']) containing a string representation of its unit.

Parameters

return_cosmetic_attributes [bool, optional. default = False] Whether to return non-spec attributes of this ParameterType

Returns

smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this ParameterType-derived object.

output_units [dict[str: simtk.unit.Unit]] A mapping from each simtk.unit.Quanitity-valued ParameterType attribute to the unit it was converted to during serialization.

add_parameter(*parameter_kwargs*)

Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs [dict] The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

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.

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.

check_parameter_compatibility(*parameter_kwargs*)

Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler

Parameters

parameter_kwargs: dict The dict that will be used to construct the ParameterType

Raises

Raises a ValueError if the parameters are incompatible.

find_matches(*entity*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology or openforcefield.topology.Molecule] Topology or molecule to search.

Returns

matches [ValenceDict[Tuple[int], ParameterType]] `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

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

list of ParameterType-derived objects A list of matching ParameterType-derived objects

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

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(*output_units=None*, *return_cosmetic_attributes=False*)
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

output_units [dict[str] A mapping from the ParameterType attribute name to the output unit its value should be converted to.

return_cosmetic_attributes [bool, optional. Default = False.] Whether to return 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**(***kwargs*)
Handle SMIRNOFF `<ProperTorsionForce>` tags

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

Attributes

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

parameters The ParameterList that holds this ParameterHandler's parameter objects

Methods

ProperTorsionType(**kwargs)	A SMIRNOFF torsion type for proper torsions.
add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
check_parameter_compatibility(parameter_kwargs)	Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler
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([output_units, ...])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

create_force

__init__(***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

permit_cosmetic_attributes [bool] Whether to accept non-spec kwargs

****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_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.

Continued on next page

Table 64 – continued from previous page

check_parameter_compatibility(parameter_kwargs)	Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler
create_force(system, topology, **kwargs)	
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([output_units, ...])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects

class ProperTorsionType(***kwargs*)

A SMIRNOFF torsion type for proper torsions.

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

Attributes

smirks

Methods

to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

to_dict(*return_cosmetic_attributes=False*)

Convert this ParameterType-derived object to dict. A unit-bearing attribute ('X') will be converted to two dict entries, one (['X'] containing the unitless value, and another (['X_unit']) containing a string representation of its unit.

Parameters

return_cosmetic_attributes [bool, optional. default = False] Whether to return non-spec attributes of this ParameterType

Returns

smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this ParameterType-derived object.

output_units [dict[str: simtk.unit.Unit]] A mapping from each simtk.unit.Quanitity-valued ParameterType attribute to the unit it was converted to during serialization.

add_parameter(*parameter_kwargs*)
Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs [dict] The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

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.

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.

check_parameter_compatibility(*parameter_kwargs*)
Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler

Parameters

parameter_kwargs: dict The dict that will be used to construct the ParameterType

Raises

Raises a ValueError if the parameters are incompatible.

find_matches(*entity*)
Find the elements of the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology or openforcefield.topology.Molecule] Topology or molecule to search.

Returns

matches [ValenceDict[Tuple[int], ParameterType]] `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

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

list of ParameterType-derived objects A list of matching ParameterType-derived objects

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

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(*output_units=None*, *return_cosmetic_attributes=False*)
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

output_units [dict[str] A mapping from the ParameterType attribute name to the output unit its value should be converted to.

return_cosmetic_attributes [bool, optional. Default = False.] Whether to return 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**(***kwargs*)
Handle SMIRNOFF <ImproperTorsionForce> tags

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

Attributes

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

parameters The ParameterList that holds this ParameterHandler's parameter objects

Methods

ImproperTorsionType(**kwargs)	A SMIRNOFF torsion type for improper torsions.

Continued on next page

Table 67 – continued from previous page

add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
check_parameter_compatibility(parameter_kwargs)	Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler
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([output_units, ...])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

create_force

__init__(***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

permit_cosmetic_attributes [bool] Whether to accept non-spec kwargs

****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_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
check_parameter_compatibility(parameter_kwargs)	Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler
create_force(system, topology, **kwargs)	
find_matches(entity)	Find the improper torsions in the topology/molecule matched by a parameter type.

Continued on next page

Table 68 – continued from previous page

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([output_units, ...])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects

class ImproperTorsionType(***kwargs*)

A SMIRNOFF torsion type for improper torsions.

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

Attributes

smirks

Methods

to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

to_dict(*return_cosmetic_attributes=False*)

Convert this ParameterType-derived object to dict. A unit-bearing attribute ('X') will be converted to two dict entries, one (['X'] containing the unitless value, and another (['X_unit']) containing a string representation of its unit.

Parameters

return_cosmetic_attributes [bool, optional. default = False] Whether to return non-spec attributes of this ParameterType

Returns

smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this ParameterType-derived object.

output_units [dict[str: simtk.unit.Unit]] A mapping from each simtk.unit.Quanitity-valued ParameterType attribute to the unit it was converted to during serialization.

find_matches(*entity*)

Find the improper torsions in the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology or openforcefield.topology.Molecule] Topology or molecule to search.

Returns

matches [ImproperDict[Tuple[int], ParameterType]] `matches[atom_indices]` is the `ParameterType` object matching the 4-tuple of atom indices in `entity`.

add_parameter(*parameter_kwargs*)
Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs [dict] The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

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.

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.

check_parameter_compatibility(*parameter_kwargs*)
Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler

Parameters

parameter_kwargs: dict The dict that will be used to construct the ParameterType

Raises

Raises a ValueError if the parameters are incompatible.

get_parameter(*parameter_attrs*)
Return the parameters in this ParameterHandler that match the parameter_attrs argument

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

list of ParameterType-derived objects A list of matching ParameterType-derived objects

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

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(*output_units=None*, *return_cosmetic_attributes=False*)
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

output_units [dict[str] A mapping from the ParameterType attribute name to the output unit its value should be converted to.

return_cosmetic_attributes [bool, optional. Default = False.] Whether to return 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**(***kwargs*)
Handle SMIRNOFF <vdW> tags

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

Attributes

combining_rules The combining_rules used to model van der Waals interactions

cutoff The cutoff used for long-range van der Waals interactions

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

method The method used to handle long-range van der Waals interactions

parameters The ParameterList that holds this ParameterHandler's parameter objects

potential The potential used to model van der Waals interactions

switch_width The switching width used for long-range van der Waals interactions

Methods

add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.

Continued on next page

Table 71 – continued from previous page

assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
check_parameter_compatibility(parameter_kwargs)	Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler
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(system, topology, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
to_dict([output_units, ...])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.
vdWType(**kwargs)	A SMIRNOFF vdWForce type.

create_force

__init__(***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

permit_cosmetic_attributes [bool] Whether to accept non-spec kwargs

****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_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
check_parameter_compatibility(parameter_kwargs)	Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler
create_force(system, topology, **kwargs)	
find_matches(entity)	Find the elements of the topology/molecule matched by a parameter type.

Continued on next page

Table 72 – continued from previous page

get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument
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([output_units, ...])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

combining_rules	The combining_rules used to model van der Waals interactions
cutoff	The cutoff used for long-range van der Waals interactions
known_kwargs	List of kwargs that can be parsed by the function.
method	The method used to handle long-range van der Waals interactions
parameters	The ParameterList that holds this ParameterHandler's parameter objects
potential	The potential used to model van der Waals interactions
switch_width	The switching width used for long-range van der Waals interactions

class vdWType(***kwargs*)

A SMIRNOFF vdWForce type.

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

Attributes

smirks

Methods

to_dict([return_cosmetic_attributes])	Convert this ParameterType-derived object to dict.

to_dict(*return_cosmetic_attributes=False*)

Convert this ParameterType-derived object to dict. A unit-bearing attribute ('X') will be converted to two dict entries, one (['X'] containing the unitless value, and another (['X_unit']) containing a string representation of its unit.

Parameters

return_cosmetic_attributes [bool, optional. default = False] Whether to return non-spec attributes of this ParameterType

Returns

smirnoff_dict [dict] The SMIRNOFF-compliant dict representation of this ParameterType-derived object.

output_units [dict[str: simtk.unit.Unit]] A mapping from each simtk.unit.Quanitity-valued ParameterType attribute to the unit it was converted to during serialization.

potential
The potential used to model van der Waals interactions

combining_rules
The combining_rules used to model van der Waals interactions

method
The method used to handle long-range van der Waals interactions

cutoff
The cutoff used for long-range van der Waals interactions

switch_width
The switching width used for long-range van der Waals interactions

check_handler_compatibility(*handler_kwargs*, *assume_missing_is_default=True*)
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. If no value is given for a field, it will be assumed to expect the ParameterHandler class default.

Parameters

handler_kwargs [dict] The kwargs that would be used to construct a ParameterHandler

assume_missing_is_default [bool] If True, will assume that parameters not specified in handler_kwargs would have been set to the default. Therefore, an exception is raised if the ParameterHandler is incompatible with the default value for a unspecified field.

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.

add_parameter(*parameter_kwargs*)
Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs [dict] The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

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.

check_parameter_compatibility(*parameter_kwargs*)

Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler

Parameters

parameter_kwargs: dict The dict that will be used to construct the ParameterType

Raises

Raises a ValueError if the parameters are incompatible.

find_matches(*entity*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology or openforcefield.topology.Molecule] Topology or molecule to search.

Returns

matches [ValenceDict[Tuple[int], ParameterType]] `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

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

list of ParameterType-derived objects A list of matching ParameterType-derived objects

known_kwargs

List of kwargs that can be parsed by the function.

parameters

The ParameterList that holds this ParameterHandler's parameter objects

to_dict(*output_units=None*, *return_cosmetic_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

output_units [dict[str] A mapping from the ParameterType attribute name to the output unit its value should be converted to.

return_cosmetic_attributes [bool, optional. Default = False.] Whether to return 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**(***kwargs*)

Handles SMIRNOFF <Electrostatics> tags.

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

Attributes

cutoff The cutoff used for long-range van der Waals interactions

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

method The method used to model long-range electrostatic interactions

parameters The ParameterList that holds this ParameterHandler's parameter objects

switch_width The switching width used for long-range electrostatics interactions

Methods

add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
check_parameter_compatibility(parameter_kwargs)	Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler
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([output_units, ...])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

create_force	

__init__(***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

permit_cosmetic_attributes [bool] Whether to accept non-spec kwargs

****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_parameter`(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
`assign_parameters`(topology, system)	Assign parameters for the given Topology to the specified System object.
`check_handler_compatibility`(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
`check_parameter_compatibility`(parameter_kwargs)	Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler
`create_force`(system, topology, **kwargs)	
`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`([output_units, ...])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

`cutoff`	The cutoff used for long-range van der Waals interactions
`known_kwargs`	List of kwargs that can be parsed by the function.
`method`	The method used to model long-range electrostatic interactions
`parameters`	The ParameterList that holds this ParameterHandler's parameter objects
`switch_width`	The switching width used for long-range electrostatics interactions

method
The method used to model long-range electrostatic interactions

cutoff
The cutoff used for long-range van der Waals interactions

switch_width
The switching width used for long-range electrostatics interactions

add_parameter(*parameter_kwargs*)
Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs [dict] The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

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.

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.

check_parameter_compatibility(*parameter_kwargs*)
Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler

Parameters

parameter_kwargs: dict The dict that will be used to construct the ParameterType

Raises

Raises a ValueError if the parameters are incompatible.

find_matches(*entity*)
Find the elements of the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology or openforcefield.topology.Molecule] Topology or molecule to search.

Returns

matches [ValenceDict[Tuple[int], ParameterType]] `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

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

list of ParameterType-derived objects A list of matching ParameterType-derived objects

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

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(*output_units=None*, *return_cosmetic_attributes=False*)
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

output_units [dict[str] A mapping from the ParameterType attribute name to the output unit its value should be converted to.

return_cosmetic_attributes [bool, optional. Default = False.] Whether to return 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**(***kwargs*)
Handle SMIRNOFF <ToolkitAM1BCC> tags

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

Attributes

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

parameters The ParameterList that holds this ParameterHandler's parameter objects

Methods

add_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_charge_from_molecules(molecule, ...)	Given an input molecule, checks against a list of molecules for an isomorphic match.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.

Continued on next page

Table 78 – continued from previous page

check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
check_parameter_compatibility(parameter_kwargs)	Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler
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(system, topology, **kwargs)	Allow the force to perform a a final post-processing pass on the System following parameter assignment, if needed.
to_dict([output_units, ...])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

create_force

__init__(***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

permit_cosmetic_attributes [bool] Whether to accept non-spec kwargs

****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_parameter(parameter_kwargs)	Add a parameter to the forcefield, ensuring all parameters are valid.
assign_charge_from_molecules(molecule, ...)	Given an input molecule, checks against a list of molecules for an isomorphic match.
assign_parameters(topology, system)	Assign parameters for the given Topology to the specified System object.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
check_parameter_compatibility(parameter_kwargs)	Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler
create_force(system, topology, **kwargs)	
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

Continued on next page

Table 79 – continued from previous page

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([output_units, ...])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects

check_handler_compatibility(*handler_kwargs*, *assume_missing_is_default=True*)

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. If no value is given for a field, it will be assumed to expect the ParameterHandler class default.

Parameters

handler_kwargs [dict] The kwargs that would be used to construct a ParameterHandler

assume_missing_is_default [bool] If True, will assume that parameters not specified in handler_kwargs would have been set to the default. Therefore, an exception is raised if the ParameterHandler is incompatible with the default value for a unspecified field.

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.

add_parameter(*parameter_kwargs*)
Add a parameter to the forcefield, ensuring all parameters are valid.

Parameters

parameter_kwargs [dict] The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor

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.

check_parameter_compatibility(*parameter_kwargs*)
Check to make sure that the fields requiring defined units are compatible with the required units for the Parameters handled by this ParameterHandler

Parameters

parameter_kwargs: dict The dict that will be used to construct the ParameterType

Raises

Raises a ValueError if the parameters are incompatible.

find_matches(*entity*)
Find the elements of the topology/molecule matched by a parameter type.

Parameters

entity [openforcefield.topology.Topology or openforcefield.topology.Molecule] Topology or molecule to search.

Returns

matches [ValenceDict[Tuple[int], ParameterType]] `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

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

list of ParameterType-derived objects A list of matching ParameterType-derived objects

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

parameters
The ParameterList that holds this ParameterHandler's parameter objects

to_dict(*output_units=None*, *return_cosmetic_attributes=False*)
Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters

output_units [dict[str] A mapping from the ParameterType attribute name to the output unit its value should be converted to.

return_cosmetic_attributes [bool, optional. Default = False.] Whether to return 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`
`XMLParameterIOHandler`

2.3 Utilities

2.3.1 Toolkit wrappers

The toolkit wrappers provide a simple uniform API for accessing minimal functionality of cheminformatics toolkits.

These toolkit wrappers are generally used through a `ToolkitRegistry`, which can be constructed with a desired precedence of toolkits:

```
from openforcefield.utils.toolkits import ToolkitRegistry
toolkit_registry = ToolkitRegistry()
toolkit_precedence = [OpenEyeToolkitWrapper, RDKitToolkitWrapper, AmberToolsToolkitWrapper]
[ toolkit_registry.register(toolkit) for toolkit in toolkit_precedence if toolkit.is_available() ]
```

Alternatively, the global toolkit registry (which will attempt to register any available toolkits) can be used:

```
from openforcefield.utils.toolkits import DEFAULT_TOOLKIT_REGISTRY as toolkit_registry
```

The toolkit wrappers can then be accessed through the registry:

```
molecule = Molecule.from_smiles('Cc1ccccc1')
smiles = toolkit_registry.call('to_smiles', molecule)
```

`ToolkitRegistry`	Registry for ToolkitWrapper objects
`ToolkitWrapper`	Toolkit wrapper base class.
`OpenEyeToolkitWrapper`	OpenEye toolkit wrapper
`RDKitToolkitWrapper`	RDKit toolkit wrapper
`AmberToolsToolkitWrapper`	AmberTools toolkit wrapper

openforcefield.utils.toolkits.ToolkitRegistry

class openforcefield.utils.toolkits.**ToolkitRegistry**(*register_imported_toolkit_wrappers=False*, *toolkit_precedence=None*, *exception_if_unavailable=True*)

Registry for ToolkitWrapper objects

Examples

Register toolkits in a specified order, skipping if unavailable

```
>>> from openforcefield.utils.toolkits import ToolkitRegistry
>>> toolkit_registry = ToolkitRegistry()
>>> toolkit_precedence = [OpenEyeToolkitWrapper, RDKitToolkitWrapper, AmberToolsToolkitWrapper]
>>> for toolkit in toolkit_precedence:
...     if toolkit.is_available():
...         toolkit_registry.register_toolkit(toolkit)
```

Register specified toolkits, raising an exception if one is unavailable

```
>>> toolkit_registry = ToolkitRegistry()
>>> toolkits = [OpenEyeToolkitWrapper, AmberToolsToolkitWrapper]
>>> for toolkit in toolkits:
...     toolkit_registry.register_toolkit(toolkit)
```

Register all available toolkits in arbitrary order

```
>>> from openforcefield.utils import all_subclasses
>>> toolkits = all_subclasses(ToolkitWrapper)
>>> for toolkit in toolkit_precedence:
...     if toolkit.is_available():
...         toolkit_registry.register_toolkit(toolkit)
```

Retrieve the global singleton toolkit registry, which is created when this module is imported from all available toolkits:

```
>>> from openforcefield.utils.toolkits import GLOBAL_TOOLKIT_REGISTRY as toolkit_registry
>>> available_toolkits = toolkit_registry.registered_toolkits
```

Warning: This API is experimental and subject to change.

Attributes

registered_toolkits List registered toolkits.

Methods

add_toolkit(toolkit_wrapper)	Append a ToolkitWrapper onto the list of toolkits in this ToolkitRegistry
call(method_name, *args, **kwargs)	Execute the requested method by attempting to use all registered toolkits in order of precedence.

Continued on next page

Table 83 – continued from previous page

register_toolkit(toolkit_wrapper[, ...])	Register the provided toolkit wrapper class, instantiating an object of it.
resolve(method_name)	Resolve the requested method name by checking all registered toolkits in order of precedence for one that provides the requested method.

__init__(*register_imported_toolkit_wrappers=False*, *toolkit_precedence=None*, *exception_if_unavailable=True*)
Create an empty toolkit registry.

Parameters

register_imported_toolkit_wrappers [bool, optional, default=False]

If True, will attempt to register all imported ToolkitWrapper subclasses that can be found, in no p
order.

toolkit_precedence [list, optional, default=None] List of toolkit wrapper classes, in order of desired precedence when performing molecule operations. If None, defaults to [OpenEyeToolkitWrapper, RDKitToolkitWrapper, AmberToolsToolkitWrapper].

exception_if_unavailable [bool, optional, default=True] If True, an exception will be raised if the toolkit is unavailable

Methods

__init__([...])	Create an empty toolkit registry.
add_toolkit(toolkit_wrapper)	Append a ToolkitWrapper onto the list of toolkits in this ToolkitRegistry
call(method_name, *args, **kwargs)	Execute the requested method by attempting to use all registered toolkits in order of precedence.
register_toolkit(toolkit_wrapper[, ...])	Register the provided toolkit wrapper class, instantiating an object of it.
resolve(method_name)	Resolve the requested method name by checking all registered toolkits in order of precedence for one that provides the requested method.

Attributes

registered_toolkits	List registered toolkits.

registered_toolkits
List registered toolkits.

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

Todo: Should this return a generator? Deep copies? Classes? Toolkit names?

Returns

toolkits [iterable of toolkit objects]

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.

Todo: This method should raise an exception if the toolkit is unavailable, unless an optional argument is specified that silently avoids registration of toolkits that are unavailable.

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

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

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(register_imported_toolkit_wrappers=True)
>>> method = toolkit_registry.resolve('to_smiles')
>>> smiles = method(molecule)
```

Todo: Is there a better way to figure out which toolkits implement given methods by introspection?

call(*method_name*, **args*, ***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

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(register_imported_toolkit_wrappers=True)
>>> 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 The name of the toolkit wrapped by this class.

Methods

from_file(filename, 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 this toolkit wrapper is available for use because the underlying toolkit can be found.
requires_toolkit()	
toolkit_is_available()	Check whether the corresponding toolkit can be imported

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

Methods

from_file(filename, 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 this toolkit wrapper is available for use because the underlying toolkit can be found.
requires_toolkit()	
toolkit_is_available()	Check whether the corresponding toolkit can be imported

Attributes

toolkit_file_read_formats	List of file formats that this toolkit can read.
toolkit_file_write_formats	List of file formats that this toolkit can write.
toolkit_installation_instructions	classmethod(function) -> method
toolkit_name	The name of the toolkit wrapped by this class.

toolkit_name
The name of the toolkit wrapped by this class.

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.

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.

static toolkit_is_available()
Check whether the corresponding toolkit can be imported

Note: This method call may be expensive.

Returns

is_installed [bool] True if corresponding toolkit is installed, False otherwise.

classmethod is_available()
Check whether this toolkit wrapper is available for use because the underlying toolkit can be found.

Note: This method caches the result of any costly checks for file paths or module imports.

Parameters

is_available [bool] True if toolkit is available for use, False otherwise

from_file(*filename*, *file_format*, *allow_undefined_stereo=False*)
Return an openforcefield.topology.Molecule from a file using this toolkit.

Parameters

filename [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.

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*)

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.

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 The name of the toolkit wrapped by this class.

Methods

compute_partial_charges(molecule[, ...])	Compute partial charges with OpenEye quacpac
compute_partial_charges_am1bcc(molecule)	Compute AM1BCC partial charges with OpenEye quacpac
compute_wiberg_bond_orders(molecule[, ...])	Update and store list of bond orders this molecule.
find_smarts_matches(molecule, smarts[, ...])	Find all SMARTS matches for the specified molecule, using the specified aromaticity model.
from_file(filename, 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_object(object)	If given an OEMol (or OEMol-derived object), this function will load it into an openforcefield.topology.molecule Otherwise, it will return False.
from_openeye(oemol[, allow_undefined_stereo])	Create a Molecule from an OpenEye molecule.
from_smiles(smiles[, hydrogens_are_explicit])	Create a Molecule from a SMILES string using the OpenEye toolkit.

Continued on next page

Table 89 – continued from previous page

generate_conformers(molecule[, ...])	Generate molecule conformers using OpenEye Omega.
is_available()	Check whether this toolkit wrapper is available for use because the underlying toolkit can be found.
requires_toolkit()	
to_file(molecule, outfile, outfile_format)	Writes an OpenFF Molecule to a file-like object
to_file_obj(molecule, file_obj, outfile_format)	Writes an OpenFF Molecule to a file-like object
to_openeye(molecule[, aromaticity_model])	Create an OpenEye molecule using the specified aromaticity model
to_smiles(molecule)	Uses the OpenEye toolkit to convert a Molecule into a SMILES string.
toolkit_is_available([oetools])	Check if the given OpenEye toolkit components are available.

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

Methods

compute_partial_charges(molecule[, ...])	Compute partial charges with OpenEye quacpac
compute_partial_charges_am1bcc(molecule)	Compute AM1BCC partial charges with OpenEye quacpac
compute_wiberg_bond_orders(molecule[, ...])	Update and store list of bond orders this molecule.
find_smarts_matches(molecule, smarts[, ...])	Find all SMARTS matches for the specified molecule, using the specified aromaticity model.
from_file(filename, 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_object(object)	If given an OEMol (or OEMol-derived object), this function will load it into an openforcefield.topology.molecule Otherwise, it will return False.
from_openeye(oemol[, allow_undefined_stereo])	Create a Molecule from an OpenEye molecule.
from_smiles(smiles[, hydrogens_are_explicit])	Create a Molecule from a SMILES string using the OpenEye toolkit.
generate_conformers(molecule[, ...])	Generate molecule conformers using OpenEye Omega.
is_available()	Check whether this toolkit wrapper is available for use because the underlying toolkit can be found.
requires_toolkit()	
to_file(molecule, outfile, outfile_format)	Writes an OpenFF Molecule to a file-like object
to_file_obj(molecule, file_obj, outfile_format)	Writes an OpenFF Molecule to a file-like object
to_openeye(molecule[, aromaticity_model])	Create an OpenEye molecule using the specified aromaticity model

Continued on next page

Table 90 – continued from previous page

to_smiles(molecule)	Uses the OpenEye toolkit to convert a Molecule into a SMILES string.
toolkit_is_available([oetools])	Check if the given OpenEye toolkit components are available.

Attributes

toolkit_file_read_formats	List of file formats that this toolkit can read.
toolkit_file_write_formats	List of file formats that this toolkit can write.
toolkit_installation_instructions	classmethod(function) -> method
toolkit_name	The name of the toolkit wrapped by this class.

static toolkit_is_available(*oetools=('oechem', 'oequacpac', 'oeiupac', 'oeomega')*)

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 given toolkits , `False` is returned.

Parameters

oetools [str or iterable of strings, optional, default=('oechem', 'oequacpac', 'oeiupac', 'oeomega')] Set of tools to check by their Python module name. Defaults to the complete set of tools supported by this function. Also accepts a single tool to check as a string instead of an iterable of length 1.

Returns

all_installed [bool] `True` if all tools in `oetools` are installed and licensed, `False` otherwise

classmethod is_available()

Check whether this toolkit wrapper is available for use because the underlying toolkit can be found.

Note: This method caches the result of any costly checks for file paths or module imports.

Parameters

is_available [bool] True if toolkit wrapper is available for use, False otherwise

from_object(*object*)

If given an OEMol (or OEMol-derived object), this function will load it into an openforcefield.topology.molecule Otherwise, it will return False.

Parameters

object [A molecule-like object] An object to by type-checked.

Returns

Molecule An openforcefield.topology.molecule Molecule.

Raises

NotImplementedError If the object could not be converted into a Molecule.

from_file(*filename*, *file_format*, *allow_undefined_stereo=False*)
Return an openforcefield.topology.Molecule from a file using this toolkit.

Parameters

filename [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.

Returns

molecules [list of Molecules] a list of Molecule objects is returned.

Raises

GAFFAtomTypeWarning If the loaded mol2 file possibly uses GAFF atom types, which are not supported.

from_file_obj(*file_obj*, *file_format*, *allow_undefined_stereo=False*)
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.

Returns

molecules [Molecule or list of Molecules] a list of Molecule objects is

to_file_obj(*molecule*, *file_obj*, *outfile_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

outfile_format The format for writing the molecule data

to_file(*molecule*, *outfile*, *outfile_format*)
Writes an OpenFF Molecule to a file-like object

Parameters

molecule [an OpenFF Molecule] The molecule to write

outfile The filename to write to

outfile_format The format for writing the molecule data

static from_openeye(*oemol*, *allow_undefined_stereo=False*)
Create a Molecule from an OpenEye molecule.

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.

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_filename
>>> ifs = oechem.oemolistream(get_data_filename('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

---

Todo:

- Should the aromaticity model be specified in some other way?

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)
```

static to_smiles(*molecule*)

Uses the OpenEye toolkit to convert a Molecule into a SMILES string.

Parameters

molecule [An openforcefield.topology.Molecule] The molecule to convert into a SMILES.

Returns

smiles [str] The SMILES of the input molecule.

from_smiles(*smiles*, *hydrogens_are_explicit=False*)

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

Returns

molecule [openforcefield.topology.Molecule] An openforcefield-style molecule.

generate_conformers(*molecule*, *num_conformers=1*, *clear_existing=True*)

Generate molecule conformers using OpenEye Omega.

Warning: This API is experimental and subject to change.

Todo:

- which parameters should we expose? (or can we implement a general system with **kwargs?)
- will the coordinates be returned in the OpenFF Molecule's own indexing system? Or is there a chance that

they'll get reindexed when we convert the input into an OEmol?

---

molecule [a `Molecule`] The molecule to generate conformers for.

num_conformers [int, default=1] The maximum number of conformers to generate.

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

compute_partial_charges(*molecule*, *quantum_chemical_method='AM1-BCC'*, *partial_charge_method='None'*)

Compute partial charges with OpenEye quacpac

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

Todo:

- Should the default be ELF?
- Can we expose more charge models?

Parameters

molecule [Molecule] Molecule for which partial charges are to be computed

charge_model [str, optional, default=None] The charge model to use. One of ['noop', 'mmff', 'mmff94', 'am1bcc', 'am1bccnosymspt', 'amber', 'amberff94', 'am1bccelf10'] If None, 'am1bcc' will be used.

Returns

charges [numpy.array of shape (natoms) of type float] The partial charges

compute_partial_charges_am1bcc(*molecule*)

Compute AM1BCC partial charges with OpenEye quacpac

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

Todo:

- Should the default be ELF?
- Can we expose more charge models?

Parameters

molecule [Molecule] Molecule for which partial charges are to be computed

Returns

charges [numpy.array of shape (natoms) of type float] The partial charges

compute_wiberg_bond_orders(*molecule*, *charge_model=None*)

Update and store list of bond orders this molecule. Can be used for initialization of bondorders list, or for updating bond orders in the list.

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

Parameters

molecule [openforcefield.topology.molecule Molecule] The molecule to assign wiberg bond orders to

charge_model [str, optional, default=None] The charge model to use. One of ['am1', 'pm3']. If None, 'am1' will be used.

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'] Aromaticity model to use during matching

.. note :: Currently, the only supported "aromaticity_model" is "OEAroModel_MDL"

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

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.

toolkit_name
The name of the toolkit wrapped by this class.

openforcefield.utils.toolkits.RDKitToolkitWrapper

class openforcefield.utils.toolkits.**RDKitToolkitWrapper**
RDKit toolkit wrapper

Warning: This API is experimental and subject to change.

Attributes

toolkit_file_read_formats List of file formats that this toolkit can read.

toolkit_file_write_formats List of file formats that this toolkit can write.

toolkit_installation_instructions classmethod(function) -> method

toolkit_name The name of the toolkit wrapped by this class.

Methods

find_smarts_matches(molecule, smarts[, ...])	Find all SMARTS matches for the specified molecule, using the specified aromaticity model.
from_file(filename, 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_object(object)	If given an rdchem.Mol (or rdchem.Mol-derived object), this function will load it into an openforcefield.topology.molecule.
from_rdkit(rdmol[, allow_undefined_stereo])	Create a Molecule from an RDKit molecule.
from_smiles(smiles[, hydrogens_are_explicit])	Create a Molecule from a SMILES string using the RDKit toolkit.
generate_conformers(molecule[, ...])	Generate molecule conformers using RDKit.
is_available()	Check whether toolkit is available for use.
requires_toolkit()	
to_file(molecule, outfile, outfile_format)	Writes an OpenFF Molecule to a file-like object
to_file_obj(molecule, file_obj, outfile_format)	Writes an OpenFF Molecule to a file-like object
to_rdkit(molecule[, aromaticity_model])	Create an RDKit molecule
to_smiles(molecule)	Uses the RDKit toolkit to convert a Molecule into a SMILES string.
toolkit_is_available()	Check whether the RDKit toolkit can be imported

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

Methods

find_smarts_matches(molecule, smarts[, ...])	Find all SMARTS matches for the specified molecule, using the specified aromaticity model.
from_file(filename, 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.

Continued on next page

Table 93 – continued from previous page

from_object(object)	If given an rdchem.Mol (or rdchem.Mol-derived object), this function will load it into an openforcefield.topology.molecule.
from_rdkit(rdmol[, allow_undefined_stereo])	Create a Molecule from an RDKit molecule.
from_smiles(smiles[, hydrogens_are_explicit])	Create a Molecule from a SMILES string using the RDKit toolkit.
generate_conformers(molecule[, ...])	Generate molecule conformers using RDKit.
is_available()	Check whether toolkit is available for use.
requires_toolkit()	
to_file(molecule, outfile, outfile_format)	Writes an OpenFF Molecule to a file-like object
to_file_obj(molecule, file_obj, outfile_format)	Writes an OpenFF Molecule to a file-like object
to_rdkit(molecule[, aromaticity_model])	Create an RDKit molecule
to_smiles(molecule)	Uses the RDKit toolkit to convert a Molecule into a SMILES string.
toolkit_is_available()	Check whether the RDKit toolkit can be imported

Attributes

toolkit_file_read_formats	List of file formats that this toolkit can read.
toolkit_file_write_formats	List of file formats that this toolkit can write.
toolkit_installation_instructions	classmethod(function) -> method
toolkit_name	The name of the toolkit wrapped by this class.

static toolkit_is_available()
Check whether the RDKit toolkit can be imported

Returns

is_installed [bool] True if RDKit is installed, False otherwise.

classmethod is_available()
Check whether toolkit is available for use.

Parameters

is_available [bool] True if toolkit is available for use, False otherwise

from_object(*object*)
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

object [A rdchem.Mol-derived object] An object to be type-checked and converted into a Molecule, if possible.

Returns

Molecule or False An openforcefield.topology.molecule Molecule.

Raises

NotImplementedError If the object could not be converted into a Molecule.

from_file(*filename*, *file_format*, *allow_undefined_stereo=False*)
Create an openforcefield.topology.Molecule from a file using this toolkit.

Parameters

filename [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.

Returns

molecules [iterable of Molecules] a list of Molecule objects is returned.

from_file_obj(*file_obj*, *file_format*, *allow_undefined_stereo=False*)

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.

Returns

molecules [Molecule or list of Molecules] a list of Molecule objects is returned.

to_file_obj(*molecule*, *file_obj*, *outfile_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

outfile_format The format for writing the molecule data

to_file(*molecule*, *outfile*, *outfile_format*)

Writes an OpenFF Molecule to a file-like object

Parameters

molecule [an OpenFF Molecule] The molecule to write

outfile The filename to write to

outfile_format The format for writing the molecule data

Returns

——

classmethod to_smiles(*molecule*)

Uses the RDKit toolkit to convert a Molecule into a SMILES string.

Parameters

molecule [An openforcefield.topology.Molecule] The molecule to convert into a SMILES.

Returns

smiles [str] The SMILES of the input molecule.

from_smiles(*smiles*, *hydrogens_are_explicit=False*)

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

Returns

molecule [openforcefield.topology.Molecule] An openforcefield-style molecule.

generate_conformers(*molecule*, *num_conformers=1*, *clear_existing=True*)

Generate molecule conformers using RDKit.

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

Todo:

- which parameters should we expose? (or can we implement a general system with **kwargs?)
- will the coordinates be returned in the OpenFF Molecule's own indexing system? Or is there a chance that they'll get reindexed when we convert the input into an RDMol?

---

molecule [a `Molecule`] The molecule to generate conformers for.

num_conformers [int, default=1] Maximum number of conformers to generate.

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

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

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.

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_filename
>>> rdmol = Chem.MolFromMolFile(get_data_filename('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()
```

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'] Aromaticity model to use during matching

.. note :: Currently, the only supported "aromaticity_model" is "OEAroModel_MDL"

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

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.

toolkit_name
The name of the toolkit wrapped by this class.

openforcefield.utils.toolkits.AmberToolsToolkitWrapper

class openforcefield.utils.toolkits.**AmberToolsToolkitWrapper**
AmberTools toolkit wrapper

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

Attributes

toolkit_file_read_formats List of file formats that this toolkit can read.

toolkit_file_write_formats List of file formats that this toolkit can write.

toolkit_installation_instructions classmethod(function) -> method

toolkit_name The name of the toolkit wrapped by this class.

Methods

compute_partial_charges(molecule[, charge_model])	Compute partial charges with AmberTools using antechamber/sqm

Continued on next page

Table 95 – continued from previous page

compute_partial_charges_am1bcc(molecule)	Compute partial charges with AmberTools using antechamber/sqm.
from_file(filename, 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 this toolkit wrapper is available for use because the underlying toolkit can be found.
requires_toolkit()	
toolkit_is_available()	Check whether the AmberTools toolkit is installed

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
compute_partial_charges(molecule[, charge_model])	Compute partial charges with AmberTools using antechamber/sqm
compute_partial_charges_am1bcc(molecule)	Compute partial charges with AmberTools using antechamber/sqm.
from_file(filename, 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 this toolkit wrapper is available for use because the underlying toolkit can be found.
requires_toolkit()	
toolkit_is_available()	Check whether the AmberTools toolkit is installed

Attributes

toolkit_file_read_formats	List of file formats that this toolkit can read.
toolkit_file_write_formats	List of file formats that this toolkit can write.
toolkit_installation_instructions	classmethod(function) -> method
toolkit_name	The name of the toolkit wrapped by this class.

static toolkit_is_available()
Check whether the AmberTools toolkit is installed

Returns

is_installed [bool] True if AmberTools is installed, False otherwise.

classmethod is_available()

Check whether this toolkit wrapper is available for use because the underlying toolkit can be found.

Note: This method caches the result of any costly checks for file paths or module imports.

Parameters

is_available [bool] True if toolkit wrapper is available for use, False otherwise

compute_partial_charges(*molecule*, *charge_model=None*)
Compute partial charges with AmberTools using antechamber/sqm

Warning: This API experimental and subject to change.

Todo:

- Do we want to also allow ESP/RESP charges?

Parameters

molecule [Molecule] Molecule for which partial charges are to be computed

charge_model [str, optional, default=None] The charge model to use. One of ['gas', 'mul', 'bcc']. If None, 'bcc' will be used.

Raises

ValueError if the requested charge method could not be handled

Notes

Currently only sdf file supported as input and mol2 as output https://github.com/choderalab/openmoltools/blob/master/openmoltools/packmol.py

compute_partial_charges_am1bcc(*molecule*)
Compute partial charges with AmberTools using antechamber/sqm. This will calculate AM1-BCC charges on the first conformer only.

Warning: This API experimental and subject to change.

Parameters

molecule [Molecule] Molecule for which partial charges are to be computed

Raises

ValueError if the requested charge method could not be handled

from_file(*filename*, *file_format*, *allow_undefined_stereo=False*)
Return an openforcefield.topology.Molecule from a file using this toolkit.

Parameters

filename [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.

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*)

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.

Returns

molecules [Molecule or list of Molecules] a list of Molecule objects is returned.

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

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.

toolkit_name
The name of the toolkit wrapped by this class.

2.3.2 Serialization support

Serializable	Mix-in to add serialization and deserialization support via JSON, YAML, BSON, TOML, MessagePack, and XML.

openforcefield.utils.serialization.Serializable

class openforcefield.utils.serialization.**Serializable**

Mix-in to add serialization and deserialization support via JSON, YAML, BSON, TOML, MessagePack, and XML.

For more information on these formats, see: JSON, BSON, YAML, TOML, MessagePack, and XML.

To use this mix-in, the class inheriting from this class must have implemented `to_dict()` and `from_dict()` methods that utilize dictionaries containing only serialiable Python objects.

Warning: The serialization/deserialiation schemes used here place some strict constraints on what kinds of `dict` objects can be serialized. No effort is made to add further protection to ensure serialization is possible. Use with caution.

Examples

Example class using `Serializable` mix-in:

```
>>> from openforcefield.utils.serialization import Serializable
>>> class Thing(Serializable):
...     def __init__(self, description):
...         self.description = description
...
...     def to_dict(self):
...         return { 'description' : self.description }
...
...     @classmethod
...     def from_dict(cls, d):
...         return cls(d['description'])
...
>>> # Create an example object
>>> thing = Thing('blorb')
```

Get JSON representation:

```
>>> json_thing = thing.to_json()
```

Reconstruct an object from its JSON representation:

```
>>> thing_from_json = Thing.from_json(json_thing)
```

Get BSON representation:

```
>>> bson_thing = thing.to_bson()
```

Reconstruct an object from its BSON representation:

```
>>> thing_from_bson = Thing.from_bson(bson_thing)
```

Get YAML representation:

```
>>> yaml_thing = thing.to_yaml()
```

Reconstruct an object from its YAML representation:

```
>>> thing_from_yaml = Thing.from_yaml(yaml_thing)
```

Get MessagePack representation:

```
>>> messagepack_thing = thing.to_messagepack()
```

Reconstruct an object from its MessagePack representation:

```
>>> thing_from_messagepack = Thing.from_messagepack(messagepack_thing)
```

Get XML representation:

```
>>> xml_thing = thing.to_xml()
```

Methods

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

from_dict	
to_dict	

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

Methods

from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(d)	
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
to_bson()	Return a BSON serialized representation.
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 Structure tools

Tools for manipulating molecules and structures

Todo: These methods are deprecated and will be removed. We recommend that no new code makes use of these functions.

generateSMIRNOFFStructure	Given an OpenEye molecule (oechem.OEMol), create an OpenMM System and use to generate a ParmEd structure using the SMIRNOFF forcefield parameters.
generateProteinStructure	Given an OpenMM PDBFile, create the OpenMM System of the protein using OpenMM's ForceField and then generate the parametrized ParmEd Structure of the protein.
combinePositions	Loops through the positions from the ParmEd structures of the protein and ligand, divides by unit.angstroms which will ensure both positions arrays are in the same units.
mergeStructure	Combines the parametrized ParmEd structures of the protein and ligand to create the Structure for the protein:ligand complex, while retaining the SMIRNOFF parameters on the ligand.
generateTopologyFromOEMol	Generate an OpenMM Topology object from an OEMol molecule.
get_testdata_filename	Get the full path to one of the reference files in testsystems.
normalize_molecules	Normalize all molecules in specified set.
read_molecules	Read molecules from an OpenEye-supported file.
setPositionsInOEMol	Set the positions in an OEMol using a position array with units from simtk.unit, i.e.
extractPositionsFromOEMol	Get the positions from an OEMol and return in a position array with units via simtk.unit, i.e.
read_typelist	Read a parameter type or decorator list from a file.
positions_from_oemol	Extract OpenMM positions from OEMol.
check_energy_is_finite	Check the potential energy is not NaN.
get_energy	Return the potential energy.
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.
getMolParamIDToAtomIndex	Take a Molecule and a SMIRNOFF forcefield object and return a dictionary, keyed by parameter ID, where each entry is a tuple of (smirks, [[atom1, . . .
merge_system	Merge two given OpenMM systems.
save_system_to_amber	Save an OpenMM System, with provided topology and positions, to AMBER prmtop and coordinate files.

Continued on next page

Table 101 – continued from previous page

save_system_to_gromacs	Save an OpenMM System, with provided topology and positions, to AMBER prmtop and coordinate files.

openforcefield.utils.structure.generateSMIRNOFFStructure

openforcefield.utils.structure.**generateSMIRNOFFStructure**(*oemol*)

Given an OpenEye molecule (oechem.OEMol), create an OpenMM System and use to generate a ParmEd structure using the SMIRNOFF forcefield parameters.

Parameters

oemol [openeye.oechem.OEMol] OpenEye molecule

Returns

molecule_structure [parmed.Structure] The resulting Structure

openforcefield.utils.structure.generateProteinStructure

openforcefield.utils.structure.**generateProteinStructure**(*proteinpdb*, *protein_forcefield='amber99sbildn.xml'*, *solvent_forcefield='tip3p.xml'*)

Given an OpenMM PDBFile, create the OpenMM System of the protein using OpenMM's ForceField and then generate the parametrized ParmEd Structure of the protein.

Parameters

proteinpdb [simtk.openmm.app.PDBFile object,] Loaded PDBFile object of the protein.

protein_forcefield [xml file, default='amber99sbildn.xml'] Forcefield parameters for protein

solvent_forcefield [xml file, default='tip3p.xml'] Forcefield parameters for solvent

Returns

solv_structure [parmed.structure.Structure] The parameterized Structure of the protein with solvent molecules. (No ligand).

openforcefield.utils.structure.combinePositions

openforcefield.utils.structure.**combinePositions**(*proteinPositions*, *molPositions*)

Loops through the positions from the ParmEd structures of the protein and ligand, divides by unit.angstroms which will ensure both positions arrays are in the same units.

Parameters

proteinPositions [list of 3-element Quantity tuples.] Positions list taken directly from the protein Structure.

molPositions [list of 3-element Quantity tuples.] Positions list taken directly from the molecule Structure.

Returns

positions [list of 3-element Quantity tuples.] ex. unit.Quantity(positions, positions_unit) Combined positions of the protein and molecule Structures.

openforcefield.utils.structure.mergeStructure

openforcefield.utils.structure.**mergeStructure**(*proteinStructure*, *molStructure*)

Combines the parametrized ParmEd structures of the protein and ligand to create the Structure for the protein:ligand complex, while retaining the SMIRNOFF parameters on the ligand. Preserves positions and box vectors. (Not as easily achieved using native OpenMM tools).

Parameters

proteinStructure [parmed.structure.Structure] The parametrized structure of the protein.

moleculeStructure [parmed.structure.Structure] The parametrized structure of the ligand.

Returns

structure [parmed.structure.Structure] The parametrized structure of the protein:ligand complex.

openforcefield.utils.structure.generateTopologyFromOEMol

openforcefield.utils.structure.**generateTopologyFromOEMol**(*molecule*)

Generate an OpenMM Topology object from an OEMol molecule.

Parameters

molecule [openeye.oechem.OEMol] The molecule from which a Topology object is to be generated.

Returns

topology [simtk.openmm.app.Topology] The Topology object generated from *molecule*.

openforcefield.utils.structure.get_testdata_filename

openforcefield.utils.structure.**get_testdata_filename**(*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.structure.normalize_molecules

openforcefield.utils.structure.**normalize_molecules**(*molecules*)

Normalize all molecules in specified set.

ARGUMENTS

molecules (list of OEMol) - molecules to be normalized (in place)

openforcefield.utils.structure.read_molecules

openforcefield.utils.structure.**read_molecules**(*filename*, *verbose=True*)
Read molecules from an OpenEye-supported file.

Parameters

filename [str] Filename from which molecules are to be read (e.g. mol2, sdf)

Returns

molecules [list of OEMol] List of molecules read from file

openforcefield.utils.structure.setPositionsInOEMol

openforcefield.utils.structure.**setPositionsInOEMol**(*oemol*, *positions*)
Set the positions in an OEMol using a position array with units from simtk.unit, i.e. from OpenMM. Atoms must have same order.

oemol [OEMol] OpenEye molecule

positions [Nx3 array] Unit-bearing via simtk.unit Nx3 array of coordinates

openforcefield.utils.structure.extractPositionsFromOEMol

openforcefield.utils.structure.**extractPositionsFromOEMol**(*oemol*)
Get the positions from an OEMol and return in a position array with units via simtk.unit, i.e. foramtted for OpenMM. Adapted from choderalab/openmoltools test function extractPositionsFromOEMOL

openforcefield.utils.structure.read_typelist

openforcefield.utils.structure.**read_typelist**(*filename*)
Read a parameter type or decorator list from a file. Lines in these files have the format "SMARTS/SMIRKS shorthand" lines beginning with '%' are ignored

Parameters

filename [str] Path and name of file to be read Could be file in openforcefield/data/

Returns

typelist [list of tuples] Typelist[i] is element i of the typelist in format (smarts, shorthand)

openforcefield.utils.structure.positions_from_oemol

openforcefield.utils.structure.**positions_from_oemol**(*mol*)
Extract OpenMM positions from OEMol.

Parameters

mol [oechem.openeye.OEMol] OpenEye molecule from which to extract coordinates.

Returns

positions [simtk.unit.Quantity of dimension (nparticles,3)]

openforcefield.utils.structure.check_energy_is_finite

openforcefield.utils.structure.**check_energy_is_finite**(*system*, *positions*)
Check the potential energy is not NaN.

Parameters

system [simtk.openmm.System] The system to check

positions [simtk.unit.Quantity of dimension (natoms,3) with units of length] The positions to use

openforcefield.utils.structure.get_energy

openforcefield.utils.structure.**get_energy**(*system*, *positions*)
Return the potential energy.

Parameters

system [simtk.openmm.System] The system to check

positions [simtk.unit.Quantity of dimension (natoms,3) with units of length] The positions to use

Returns

———

energy

openforcefield.utils.structure.get_molecule_parameterIDs

openforcefield.utils.structure.**get_molecule_parameterIDs**(*molecules*, *forcefield*)
Process a list of molecules with a specified SMIRNOFF ffxml file and determine which parameters are used by which molecules, returning collated results.

Parameters

molecules [list of openforcefield.topology.Molecule] List of molecules (with explicit hydrogens) to parse

forcefield [openforcefield.typing.engines.smirnoff.ForceField] The ForceField to apply

Returns

parameters_by_molecule [dict] Parameter IDs used in each molecule, keyed by isomeric SMILES generated from provided OEMols. Each entry in the dict is a list which does not necessarily have unique entries; i.e. parameter IDs which are used more than once will occur multiple times.

parameters_by_ID [dict] Molecules in which each parameter ID occur, keyed by parameter ID. Each entry in the dict is a set of isomeric SMILES for molecules in which that parameter occurs. No frequency information is stored.

openforcefield.utils.structure.getMolParamIDToAtomIndex

openforcefield.utils.structure.**getMolParamIDToAtomIndex**(*molecule*, *forcefield*)
Take a Molecule and a SMIRNOFF forcefield object and return a dictionary, keyed by parameter ID, where each entry is a tuple of (smirks, [[atom1, . . . atomN], [atom1, . . . atomN]) giving the SMIRKS

corresponding to that parameter ID and a list of the atom groups in that molecule that parameter is applied to.

Parameters

molecule [openforcefield.topology.Molecule] Molecule to investigate

forcefield [ForceField] SMIRNOFF ForceField object (obtained from an ffxml via ForceField(ffxml)) containing FF of interest.

Returns

param_usage [dictionary] Dictionary, keyed by parameter ID, where each entry is a tuple of (smirks, [[atom1, ... atomN], [atom1, ... atomN]) giving the SMIRKS corresponding to that parameter ID and a list of the atom groups in that molecule that parameter is applied to.

openforcefield.utils.structure.merge_system

openforcefield.utils.structure.**merge_system**(*topology0*, *topology1*, *system0*, *system1*, *positions0*, *positions1*, *label0='AMBER system'*, *label1='SMIRNOFF system'*, *verbose=True*)

Merge two given OpenMM systems. Returns the merged OpenMM System.

Parameters

topology0 [OpenMM Topology] Topology of first system (i.e. a protein)

topology1 [OpenMM Topology] Topology of second system (i.e. a ligand)

system0 [OpenMM System] First system for merging (usually from AMBER)

system1 [OpenMM System] Second system for merging (usually from SMIRNOFF)

positions0 [simtk.unit.Quantity wrapped] Positions to use for energy evaluation comparison

positions1 (optional) [simtk.unit.Quantity wrapped (optional)] Positions to use for second OpenMM system

label0 (optional) [str] String labeling system0 for output. Default, "AMBER system"

label1 (optional) [str] String labeling system1 for output. Default, "SMIRNOFF system"

verbose (optional) [bool] Print out info on topologies, True/False (default True)

Returns

topology [OpenMM Topology]

system [OpenMM System]

positions: unit.Quantity position array

openforcefield.utils.structure.save_system_to_amber

openforcefield.utils.structure.**save_system_to_amber**(*openmm_topology*, *system*, *positions*, *prmtop*, *inpcrd*)

Save an OpenMM System, with provided topology and positions, to AMBER prmtop and coordinate files.

Parameters

openmm_topology [OpenMM Topology] Topology of the system to be saved, perhaps as loaded from a PDB file or similar.

system [OpenMM System] Parameterized System to be saved, containing components represented by Topology

positions [unit.Quantity position array] Position array containing positions of atoms in topology/system

prmtop [filename] AMBER parameter file name to write

inpcrd [filename] AMBER coordinate file name (ASCII crd format) to write

openforcefield.utils.structure.save_system_to_gromacs

openforcefield.utils.structure.**save_system_to_gromacs**(*openmm_topology*, *system*, *positions*, *top*, *gro*)

Save an OpenMM System, with provided topology and positions, to AMBER prmtop and coordinate files.

Parameters

openmm_topology [OpenMM Topology] Topology of the system to be saved, perhaps as loaded from a PDB file or similar.

system [OpenMM System] Parameterized System to be saved, containing components represented by Topology

positions [unit.Quantity position array] Position array containing positions of atoms in topology/system

top [filename] GROMACS topology file name to write

gro [filename] GROMACS coordinate file name (.gro format) to write

2.3.4 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.
temporary_directory	Context for safe creation of temporary directories.
get_data_filename	Get the full path to one of the reference files in test-systems.

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

Examples

```
>>> dir_path = '/tmp'
>>> with temporary_cd(dir_path):
...     pass  # do something in dir_path
```

openforcefield.utils.utils.temporary_directory

openforcefield.utils.utils.**temporary_directory**()
Context for safe creation of temporary directories.

openforcefield.utils.utils.get_data_filename

openforcefield.utils.utils.**get_data_filename**(*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).

Index

Symbols

A

B

C

E

F

G

I

K

M

N

O

P

R

S

T

V