foyer Documentation

Release 0.12.1

Author

Jan 12, 2024

GETTING STARTED

1 Overview 3

2 Resources 5

3 Citation 7

4 Installation 9

5 Example 11

6 Getting started? 13

7 Credits 15

8 Table of Contents 17
- 8.1 Installation 17
- 8.2 Using foyer with Docker 18
- 8.3 Quickstart 20
- 8.4 SMARTS-based atomtyping 22
- 8.5 Parameter definitions 23
- 8.6 Applying a force field 24
- 8.7 Usage Examples 25
- 8.8 Examples from Foyer paper 26
- 8.9 Units 28
- 8.10 Validation of force field files 28
- 8.11 Forcefield Class 29
- 8.12 Citing foyer 32
- 8.13 License 33

Index 35

A package for atom-typing as well as applying and disseminating force fields

CHAPTER

ONE

OVERVIEW

Foyer is an open-source Python tool that provides a framework for the application and dissemination of classical molecular modeling force fields. Importantly, it enables users to define and apply atom-typing rules in a format that is simultaneously human- and machine-readable. A primary goal of **foyer** is to eliminate ambiguity in the atom-typing and force field application steps of molecular simulations in order to improve reproducibility. Foyer force fields are defined in an XML format derived from the OpenMM XML specification. SMARTS strings are used to define the chemical context of each atom type and "overrides" are used to define clear precedence of different atom types. **Foyer** is designed to be compatible with the other tools in the Molecular Simulation Design Framework (MoSDeF) ecosystem.

CHAPTER

TWO

RESOURCES

- *Installation guide*: Instructions for installing foyer.
- *Quickstart*: A brief introduction to foyer.
- MoSDeF: Learn more about the **Mo**lecular **S**imulation **De**sign **F**ramework.
- Foyer paper: The journal article describing foyer.
- GitHub repository: Download the source code or contribute to the development of foyer.
- Issue Tracker: Report issues and request features.

CHAPTER

THREE

CITATION

If you use foyer in your research, please cite the foyer paper. See *here* for details.

CHAPTER

FOUR

INSTALLATION

Complete installation instructions are *here*. A conda installation is available:

```
conda create --name foyer foyer -c conda-forge
```

CHAPTER

FIVE

EXAMPLE

Annotate an OpenMM .xml force field file with SMARTS-based atomtypes:

```xml
<ForceField>
 <AtomTypes>
  <Type name="opls_135" class="CT" element="C" mass="12.01100" def="[C;X4](C)(H)(H)H"␣
↪desc="alkane CH3"/>
  <Type name="opls_140" class="HC" element="H" mass="1.00800"  def="H[C;X4]" desc=
↪"alkane H"/>
 </AtomTypes>
</ForceField>
```

Apply the forcefield to arbitrary chemical topologies. We currently support:

- OpenMM.Topology
- ParmEd.Structure
- mBuild.Compound

```python
from foyer import Forcefield
import parmed as pmd

untyped_ethane = pmd.load_file('ethane.mol2', structure=True)
oplsaa = Forcefield(forcefield_files='oplsaa.xml')
ethane = oplsaa.apply(untyped_ethane)

# Save to any format supported by ParmEd
ethane.save('ethane.top')
ethane.save('ethane.gro')
```

CHAPTER

SIX

GETTING STARTED?

Check out our example template for disseminating force fields: https://github.com/mosdef-hub/forcefield_template

CHAPTER SEVEN

CREDITS

This material is based upon work supported by the National Science Foundation under grants NSF ACI-1047828 and NSF ACI-1535150. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

CHAPTER

EIGHT

TABLE OF CONTENTS

8.1 Installation

For most users we recommend a conda installation:

```
conda install -c conda-forge -c omnia foyer
```

If you wish to install from source, you can use the following commands:

```
git clone https://github.com/mosdef-hub/foyer.git
cd foyer
conda env create -f environment.yml
conda activate foyer
pip install .
```

If you are using windows, you should use `environment-win.yml` rather than `environment.yml`.

If you plan on contributing to the development of foyer, we recommend you create an editable installation with all the required dependencies:

```
git clone https://github.com/mosdef-hub/foyer.git
cd foyer
conda env create -f environment-dev.yml
conda activate foyer-dev
pip install -e .
```

8.1.1 Install pre-commit

We use [pre-commit](https://pre-commit.com/) to automatically handle our code formatting and this package is included in the dev environment. With the `foyer-dev` conda environment active, pre-commit can be installed locally as a git hook by running:

```
$ pre-commit install
```

And (optional) all files can be checked by running:

```
$ pre-commit run --all-files
```

8.1.2 Supported Python Versions

Python 3.6 and 3.7 are officially supported, including testing during development and packaging. Other Python versions, such as 3.8 and 3.5 and older, may successfully build and function but no guarantee is made.

8.1.3 Testing your installation

foyer uses `py.test` for unit testing. To run them simply type run the following while in the base directory:

```
$ conda install pytest
$ py.test -v
```

8.1.4 Building the documentation

foyer uses sphinx to build its documentation. To build the docs locally, run the following while in the `docs` directory:

```
$ conda env create -f docs-env.yml
$ conda activate foyer-docs
$ make html
```

8.2 Using foyer with Docker

As much of scientific software development happens in unix platforms, to avoid the quirks of development dependent on system you use, a recommended way is to use docker or other containerization technologies. This section is a how to guide on using `foyer` with docker.

8.2.1 Prerequisites

A docker installation in your machine. Follow this link to get a docker installation working on your machine. If you are not familiar with docker and want to get started with docker, the Internet is full of good tutorials like the ones here and here.

8.2.2 Quick Start

After you have a working docker installation, please use the following command to use run a jupyter-notebook with all the dependencies for *foyer* installed:

```
$ docker pull mosdef/foyer:latest
$ docker run -it --name foyer -p 8888:8888 mosdef/foyer:latest
```

If no command is provided to the container (as above), the container starts a `jupyter-notebook` at the container location `/home/anaconda/data`. Then, the notebook can be accessed by copying and pasting the notebook URL into a web browser on your computer. When finished with the session, you can use *Ctr`+`C* and follow instruction to exit the notebook as usual. The docker container will exit upon notebook shutdown.

Warning: Containers by nature are ephemeral, so filesystem changes (e.g., adding a new notebook) only persist until the end of the container's lifecycle. If the container is removed, any changes or code addition will not persist. See the section below for persistent data.

Note: The `-it` flags connect your keyboard to the terminal running in the container. You may run the prior command without those flags, but be aware that the container will not respond to any keyboard input. In that case, you would need to use the `docker ps` and `docker kill` commands to shut down the container.

8.2.3 Persisting User Volumes

If you will be using *foyer* from a docker container, a recommended way is to mount what are called user volumes in the container. User volumes will provide a way to persist all filesystem/code additions made to a container regardless of the container lifecycle. For example, you might want to create a directory called *foyer-notebooks* in your local system, which will store all your *foyer* notebooks/code. In order to make that accessible to the container(where the notebooks will be created/edited), use the following steps:

```
$ mkdir -p /path/to/foyer-notebooks
$ cd /path/to/foyer-notebooks
$ docker run -it --name foyer --mount type=bind,source=$(pwd),target=/home/anaconda/data␣
↪-p 8888:8888 mosdef/foyer:latest
```

You can easily mount a different directory from your local machine by changing `source=$(pwd)` to `source=/path/to/my/favorite/directory`.

Note: The `--mount` flag mounts a volume into the docker container. Here we use a `bind` mount to bind the current directory on our local filesystem to the `/home/anaconda/data` location in the container. The files you see in the `jupyter-notebook` browser window are those that exist on your local machine.

Warning: If you are using the container with jupyter notebooks you should use the `/home/anaconda/data` location as the mount point inside the container; this is the default notebook directory.

Jupyter notebooks are a great way to explore new software and prototype code. However, when it comes time for production sciences, it is often better to work with python scripts. In order to execute a python script (`example.py`) that exists in the current working directory of your local machine, run:

```
$ docker run --mount type=bind,source=$(pwd),target=/home/anaconda/data mosdef/foyer:
↪latest "python data/test.py"
```

Note that once again we are `bind` mounting the current working directory to `/home/anaconda/data`. The command we pass to the container is `python data/test.py`. Note the prefix `data/` to the script; this is because we enter the container in the home folder (`/home/anaconda`), but our script is located under `/home/anaconda/data`.

Warning: Do not bind mount to `target=/home/anaconda`. This will cause errors.

If you don't want a Jupyter notebook, but just want a Python interpreter, you can run:

If you don't need access to any local data, you can of course drop the `--mount` command:

8.2.4 Cleaning Up

You can remove the created container by using the following command:

```
$ docker container rm foyer
```

Note: Instead of using *latest*, you can use the image `mosdef/foyer:stable` for most recent stable release of *foyer* and run the tutorials.

8.3 Quickstart

Foyer is distributed with partial support for the OPLS-AA force field. First, we will load a PDB file (`here`) with `parmed` and then load the OPLS-AA force field with foyer.

```
import foyer
import parmed

mol = parmed.load("ethane.pdb")
ff = foyer.forcefield.load_OPLSAA()

mol_ff = ff.apply(mol)

mol_ff.save("ethane.gro")
mol_ff.save("ethane.top")
```

The first step loads `ethane.pdb`. `mol` is a `parmed.Structure`. Next, we load the OPLS force field as a `foyer.forcefield` object. Then, we apply the force field to the molecule. `mol_ff` is also a `parmed.Structure`, but it now contains all of the force field parameters (Lennard-Jones parameters, partial charges, bond, angle, dihedral parameters). Finally, we save the parameterized structure to input formats for GROMACS. In this example, `ff` has the force field parameters for the OPLS-AA force field. You can view the XML file with all of the parameters here. As you can see from the XML file, SMARTS strings have been added for many of the OPLS-AA atom types. You should always verify that the atom-typing is performed correctly for a single molecule before applying the force field to your entire system. If you wish to add supported molecules to the OPLS-AA force field, you can view this issue

The real power of **foyer** is to build and distribute XML files with your own parameter sets. Here we show an example of an XML file for pentafluoroethane. This is a custom parameter set taken from this paper.

```
<ForceField name="example_custom" version="0.0.1">
 <AtomTypes>
  <Type name="C1" class="c3" element="C" mass="12.011" def="C(C)(H)(F)(F)" desc="carbon␣
↪bonded to 2 Fs, a H, and another carbon"/>
  <Type name="C2" class="c3" element="C" mass="12.011" def="C(C)(F)(F)(F)" desc="carbon␣
↪bonded to 3 Fs and another carbon"/>
  <Type name="F1" class="f" element="F" mass="18.998" def="FC(C)(F)H" desc="F bonded to␣
↪C1"/>
  <Type name="F2" class="f" element="F" mass="18.998" def="FC(C)(F)F" desc="F bonded to␣
↪C2"/>
  <Type name="H1" class="h2" element="H" mass="1.008" def="H(C)" desc="single H bonded␣
↪to C1"/>
 </AtomTypes>
 <HarmonicBondForce>
```

(continues on next page)

(continued from previous page)

```
  <Bond class1="c3" class2="c3" length="0.15375" k="251793.12"/>
  <Bond class1="c3" class2="f" length="0.13497" k="298653.92"/>
  <Bond class1="c3" class2="h2" length="0.10961" k="277566.56"/>
 </HarmonicBondForce>
 <HarmonicAngleForce>
  <Angle class1="c3" class2="c3" class3="f" angle="1.9065976748786053" k="553.1248"/>
  <Angle class1="c3" class2="c3" class3="h2" angle="1.9237019015481498" k="386.6016"/>
  <Angle class1="f" class2="c3" class3="f" angle="1.8737854849411122" k="593.2912"/>
  <Angle class1="f" class2="c3" class3="h2" angle="1.898743693244631" k="427.6048"/>
 </HarmonicAngleForce>
 <PeriodicTorsionForce>
  <Proper class1="f" class2="c3" class3="c3" class4="f" periodicity1="3" k1="0.0" phase1=
↪"0.0" periodicity2="1" k2="5.0208" phase2="3.141592653589793"/>
  <Proper class1="" class2="c3" class3="c3" class4="" periodicity1="3" k1="0.
↪6508444444444444" phase1="0.0"/>
 </PeriodicTorsionForce>
 <NonbondedForce coulomb14scale="0.833333" lj14scale="0.5">
  <Atom type="C1" charge="0.224067"  sigma="0.371084" epsilon="0.304665"/>
  <Atom type="C2" charge="0.500886"  sigma="0.393872" epsilon="0.222541"/>
  <Atom type="F1" charge="-0.167131" sigma="0.298239" epsilon="0.208221"/>
  <Atom type="F2" charge="-0.170758" sigma="0.276783" epsilon="0.237635"/>
  <Atom type="H1" charge="0.121583"  sigma="0.264229" epsilon="0.071381"/>
 </NonbondedForce>
</ForceField>
```

The SMARTS strings are defined in the `<AtomTypes>` section. The bond, angle, and dihedral parameters are defined in the following sections. This time we load a `.gro` file for HFC-125 (`here`), load our custom force field XML file (`here`), and then apply the force field parameters.

```
import foyer
import parmed

mol = parmed.load("hfc125.gro")
ff = foyer.ForceField("ff_custom.xml")

mol_ff = ff.apply(mol)

    mol_ff.save("hfc125.top")
```

Foyer can be used to save input files for any simulation engine supported by `parmed`. If you also install `mbuild`, then a variety of other simulation engines are also supported.

8.4 SMARTS-based atomtyping

Foyer allows users to describe atomtypes using a modified version of SMARTS You may already be familiar with SMILES representations for describing chemical structures. SMARTS is a straightforward extension of this notation.

8.4.1 Basic usage

Consider the following example defining the OPLS-AA atomtypes for a methyl group carbon and its hydrogen atoms:

```
<ForceField>
 <AtomTypes>
  <Type name="opls_135" class="CT" element="C" mass="12.01100" def="[C;X4](C)(H)(H)H"␣
↪desc="alkane CH3"/>
  <Type name="opls_140" class="HC" element="H" mass="1.00800"  def="H[C;X4]" desc=
↪"alkane H"/>
 </AtomTypes>
</ForceField>
```

This `.xml` format is an extension of the OpenMM force field format The above example utilizes two additional `.xml` attributes supported by foyer: `def` and `desc`. The atomtype that we are attempting to match is always the **first** token in the SMARTS string, in the above example, `[C;X4]` and `H`. The `opls_135` (methyl group carbon) is defined by a SMARTS string indicated a carbon with 4 bonds, a carbon neighbor and 3 hydrogen neighbors. The `opls_140` (alkane hydrogen) is defined simply as a hydrogen atom bonded to a carbon with 4 bonds.

8.4.2 Overriding atomtypes

When multiple atomtype definitions can apply to a given atom, we must establish precedence between those definitions. Many other atomtypers determine rule precedence by placing more specific rules first and evaluate those in sequence, breaking out of the loop as soon as a match is found.

While this approach works, it becomes more challenging to maintain the correct ordering of rules as the number of atomtypes grows. Foyer iteratively runs all rules on all atoms and each atom maintains a whitelist (rules that apply) and a blacklist (rules that have been superseded by another rule). The set difference between the white- and blacklists yields the correct atomtype if the force field is implemented correctly.

We can add a rule to a blacklist using the `overrides` attribute in the `.xml` file. To illustrate an example where overriding can be used consider the following types describing alkenes and benzene:

```
<ForceField>
 <AtomTypes>
  <Type name="opls_141" class="CM" element="C" mass="12.01100" def="[C;X3](C)(C)C" desc=
↪"alkene C (R2-C=)"/>
  <Type name="opls_142" class="CM" element="C" mass="12.01100" def="[C;X3](C)(C)H" desc=
↪"alkene C (RH-C=)"/>
  <Type name="opls_144" class="HC" element="H" mass="1.00800"  def="[H][C;X3]" desc=
↪"alkene H"/>
  <Type name="opls_145" class="CA" element="C" mass="12.01100" def="[C;X3;r6]1[C;X3;
↪r6][C;X3;r6][C;X3;r6][C;X3;r6][C;X3;r6]1" overrides="opls_141,opls_142"/>
  <Type name="opls_146" class="HA" element="H" mass="1.00800"  def="[H][C;%opls_145]"␣
↪overrides="opls_144" desc="benzene H"/>
 </AtomTypes>
</ForceField>
```

If we're atomtyping a benzene molecule, the carbon atoms will match the SMARTS patterns for both `opls_142` and `opls_145`. Without the `overrides` attribute, foyer will notify you that multiple atomtypes were found for each carbon. Providing the `overrides` indicates that if the `opls_145` pattern matches, it should supersede the specified rules.

8.4.3 Supported SMARTS Grammar

We currently do not (yet) support all of SMARTS' features. Here we're keeping track of which portions are supported.

8.5 Parameter definitions

Parameter definitions within force field XMLs follow the same conventions as defined in the OpenMM documentation. Currently, only certain functional forms for molecular forces are supported, while future developments are expected to allow Foyer to support any desired functional form, including reactive and tabulated potentials. The currently supported functional forms for molecular forces are:

- **Nonbonded**
  - Lennard-Jones (12-6)
- **Bonds**
  - Harmonic
- **Angles**
  - Harmonic
- **Torsions (proper)**
  - Periodic
  - Ryckaert-Bellemans
- **Torsions (improper)**
  - Periodic

Definitions for each molecular force follow the OpenMM standard.

The harmonic bond potential is defined as

$$E = \frac{1}{2}k(r - r_0)^2$$

where k is the bond coefficient ($\frac{energy}{distance^2}$) and r_0 is the equilibrium bond distance. Note the factor of $\frac{1}{2}$.

Dihedral potentials reported as a fourier series (e.g., OPLS) can be converted to Ryckaert-Bellemans (RB) torsions as specified in the GROMACS User Manual.

8.5.1 Classes vs. Types

OpenMM allows users to specify either a `class` or a `type` (See Atom Types and Atom Classes), to define each particle within the force definition. Here, `type` refers to a specific atom type (as defined in the `<AtomTypes>` section), while `class` refers to a more general description that can apply to multiple atomtypes (i.e. multiple atomtypes may share the same class). This aids in limiting the number of force definitions required in a force field XML, as many similar atom types may share force parameters.

8.5.2 Assigning parameters by specificity

Foyer deviates from OpenMM's convention when matching force definitions in a force field XML to instances of these forces in a molecular system. In OpenMM, forces are assigned according to the first matching definition in a force field XML, even if multiple matching definitions exist. In contrast, Foyer assigns force parameters based on definition specificity, where definitions containing more `type` attributes are considered to be more specific.

Example:

```
<RBTorsionForce>
  <Proper class1="CT" class2="CT" class3="CT" class4="CT" c0="2.9288" c1="-1.4644" c2="0.
↪2092" c3="-1.6736" c4="0.0" c5="0.0"/>
  <Proper type1="opls_961" type2="opls_136" type3="opls_136" type4="opls_136" c0="-0.
↪987424" c1="0.08363" c2="-0.08368" c3="-0.401664" c4="1.389088" c5="0.0"/>
</RBTorsionForce>
```

Above, two proper torsions are defined, both describing a torsional force between four tetrahedral carbons. However, the first definition features four `class` attributes and zero `type` attributes, as this describes a general dihedral for all tetrahedral carbons. The second definition features zero `class` attributes and four `type` attributes, and describes a more specific dihedral for the case where one end carbon is of type `'opls_961'` (a fluorinated carbon) and the remaining three carbons are of type `'opls_136'` (alkane carbons). Now consider we want to use a force field containing the above torsion definitions to assign parameters to some molecular system that features partially fluorinated alkanes. When assigning torsion parameters to a quartet of atoms where one end carbon is fluorinated (`'opls_961'`) and the remaining three are hydrogenated (`'opls_136'`), if using the OpenMM procedure for parameter assignment the more general `'CT-CT-CT-CT'` torsion parameters (the first definition above) would be assigned because this would be the first matching definition in the force field XML. However, with Foyer, the second definition will be auto-detected as more specific, due to the greater number of `type` attributes (4 vs. 0) and those parameters will be assigned instead.

It should be noted that if two definitions feature the same specificity level (i.e. the same number of `type` definitions) then automatic detection of precedence is not possible and parameter assignment will follow the OpenMM procedure whereby parameters from the first matching force definition in the XML will be assigned.

8.6 Applying a force field

The main method you will use in **foyer** is the `Forcefield.apply()` method. There are a few important arguments you should understand.

The first several are the `assert_bond_params` `assert_angle_params`, `assert_dihedral_params`, and `assert_improper_params`. These arguments require that the supplied force field has parameters for every bond, angle, dihedral, and improper in the system. In most cases, if you get an error, it means that your force field is missing parameters for one of the bonds/angles/dihedrals/impropers in the system. This could be because the parameters are missing or because the atom-typing (i.e., the SMARTS strings) are incorrect. These arguments are `True` by default, with the exception of `assert_improper_params`. In all cases, it is wise to verify that the final files you generate have the expected number of bonds/angles/dihedrals/impropers for your system.

The other important optional argument is the `combining_rule` option, which is `"lorentz"` (Lorentz-Berthelot) by default. The other valid option is `"geometric"`, if your force field uses geometric combining rules.

8.7 Usage Examples

Foyer supports atomtyping of both all-atom and coarse-grained molecular systems, and also allows for separate force fields to be used to atom-type separate portions of a molecular system.

8.7.1 Creating a box of ethane

Here we use mBuild to construct a box filled with ethane molecules and use Foyer to atom-type the system, applying the OPLS force field, and save to run-able GROMACS files.

```
import mbuild as mb
from mbuild.lib.molecules import Ethane

ethane_box = mb.fill_box(compound=Ethane(), n_compounds=100, box=[2, 2, 2])
ethane_box.save('ethane-box.gro')
ethane_box.save('ethane-box.top', forcefield_name='oplsaa')
```

8.7.2 Creating a box of coarse-grained ethane

Again we will use mBuild to construct a box filled with ethane molecules. However, now we will model ethane using a united-atom description and apply the TraPPE force field during atom-typing. Note how particle names are prefixed with an underscore so that Foyer knows these particles are non-atomistic.

```
import mbuild as mb

ethane_UA = mb.Compound()
ch3_1 = mb.Particle(name='_CH3', pos=[0, 0, 0])
ch3_2 = mb.Particle(name='_CH3', pos=[0.15, 0, 0])
ethane_UA.add([ch3_1, ch3_2])
ethane_UA.add_bond((ch3_1, ch3_2))

ethane_UA_box = mb.fill_box(ethane_UA, 100, box=[2, 2, 2])
ethane_UA_box.save('ethane-UA-box.gro')
ethane_UA_box.save('ethane-UA-box.top', forcefield_name='trappe-ua')
```

8.7.3 Combining force fields

In some instances, the use of multiple force fields may be desired to describe a molecular system. For example, the user may want to use one force field for a surface and another for a fluid in the same system. Foyer supports this functionality by allowing the user to separately atom-type parts of a system. In this example, we take a system featuring bulk united atom ethane above a silica surface and apply the OPLS force field to the surface and the TraPPE force field to the ethane. The two atomtyped Parmed structures are then combined using a simple '+' operator and can be saved to Gromacs files.

```
from foyer import Forcefield
from foyer.examples.utils import example_file_path
import mbuild as mb
```

(continues on next page)

(continued from previous page)

```
from mbuild.examples import Ethane
from mbuild.lib.atoms import H
from mbuild.lib.bulk_materials import AmorphousSilica
from mbuild.lib.recipes import SilicaInterface
from mbuild.lib.recipes import Monolayer

interface = SilicaInterface(bulk_silica=AmorphousSilica())
interface = Monolayer(surface=interface, chains=H(), guest_port_name='up')

box = mb.Box(mins=[0, 0, max(interface.xyz[:,2])],
             maxs=interface.periodicity + [0, 0, 4])

ethane_box = mb.fill_box(compound=Ethane(), n_compounds=200, box=box)

opls = Forcefield(name='oplsaa')
opls_silica = Forcefield(forcefield_files=example_file_path('opls-silica.xml'))
ethane_box = opls.apply(ethane_box)
interface = opls_silica.apply(interface)

system = interface + ethane_box

system.save('ethane-silica.gro')
system.save('ethane-silica.top')
```

8.8 Examples from Foyer paper

Contained below are the toy examples from the *Usage Examples* section of the foyer paper. The source code selections are listed below on this page, there are Jupyter Notebooks where you can try these examples yourself. Note that these examples are meant to showcase the abilities of `foyer` through simple examples. If the user would like to examine more in-depth examples using `foyer` with `mBuild`, please refer to the tutorial repository.

Below is *Listing 6* from the paper, a python script to fill a $2x2x2nm$ box with 100 ethane molecules. The system is then atomtyped using the OPLS-AA forcefield. There are two approaches to the same problem detailed below in this listing, the first approach uses the `forcefield_files` function argument from mBuild to atomptype the system (using foyer under the hood). While the second approach creates a `foyer Forcefield` object, which then calls its `apply` function, operating on the `mBuild Compound` to return the properly atomtyped structure. Note that in all instances when using `foyer`, the chemical system of interest is converted into a `ParmEd Structure`. Even the `mBuild Compounds`, when calling the `save` routine, are converted into a `ParmEd Structure` before `foyer` can atomtype them. The object returned by `foyer` after the atomtypes have been applied are `ParmEd Structures`. This is subject to change in later iterations of `foyer`.

8.8.1 Homogeneous fluid

```
import mbuild as mb
from mbuild.lib.molecules import Ethane
from foyer.examples.utils import example_file_path
from foyer import Forcefield

""" Applying a force field while saving from mBuild """
# Create the chemical topology
ethane_fluid = mb.fill_box(compound=Ethane(), n_compounds=100, box=[2, 2, 2])
# Apply and save the topology
ethane_fluid.save("ethane-box.top", forcefield_files=example_file_path("oplsaa_alkane.xml
↪"))
ethane_fluid.save("ethane-box.gro")

""" Applying a force field directly with foyer """
# Create the chemical topology
ethane_fluid = mb.fill_box(compound=Ethane(), n_compounds=100, box=[2, 2, 2])
# Load the forcefield
opls_alkane = Forcefield(forcefield_files=example_file_path("oplsaa_alkane.xml"))
# Apply the forcefield to atom-type
ethane_fluid = opls_alkane.apply(ethane_fluid)
# Save the atom-typed system
ethane_fluid.save("ethane-box.top", overwrite=True)
ethane_fluid.save("ethane-box.gro", overwrite=True)
```

8.8.2 Fluid on silica substrate

The other example listing from the text showcases the ability to create two separate chemical topologies and applying different forcefield files to each. The two parameterized systems that are generated are then combined into a single `ParmEd Structure` and saved to disk.

```
from foyer import Forcefield
from foyer.examples.utils import example_file_path
import mbuild as mb
from mbuild.examples import Ethane
from mbuild.lib.atoms import H
from mbuild.lib.bulk_materials import AmorphousSilicaBulk
from mbuild.lib.recipes import SilicaInterface
from mbuild.lib.recipes import Monolayer

# Create a silica substrate, capping surface oxygens with hydrogen
silica=SilicaInterface(bulk_silica=AmorphousSilicaBulk())
silica_substrate=Monolayer(surface=silica,chains=H(),guest_port_name="up")
# Determine the box dimensions dictated by the silica substrate
box=mb.Box(mins=[0, 0,max(silica.xyz[:,2])],maxs=silica.periodicity+ [0, 0, 4])
# Fill the box with ethane
ethane_fluid=mb.fill_box(compound=Ethane(),n_compounds=200,box=box)
# Load the forcefields
#opls_silica=Forcefield(forcefield_files=get_example_file("oplsaa_with_silica.xml"))
```

(continues on next page)

(continued from previous page)

```
opls_silica=Forcefield(forcefield_files=example_file_path("output.xml"))
opls_alkane=Forcefield(forcefield_files=example_file_path("oplsaa_alkane.xml"))
# Apply the forcefields
silica_substrate=opls_silica.apply(silica_substrate)
ethane_fluid=opls_alkane.apply(ethane_fluid)
# Merge the two topologies
system=silica_substrate+ethane_fluid
# Save the atom-typed system
system.save("ethane-silica.top")
system.save("ethane-silica.gro")
```

8.9 Units

Foyer forcefield XML files use the following units. These follow from the units used in OpenMM.

Quantity	Units
distance	nm
angle	radians
mass	amu
energy	kJ/mol

Please note that the units of the parameters found in the parameterized `parmed.Structure` will follow the units used by ParmEd

8.10 Validation of force field files

Foyer performs several validation steps to help prevent malformed force field files and SMARTS strings from making it into production code. Our goal is to provide human readable error messages that users who may not be intimately familiar with XML or our SMARTS parsing grammar can easily act upon.

However, if you receive any unclear error messages or warnings we strongly encourage you to submit an issue detailing the error message you received and, if possible, attach a minimal example of the force field file that created the problem.

8.10.1 XML schema

As a first line of defense, any force field files loaded by foyer is validated by this XML schema definition. Here we enforce which elements (e.g. `HarmonicBondForce`) are valid and how their attributes should be formatted. Additionally, the schema ensures that atomtypes are not 1) defined more than once and that 2) atomtypes referenced in other sections are actually defined in the `<AtomTypes>` element.

8.10.2 SMARTS validation

All SMARTS strings used to define atomtypes are parsed. Parsing errors are captured and re-raised with error messages that allow you to pin point the location of the problem in the XML file and within the SMARTS string. Wherever possible, we attempt to provide helpful hints and we welcome any contributions that help improve the clarity of our error messages.

Additionally, we ensure that any atomtypes referenced using the `%type` or `overrides` syntax are actually defined in the `<AtomTypes>` element.

8.11 Forcefield Class

The primary data structure in foyer is the `Forcefield` class, which inherits from the OpenMM class of the same name. The primary operation on this class is the `.apply()` function, which takes a chemical topology and returns a parametrized `ParmEd Structure`. The user may pass some options to this function based on a particular use case.

`class foyer.forcefield.`**`Forcefield`**(*forcefield_files=None*, *name=None*, *validation=True*, *debug=False*)

Specialization of OpenMM's Forcefield allowing SMARTS based atomtyping.

Parameters

- **forcefield_files** (*list of str, optional, default=None*) – List of forcefield files to load.
- **name** (*str, optional, default=None*) – Name of a forcefield to load that is packaged within foyer.

`apply`(*structure*, *references_file=None*, *use_residue_map=False*, *assert_bond_params=True*, *assert_angle_params=True*, *assert_dihedral_params=True*, *assert_improper_params=False*, *verbose=False*, **args*, ***kwargs*)

Apply the force field to a molecular structure.

Parameters

- **structure** (*parmed.Structure or mbuild.Compound*) – Molecular structure to apply the force field to
- **references_file** (*str, optional, default=None*) – Name of file where force field references will be written (in Bibtex format)
- **use_residue_map** (*boolean, optional, default=False*) – Store atomtyped topologies of residues to a dictionary that maps them to residue names. Each topology, including atomtypes, will be copied to other residues with the same name. This avoids repeatedly calling the subgraph isomorphism on identical residues and should result in better performance for systems with many identical residues, i.e. a box of water. Note that for this to be applied to independent molecules, they must each be saved as different residues in the topology.
- **assert_bond_params** (*bool, optional, default=True*) – If True, Foyer will exit if parameters are not found for all system bonds.
- **assert_angle_params** (*bool, optional, default=True*) – If True, Foyer will exit if parameters are not found for all system angles.
- **assert_dihedral_params** (*bool, optional, default=True*) – If True, Foyer will exit if parameters are not found for all system proper dihedrals.
- **assert_improper_params** (*bool, optional, default=False*) – If True, Foyer will exit if parameters are not found for all system improper dihedrals.

- **verbose** (*bool, optional, default=False*) – If True, Foyer will print debug-level information about notable or potentially problematic details it encounters.

`property combining_rule`

Return the combining rule of this force field.

`property coulomb14scale`

Get Coulombic 1-4 scale for this forcefield.

`createSystem`(*topology*, *nonbondedMethod=NoCutoff*, *nonbondedCutoff=Quantity(value=1.0, unit=nanometer)*, *constraints=None*, *rigidWater=True*, *removeCMMotion=True*, *hydrogenMass=None*, *switchDistance=None*, ***args*)

Construct an OpenMM System representing a Topology with this force field.

Parameters

- **topology** (*Topology*) – The Topology for which to create a System
- **nonbondedMethod** (*object=NoCutoff*) – The method to use for nonbonded interactions. Allowed values are NoCutoff, CutoffNonPeriodic, CutoffPeriodic, Ewald, or PME.
- **nonbondedCutoff** (*distance=1*nanometer*) – The cutoff distance to use for nonbonded interactions
- **constraints** (*object=None*) – Specifies which bonds and angles should be implemented with constraints. Allowed values are None, HBonds, AllBonds, or HAngles.
- **rigidWater** (*boolean=True*) – If true, water molecules will be fully rigid regardless of the value passed for the constraints argument
- **removeCMMotion** (*boolean=True*) – If true, a CMMotionRemover will be added to the System
- **hydrogenMass** (*mass=None*) – The mass to use for hydrogen atoms bound to heavy atoms. Any mass added to a hydrogen is subtracted from the heavy atom to keep their total mass the same.
- **switchDistance** (*float=None*) – The distance at which the potential energy switching function is turned on for
- **args** – Arbitrary additional keyword arguments may also be specified. This allows extra parameters to be specified that are specific to particular force fields.

Returns

the newly created System

Return type

system

`static get_generator`(*ff*, *gen_type*)

Return a specific force generator for this Forcefield.

Parameters

- **ff** (*foyer.Forcefield*) – The forcefield to return generator for
- **gen_type** (*Type*) – The generator type to return

Returns

The instance of *gen_type* for this Forcefield

Return type

instance of gen_type

Raises
`MissingForceError` – If the Forcefield doesn't have a generator of type *gen_type*

`get_parameters`(*group*, *key*, *keys_are_atom_classes=False*)

Get parameters for a specific group of Forces in this Forcefield.

Parameters

- **group** (*str*) – One of {"atoms", "harmonic_bonds", "harmonic_angles", "periodic_propers", "periodic_impropers", "rb_propers", "rb_impropers"}. Note that these entries are case insensitive
- **key** (*str, list of str*) – The atom type(s)/class(es) to extract parameters for
- **keys_are_atom_classes** (*bool, default=False*) – If True, the entries in key are considered to be atom classes rather than atom types

Examples

Following shows some example usage of this function:

```
>>> from foyer import Forcefield
>>> ff = Forcefield(name='oplsaa')
>>> ff.get_parameters('atoms', key='opls_235')  # NonBonded Parameters
{'charge': -0.18, 'sigma': 0.35, 'epsilon': 0.276144}
>>> ff.get_parameters('rb_propers', ['opls_137', 'opls_209', 'opls_193',
↪'opls_154'])  # RBPropers
{'c0': 2.87441, 'c1': 0.58158, 'c2': 2.092, 'c3': -5.54799, 'c4': 0.0, 'c5':
↪ 0.0}
```

Returns
A dictionary (or an item in the list of dictionaries) with parameter names as key and the parameter value as value as it pertains to specific force. Note that the keys of the dictionary can be different for different forces.

Return type
dict or list of dict

Raises

- `MissingParametersError` – Raised when parameters are missing from the forcefield or no matching parameters found
- `MissingForceError` – Raised when a particular force generator is missing from the Forcefield

`property included_forcefields`

Return a dictionary mappying for all included forcefields.

`property lj14scale`

Get LJ 1-4 scale for this forcefield.

`map_atom_classes_to_types`(*atom_classes_keys*, *strict=False*)

Replace the atom_classes in the provided list by atom_types.

`property name`

Return the name of the force field XML.

`parametrize_system`(*structure=None*, *references_file=None*, *assert_bond_params=True*, *assert_angle_params=True*, *assert_dihedral_params=True*, *assert_improper_params=False*, *verbose=False*, **args*, ***kwargs*)

Create system based on resulting typemapping.

`registerAtomType`(*parameters*)

Register a new atom type.

`run_atomtyping`(*structure*, *use_residue_map=True*, ***kwargs*)

Atomtype the topology.

Parameters

- **structure** (*parmed.structure.Structure*) – Molecular structure to find atom types of
- **use_residue_map** (*boolean, optional, default=True*) – Store atomtyped topologies of residues to a dictionary that maps them to residue names. Each topology, including atomtypes, will be copied to other residues with the same name. This avoids repeatedly calling the subgraph isomorphism on identical residues and should result in better performance for systems with many identical residues, i.e. a box of water. Note that for this to be applied to independent molecules, they must each be saved as different residues in the topology.

`static substitute_wildcards`(*atom_types*, *wildcard*)

Return possible wildcard options.

`property version`

Return version number of the force field XML file.

8.12 Citing foyer

If you use foyer for your research, please cite our paper or its pre-print:

ACS

Klein, C.; Summers, A. Z.; Thompson, M. W.; Gilmer, J. B.; Mccabe, C.; Cummings, P. T.; Sallai, J.; Iacovella, C. R. Formalizing Atom-Typing and the Dissemination of Force Fields with Foyer. Computational Materials Science 2019, 167, 215–227.

BibTeX

```
@article{klein2019,
   title = "Formalizing atom-typing and the dissemination of force fields with foyer",
   journal = "Computational Materials Science",
   volume = "167",
   pages = "215 - 227",
   year = "2019",
   issn = "0927-0256",
   doi = "https://doi.org/10.1016/j.commatsci.2019.05.026",
   url = "http://www.sciencedirect.com/science/article/pii/S0927025619303040",
   author = "Christoph Klein and Andrew Z. Summers and Matthew W. Thompson and Justin B.␣
↪Gilmer and Clare McCabe and Peter T. Cummings and Janos Sallai and Christopher R.␣
↪Iacovella",
   keywords = "Molecular simulation, Force fields, Reproducibility, Open-source software
↪",
}
```

Download as `BibTeX` or `RIS`

8.13 License

Various sub-portions of this library may be independently distributed under different licenses. See those files for their specific terms.

Foyer is licensed under the MIT license.

```
The MIT License (MIT)

Copyright (c) 2015 Vanderbilt University

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

INDEX

A

`apply()` (*foyer.forcefield.Forcefield method*), 29

C

`combining_rule` (*foyer.forcefield.Forcefield property*), 30

`coulomb14scale` (*foyer.forcefield.Forcefield property*), 30

`createSystem()` (*foyer.forcefield.Forcefield method*), 30

F

`Forcefield` (*class in foyer.forcefield*), 29

G

`get_generator()` (*foyer.forcefield.Forcefield static method*), 30

`get_parameters()` (*foyer.forcefield.Forcefield method*), 31

I

`included_forcefields` (*foyer.forcefield.Forcefield property*), 31

L

`lj14scale` (*foyer.forcefield.Forcefield property*), 31

M

`map_atom_classes_to_types()` (*foyer.forcefield.Forcefield method*), 31

N

`name` (*foyer.forcefield.Forcefield property*), 31

P

`parametrize_system()` (*foyer.forcefield.Forcefield method*), 31

R

`registerAtomType()` (*foyer.forcefield.Forcefield method*), 32

`run_atomtyping()` (*foyer.forcefield.Forcefield method*), 32

S

`substitute_wildcards()` (*foyer.forcefield.Forcefield static method*), 32

V

`version` (*foyer.forcefield.Forcefield property*), 32