gmso

Release 0.12.1

Matt Thompson, Alex Yang, Ray Matsumoto, Parashara Shamapra

Mar 21, 2024

CONTENTS

GMSO is a flexible storage of chemical topology for molecular simulation. With a few lines of *GMSO* code, together with mBuild and foyer, users can rapidly prototype arbitrary parameterized chemical systems and generate data files for a wide variety of simulation engines.

CHAPTER

ONE

GMSO IS A PART OF THE MOSDEF ECOSYSTEM

GMSO is designed to be a general and flexible representation of chemical topolgies for molecular simulation. With an emphasis on assuming as little as possible about the chemical system, model, or engine, *GMSO* can enable support for a variety of systems. *GMSO* is a part of the MoSDeF (Molecular Simulation and Design Framework) ecosystem, and is intended to be a generalized alternative for the foyer package. Libraries in the MoSDeF ecosystem are designed to provide utilities neccessary to streamline a researcher's simulation workflow. When setting up simulation studies, we also recommend users to follow the TRUE (Transparent, Reproducible, Usable-by-others, and Extensible) standard, which is a set of common practices meant to improve the reproducibility of computational simulation research.

CHAPTER

TWO

GOALS AND FEATURES

GMSO's goal is to provide a flexible backend framework to store topological information of a chemical system in a reproducible fashion. **Topology** in this case is defined as the information needed to initialize a molecular simulation. Depending on the type of simulation performed, this ranges from:

- particle positions
- particle connectivity
- box information
- forcefield data
  - functional forms defined as sympy expressions
  - parameters with defined units
  - partial charges
  - tabulated data
  - etc.
- Other optional data
  - particle mass
  - elemental data
  - etc.

With these driving goals for *GMSO*, the following features are enabled:

1. **Supporting a variety of models**: in the molecular simulation/computational chemistry community: No assumptions are made about an interaction site representing an atom or bead, instead these can be atomistic, united-atom/coarse-grained, polarizable, and other models!
2. **Greater flexibility for exotic potentials**: The *AtomType* (and analogue classes for intramolecular interactions) uses sympy to store any potential that can be represented by a mathematical expression.
3. **Adaptable for new engines**: by not being designed for compatibility with any particular molecular simulation engine or ecosystem, it becomes more tractable for developers in the community to add glue for engines that are not currently supported.
4. **Compatibility with existing community tools**: No single molecular simulation tool will ever be a silver bullet, so `GMSO` includes functions to convert between various file formats and libraries. These can be used in their own right to convert between objects in-memory and also to support conversion to file formats not natively supported at any given time. Currently supported conversions include:
   - *ParmEd*
   - *OpenMM*

- *mBuild*
- more in the future!

5. **Native support for reading and writing many common file formats**:
   - *XYZ*
   - *GRO*
   - *TOP*
   - *LAMMPSDATA*
   - *GSD*
   - indirect support, through other libraries, for many more!

2.1 Design Principles

2.1.1 Scope and Features of GMSO

GMSO is designed to enable the flexible, general representation of chemical topologies for molecular simulation. Efforts are made to enable lossless, bias-free storage of data, without assuming particular chemistries, models, or using any particular engine's ecosystem as a starting point. The scope is generally restrained to the preparation, manipulation, and conversion of and of input files for molecular simulation, i.e. before engines are called to execute the simulations themselves. GMSO currently does not support conversions between trajectory file formats for analysis codes. In the scope of molecular simulation, we loosely define a chemical topology as everything needed to reproducibly prepare a chemical system for simulation. This includes particle coordinates and connectivity, box information, force field data (functional forms, parameters tagged with units, partial charges, etc.) and some optional information that may not apply to all systems (i.e. specification of elements with each particle).

GMSO enables the following features:

- Supporting a variety of models in the molecular simulation/computational chemistry community: No assumptions are made about an interaction site representing an atom or bead, instead supported atomistic, united-atom/coarse-grained, polarizable, and other models!
- Greater flexibility for exotic potentials: The `AtomType` (and analogue classes for intramolecular interactions) uses `sympy` to store any potential that can be represented by a mathematical expression. If you can write it down, it can be stored!
- Easier development for glue to new engines: by not being designed for compatibility with any particular molecular simulation engine or ecosystem, it becomes more tractable for developers in the community to add glue for engines that are not currently supported (and even ones that do not exist at present)!
- Compatibility with existing community tools: No single molecular simulation tool will be a silver bullet, so GMSO includes functions to convert objects. These can be used in their own right to convert between objects in-memory and also to support conversion to file formats not natively supported at any given time. Currently supported conversions include `ParmEd`, `OpenMM`, `mBuild`, `MDTraj`, with others coming in the future!
- Native support for reading and writing many common file formats (`XYZ`, `GRO`, `TOP`, `LAMMPSDATA`) and indirect support, through other libraries, for many more!

2.1.2 Structure of GMSO

There are three main modules within the Python package:

- `gmso.core` stores the classes that constitute the core data structures.
- `gmso.formats` stores readers and writers for (on-disk) file formats.
- `gmso.external` includes functions that convert core data structures between external libraries and their internal representation.

2.2 Data Structures in GMSO

Following data structures are available within GMSO.

2.2.1 Core Classes

gmso.Topology	A topology.
gmso.Atom	An atom represents a single element association in a topology.
gmso.Bond	A 2-partner connection between sites.
gmso.Angle	A 3-partner connection between Atoms.
gmso.Dihedral	A 4-partner connection between sites.
gmso.Improper	A 4-partner connection between sites.

Topology

class gmso.**Topology**(*name='Topology'*, *box=None*)

A topology.

A topology represents a chemical structure wherein lie the collection of sites which together form a chemical structure containing connections (gmso.Bond, gmso.Angle and gmso.Dihedral (along with their associated types). A topology is the fundamental data structure in GMSO, from which we can gather various information about the chemical structure and apply a forcefield before converting the structure into a format familiar to various simulation engines.

Parameters

- **name** (*str, optional, default='Topology'*) – A name for the Topology.
- **box** (*gmso.Box, optional, default=None*) – A gmso.Box object bounding the topology

Variables

- **typed** (*bool*) – True if the topology is typed
- **combining_rule** (*str, ['lorentz', 'geometric']*) – The combining rule for the topology, can be either 'lorentz' or 'geometric'
- **scaling_factors** (*dict*) – A collection of scaling factors used in the forcefield
- **n_sites** (*int*) – Number of sites in the topology

- **n_connections** (*int*) – Number of connections in the topology (Bonds, Angles, Dihedrals, Impropers)
- **n_bonds** (*int*) – Number of bonds in the topology
- **n_angles** (*int*) – Number of angles in the topology
- **n_dihedrals** (*int*) – Number of dihedrals in the topology
- **n_impropers** (*int*) – Number of impropers in the topology
- **connections** (*tuple of gmso.Connection objects*) – A collection of bonds, angles, dihedrals, and impropers in the topology
- **bonds** (*tuple of gmso.Bond objects*) – A collection of bonds in the topology
- **angles** (*tuple of gmso.Angle objects*) – A collection of angles in the topology
- **dihedrals** (*tuple of gmso.Dihedral objects*) – A collection of dihedrals in the topology
- **impropers** (*tuple of gmso.Improper objects*) – A collection of impropers in the topology
- **connection_types** (*tuple of gmso.Potential objects*) – A collection of BondTypes, AngleTypes, DihedralTypes, and ImproperTypes in the topology
- **atom_types** (*tuple of gmso.AtomType objects*) – A collection of AtomTypes in the topology
- **bond_types** (*tuple of gmso.BondType objects*) – A collection of BondTypes in the topology
- **angle_types** (*tuple of gmso.AngleType objects*) – A collection go AngleTypes in the topology
- **dihedral_types** (*tuple of gmso.DihedralType objects*) – A collection of DihedralTypes in the topology
- **improper_types** (*tuple of gmso.ImproperType objects*) – A collection of ImproperTypes in the topology
- **pairpotential_types** (*tuple of gmso.PairPotentialType objects*) – A collection of PairPotentialTypes in the topology
- **atom_type_expressions** (*list of gmso.AtomType.expression objects*) – A collection of all the expressions for the AtomTypes in topology
- **connection_type_expressions** (*list of gmso.Potential.expression objects*) – A collection of all the expressions for the Potential objects in the topology that represent a connection type
- **bond_type_expressions** (*list of gmso.BondType.expression objects*) – A collection of all the expressions for the BondTypes in topology
- **angle_type_expressions** (*list of gmso.AngleType.expression objects*) – A collection of all the expressions for the AngleTypes in topology
- **dihedral_type_expressions** (*list of gmso.DihedralType.expression objects*) – A collection of all the expression for the DihedralTypes in the topology
- **improper_type_expressions** (*list of gmso.ImproperType.expression objects*) – A collection of all the expression for the ImproperTypes in the topology

- **pairpotential_type_expressions** (*list of gmso.PairPotentialType.expression objects*) – A collection of all the expression for the PairPotentialTypes in the topology

add_connection(*connection*, *update_types=False*)

Add a gmso.Connection object to the topology.

This method will add a gmso.Connection object to the topology, it can be used to generically include any Connection object i.e. Bond or Angle or Dihedral to the topology. According to the type of object added, the equivalent collection in the topology is updated. For example- If you add a Bond, this method will update topology.connections and topology.bonds object. Additionally, if update_types is True (default behavior), it will also update any Potential objects associated with the connection.

Parameters

- **connection** (*one of gmso.Connection, gmso.Bond, gmso.Angle, gmso.Dihedral, or gmso.Improper object*)
- **update_types** (*bool, default=True*) – If True also add any Potential object associated with connection to the topology.

Returns

The Connection object or equivalent Connection object that is in the topology

Return type

gmso.Connection

add_site(*site*, *update_types=False*)

Add a site to the topology.

This method will add a site to the existing topology, since sites are stored in an indexed set, adding redundant site will have no effect. If the update_types parameter is set to true (default behavior), this method will also check if there is an gmso.AtomType associated with the site and it to the topology's AtomTypes collection.

Parameters

- **site** (*gmso.core.Site*) – Site to be added to this topology
- **update_types** (*(bool), default=True*) – If true, add this site's atom type to the topology's set of AtomTypes

update_topology()

Update the entire topology.

Atom

class gmso.**Atom**(*, *name: str = ''*, *label: str = ''*, *group: str | None = None*, *molecule: Molecule | None = None*, *residue: Residue | None = None*, *position: Sequence[float] | ndarray | unyt_array = None*, *charge: unyt_quantity | float | None = None*, *mass: unyt_quantity | float | None = None*, *element: Element | None = None*, *atom_type: AtomType | None = None*, *restraint: dict | None = None*)

An atom represents a single element association in a topology.

Atoms are the representation of an element within *gmso* that describes any general atom in a molecular simulation. Atoms also contain information that are unique to elements vs other types of interaction sites in molecular simulations. For example, charge, mass, and periodic table information.

Notes

Atoms have all the attributes inherited from the base Site class, The order of precedence when attaining properties *charge* and *mass* is:

1. atom.charge > atom.atom_type.charge
2. atom.mass > atom.atom_type.mass

Examples

```
>>> from gmso.core.atom import Atom
>>> atom1 = Atom(name='lithium')
```

See also:

gmso.abc.AbstractSite
An Abstract Base class for implementing site objects in GMSO. The class Atom bases from the gmso.abc.abstract site class

property atom_type: *AtomType* | property
Return the atom_type associated with the atom.

property charge: unyt_quantity | None
Return the charge of the atom.

clone()
Clone this atom.

property element: Element | None
Return the element associated with the atom.

classmethod is_valid_charge(*charge*)
Ensure that the charge is physically meaningful.

classmethod is_valid_mass(*mass*)
Ensure that the mass is physically meaningful.

property mass: unyt_quantity | None
Return the mass of the atom.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}
A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

property restraint
Return the restraint of this atom.

Bond

class gmso.Bond(*, *name: str = ''*, *connection_members: Tuple[Atom, Atom]*, *bond_type: BondType | None = None*, *restraint: dict | None = None*)

A 2-partner connection between sites.

This is a subclass of the gmso.abc.Connection superclass. This class has strictly 2 members in its connection_members. The connection_type in this class corresponds to gmso.BondType.

Notes

Inherits some methods from Connection: __eq__, __repr__, _validate methods.

Additional _validate methods are presented.

---

property bond_type

Return parameters of the potential type.

property connection_type

Return parameters of the potential type.

equivalent_members()

Get a set of the equivalent connection member tuples.

Returns
A unique set of tuples of equivalent connection members

Return type
frozenset

Notes

For a bond:
i, j == j, i
where i and j are the connection members.

---

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

property restraint

Return the restraint of this bond.

Angle

class gmso.Angle(*, *name: str = ''*, *connection_members: Tuple[*Atom, Atom, Atom*]*, *angle_type:* AngleType | *None* = *None*, *restraint: dict* | *None* = *None*)

A 3-partner connection between Atoms.

This is a subclass of the gmso.Connection superclass. This class has strictly 3 members in its connection members. The connection_type in this class corresponds to gmso.AngleType.

Notes

Inherits some methods from Connection: __eq__, __repr__, _validate methods

Additional _validate methods are presented.

---

property angle_type

Return the angle type if the angle is parametrized.

property connection_type

Return the angle type if the angle is parametrized.

equivalent_members()

Return a set of the equivalent connection member tuples.

Returns
A unique set of tuples of equivalent connection members

Return type
frozenset

Notes

For an angle:
i, j, k == k, j, i
where i, j and k are the connection members.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

property restraint

Return the restraint of this angle.

Dihedral

class gmso.**Dihedral**(*, *name: str = ''*, *connection_members: Tuple[*Atom, Atom, Atom, Atom*]*, *dihedral_type:* DihedralType | *None = None*, *restraint: dict | None = None*)

A 4-partner connection between sites.

This is a subclass of the gmso.Connection superclass. This class has strictly 4 members in its connection_members. The connection_type in this class corresponds to gmso.DihedralType. The connectivity of a dihedral is: m1–m2–m3–m4

where m1, m2, m3, and m4 are connection members 1-4, respectively.

Notes

Inherits some methods from Connection: __eq__, __repr__, _validate methods

Additional _validate methods are presented.

equivalent_members()

Get a set of the equivalent connection member tuples

Returns
A unique set of tuples of equivalent connection members

Return type
frozenset

Notes

For a dihedral:
i, j, k, l == l, k, j, i
where i, j, k, and l are the connection members.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

property restraint

Return the restraint of this dihedral.

Improper

class gmso.**Improper**(*, *name: str = ''*, *connection_members: Tuple[Atom, Atom, Atom, Atom]*, *improper_type:* ImproperType | *None = None*)

A 4-partner connection between sites.

This is a subclass of the gmso.Connection superclass. This class has strictly 4 members in its connection_members. The connection_type in this class corresponds to gmso.ImproperType The connectivity of an improper is:

m2

m3 - m1 - m4

where m1, m2, m3, and m4 are connection members 1-4, respectively.

Notes

Inherits some methods from Connection: __eq__, __repr__, _validate methods

Additional _validate methods are presented.

---

property connection_type

Return Potential object for this connection if it exists.

equivalent_members()

Get a set of the equivalent connection member tuples.

Returns
A unique set of tuples of equivalent connection members

Return type
frozenset

Notes

For an improper:
i, j, k, l == i, k, j, l
where i, j, k, and l are the connection members.

---

property improper_type

Return Potential object for this connection if it exists.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

2.2.2 Potential Classes

gmso.AtomType	A description of non-bonded interactions between sites.
gmso.BondType	A descripton of the interaction between 2 bonded partners.
gmso.AngleType	A descripton of the interaction between 3 bonded partners.
gmso.DihedralType	A descripton of the interaction between 4 bonded partners.
gmso.ImproperType	A description of the interaction between 4 bonded partners.
gmso.PairPotentialType	A description of custom pairwise potential between 2 AtomTypes that does not follow combination rule.

AtomType

class gmso.**AtomType**(*name='AtomType'*, *mass=unyt_quantity(0., 'g/mol')*, *charge=unyt_quantity(0., 'C')*, *expression=None*, *parameters=None*, *potential_expression=None*, *independent_variables=None*, *atomclass=''*, *doi=''*, *overrides=None*, *definition=''*, *description=''*, *tags=None*)

A description of non-bonded interactions between sites.

This is a subclass of the gmso.core.Potential superclass.

AtomType represents an atom type and includes the functional form describing its interactions and, optionally, other properties such as mass and charge. This class inhereits from Potential, which stores the non-bonded interaction between atoms or sites. The functional form of the potential is stored as a *sympy* expression and the parameters, with units, are stored explicitly.

property atomclass

Return the atomclass of the atom_type.

property charge

Return the charge of the atom_type.

clone(*fast_copy=False*)

Clone this AtomType, faster alternative to deepcopying.

property definition

Return the SMARTS string of the atom_type.

property description

Return the description of the atom_type.

property doi

Return the doi of the atom_type.

property mass

Return the mass of the atom_type.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

property overrides

Return the overrides of the atom_type.

classmethod validate_charge(*charge*)

Check to see that a charge is a unyt array of the right dimension.

classmethod validate_mass(*mass*)

Check to see that a mass is a unyt array of the right dimension.

BondType

class gmso.**BondType**(*name='BondType'*, *expression=None*, *parameters=None*, *independent_variables=None*, *potential_expression=None*, *member_types=None*, *member_classes=None*, *tags=None*)

A descripton of the interaction between 2 bonded partners.

This is a subclass of the gmso.core.Potential superclass.

BondType represents a bond type and includes the functional form describing its interactions. The functional form of the potential is stored as a *sympy* expression and the parameters, with units, are stored explicitly. The AtomTypes that are used to define the bond type are stored as *member_types*.

Notes

Inherits many functions from gmso.ParametricPotential:
__eq__, _validate functions

---

property member_types

Return the members involved in this bondtype.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

AngleType

class gmso.**AngleType**(*name='AngleType'*, *expression=None*, *parameters=None*, *independent_variables=None*, *potential_expression=None*, *member_types=None*, *member_classes=None*, *tags=None*)

A descripton of the interaction between 3 bonded partners.

This is a subclass of the gmso.core.Potential superclass.

AngleType represents an angle type and includes the functional form describing its interactions. The functional form of the potential is stored as a *sympy* expression and the parameters, with units, are stored explicitly. The AtomTypes that are used to define the angle type are stored as *member_types*.

Notes

Inherits many functions from gmso.ParametricPotential:
__eq__, _validate functions

---

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

DihedralType

class gmso.**DihedralType**(*name='DihedralType'*, *expression=None*, *parameters=None*, *independent_variables=None*, *potential_expression=None*, *member_types=None*, *member_classes=None*, *tags=None*)

A descripton of the interaction between 4 bonded partners.

This is a subclass of the gmso.core.Potential superclass.

DihedralType represents a dihedral type and includes the functional form describing its interactions. The functional form of the potential is stored as a *sympy* expression and the parameters, with units, are stored explicitly. The AtomTypes that are used to define the dihedral type are stored as *member_types*.

The connectivity of a dihedral is:

m1–m2–m3–m4

where m1, m2, m3, and m4 are connection members 1-4, respectively.

Notes

Inherits many functions from gmso.ParametricPotential:
__eq__, _validate functions

---

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

ImproperType

class gmso.**ImproperType**(*name='ImproperType'*, *expression=None*, *parameters=None*, *independent_variables=None*, *potential_expression=None*, *member_types=None*, *member_classes=None*, *tags=None*)

A description of the interaction between 4 bonded partners.

This is a subclass of the gmso.core.Potential superclass.

ImproperType represents a improper type and includes the functional form describing its interactions. The functional form of the potential is stored as a *sympy* expression and the parameters, with units, are stored explicitly. The AtomTypes that are used to define the improper type are stored as *member_types*. The connectivity of an improper is:

m2

m3 - m1 - m4

where m1, m2, m3, and m4 are connection members 1-4, respectively.

Notes

Inherits many functions from gmso.ParametricPotential: __eq__, _validate functions

---

property member_types

Return member information for this ImproperType.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

PairPotentialType

class gmso.**PairPotentialType**(*name='PairPotentialType'*, *expression=None*, *parameters=None*, *independent_variables=None*, *potential_expression=None*, *member_types=None*, *tags=None*)

A description of custom pairwise potential between 2 AtomTypes that does not follow combination rule.

This is a subclass of the gmso.core.ParametricPotential superclass.

PairPotentialType represents a type of pairwise potential between two Atomtypes that does not follow a specific combination rule, and includes the functional form describing its interactions. The functional form of the potential is stored as a *sympy* expression and the parameters, with units, are stored explicitly. The AtomTypes that are used to define the dihedral type are stored as *member_types*.

Notes

Inherits many functions from gmso.ParametricPotential:
__eq__, _validate functions

---

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

2.2.3 ForceField

class gmso.**ForceField**(*xml_loc=None*, *strict=True*, *greedy=True*, *backend='forcefield-utilities'*)

A generic implementation of the forcefield class.

The ForceField class is one of the core data structures in gmso, which is used to hold a collection of gmso.core.Potential subclass objects along with some metadata to represent a forcefield. The forcefield object can be applied to any gmso.Topology which has effects on its Sites, Bonds, Angles and Dihedrals.

Parameters

- **xml_loc** (*str*) – Path to the forcefield xml. The forcefield xml can be either in Foyer or GMSO style.
- **strict** (*bool, default=True*) – If true, perform a strict validation of the forcefield XML file
- **greedy** (*bool, default=True*) – If True, when using strict mode, fail on the first error/mismatch
- **backend** (*str, default="forcefield-utilities"*) – Can be "gmso" or "forcefield-utilities". This will define the methods to load the forcefield.

Variables

- **name** (*str*) – Name of the forcefield
- **version** (*str*) – Version of the forcefield
- **atom_types** (*dict*) – A collection of atom types in the forcefield
- **bond_types** (*dict*) – A collection of bond types in the forcefield
- **angle_types** (*dict*) – A collection of angle types in the forcefield

- **dihedral_types** (*dict*) – A collection of dihedral types in the forcefield
- **units** (*dict*) – A collection of unyt.Unit objects used in the forcefield
- **scaling_factors** (*dict*) – A collection of scaling factors used in the forcefield

See also:

gmso.ForceField.from_xml
A class method to create forcefield object from XML files

gmso.utils.ff_utils.validate
Function to validate the gmso XML file

property atom_class_groups

Return a dictionary of atomClasses in the Forcefield.

classmethod from_xml(*xmls_or_etrees*, *strict=True*, *greedy=True*)

Create a gmso.Forcefield object from XML File(s).

This class method creates a ForceField object from the reference XML file. This method takes in a single or collection of XML files with information about gmso.AtomTypes, gmso.BondTypes, gmso.AngleTypes , gmso.PairPotentialTypes and gmso.DihedralTypes to create the ForceField object.

Parameters

- **xmls_or_etrees** (*Union[str, Iterable[str], etree._ElementTree, Iterable[etree._ElementTree]]*) – The forcefield XML locations or XML Element Trees
- **strict** (*bool, default=True*) – If true, perform a strict validation of the forcefield XML file
- **greedy** (*bool, default=True*) – If True, when using strict mode, fail on the first error/mismatch

Returns

forcefield – A gmso.Forcefield object with a collection of Potential objects created using the information in the XML file

Return type

gmso.ForceField

get_parameters(*group*, *key*, *warn=False*, *copy=False*)

Return parameters for a specific potential by key in this ForceField.

This function uses the *get_potential* function to get Parameters

See also:

gmso.ForceField.get_potential
Get specific potential/parameters from a forcefield potential group by key

get_potential(*group*, *key*, *return_match_order=False*, *warn=False*)

Return a specific potential by key in this ForceField.

Parameters

- **group** (*{'atom_type', 'bond_type', 'angle_type', 'dihedral_type', 'improper_type'}*) – The potential group to perform this search on
- **key** (*str (for atom type) or list of str (for connection types)*) – The key to lookup for this potential group
- **return_match_order** (*bool, default=False*) – If true, return the order of connection member types/classes that got matched
- **warn** (*bool, default=False*) – If true, raise a warning instead of Error if no match found

Returns
The parametric potential requested
Return type
gmso.ParametricPotential
Raises
MissingPotentialError – When the potential specified by *key* is not found in the ForceField potential group *group*

`group_angle_types_by_expression()`

Return all AngleTypes in this ForceField with grouped by expression.

See also:

`_group_by_expression`
Groups a dictionary of gmso.ParametricPotentials by their expression

Returns
A dictionary where the key, value -> expression, list of AngleTypes with that expression
Return type
dict

`group_atom_types_by_expression()`

Return all AtomTypes in this ForceField with grouped by expression.

See also:

`_group_by_expression`
Groups a dictionary of gmso.ParametricPotentials by their expression

Returns
A dictionary where the key, value -> expression, list of atom_types with that expression
Return type
dict

`group_bond_types_by_expression()`

Return all BondTypes in this ForceField with grouped by expression.

See also:

`_group_by_expression`
Groups a dictionary of gmso.ParametricPotentials by their expression

Returns
A dictionary where the key, value -> expression, list of BondTypes with that expression
Return type
dict

`group_dihedral_types_by_expression()`

Return all DihedralTypes in this ForceField with grouped by expression.

See also:

`_group_by_expression`
Groups a dictionary of gmso.ParametricPotentials by their expression

Returns
A dictionary where the key, value -> expression, list of DihedralTypes with that expression

Return type
dict

`group_improper_types_by_expression()`

Return all ImproperTypes in this ForceField with grouped by expression.

See also:

`_group_by_expression`
Groups a dictionary of gmso.ParametricPotentials by their expression

Returns
A dictionary where the key, value -> expression, list of ImproperTypes with that expression
Return type
dict

`group_pairpotential_types_by_expression()`

Return all PairPotentialTypes in this ForceField with grouped by expression

See also:

`_group_by_expression`
Groups a dictionary of gmso.ParametricPotentials by their expression

Returns
A dictionary where the key, value -> expression, list of PairPotentialTypes with that expression
Return type
dict

`property non_element_types`

Get the non-element types in the ForceField.

`to_xml(`*filename*, *overwrite=False*, *backend='gmso'*`)`

Get an lxml ElementTree representation of this ForceField
Parameters
- **filename** (*Union[str, pathlib.Path], default=None*) – The filename to write the XML file to
- **overwrite** (*bool, default=False*) – If True, overwrite an existing file if it exists
- **backend** (*str, default="gmso"*) – Can be "gmso" or "forcefield-utilities". This will define the methods to write the xml.

2.3 Formats

This submodule provides readers and writers for (on-disk) file formats.

2.3.1 GROMACS

The following methods are available for reading and writing GROMACS files.

gmso.formats.**read_gro**(*filename*)

Create a topology from a provided gro file.

The Gromos87 (gro) format is a common plain text structure file used commonly with the GROMACS simulation engine. This file contains the simulation box parameters, number of atoms, the residue and atom number for each atom, as well as their positions and velocities (velocity is optional). This method will receive a filepath representation either as a string, or a file object and return a *topology*.

Parameters
filename (*str or file object*) – The path to the gro file either as a string, or a file object that points to the gro file.

Returns
A *topology* object containing site information

Return type
gmso.core.topology

Notes

Gro files do not specify connections between atoms, the returned topology will not have connections between sites either.

Currently this implementation does not support parsing velocities from a gro file or gro file with more than 1 frame.

All residues and resid information from the gro file are currently lost when converting to *topology*.

---

gmso.formats.**write_gro**(*top*, *filename*, *n_decimals=3*, *shift_coord=False*)

Write a topology to a gro file.

The Gromos87 (gro) format is a common plain text structure file used commonly with the GROMACS simulation engine. This file contains the simulation box parameters, number of atoms, the residue and atom number for each atom, as well as their positions and velocities (velocity is optional). This method takes a topology object and a filepath string or file object and saves a Gromos87 (gro) to disk.

Parameters

- **top** (*gmso.core.topology*) – The *topology* to write out to the gro file.
- **filename** (*str or file object*) – The location and name of file to save to disk.
- **n_decimals** (*int, optional, default=3*) – The number of sig fig to write out the position in.
- **shift_coord** (*bool, optional, default=False*) – If True, shift the coordinates of all sites by the minimum position to ensure all sites have non-negative positions. This is not a requirement for GRO files, but can be useful for visualizing.

Notes

Multiple residue assignment has not been added, each *site* will belong to the same resid of 1 currently.

Velocities are not written out.

2.3.2 GSD

The following methods are available for reading and writing GSD files.

gmso.formats.**write_gsd**(*top*, *filename*, *base_units=None*, *rigid_bodies=None*, *shift_coords=True*, *write_special_pairs=True*)

Output a GSD file (HOOMD v3 default data format).

The *GSD* binary file format is the native format of HOOMD-Blue. This file can be used as a starting point for a HOOMD-Blue simulation, for analysis, and for visualization in various tools.

Parameters

- **top** (*gmso.Topology*) – gmso.Topology object
- **filename** (*str*) – Path of the output file.
- **rigid_bodies** (*list of int, optional, default=None*) – List of rigid body information. An integer value is required for each atom corresponding to the index of the rigid body the particle is to be associated with. A value of None indicates the atom is not part of a rigid body.
- **shift_coords** (*bool, optional, default=True*) – Shift coordinates from (0, L) to (-L/2, L/2) if necessary.
- **write_special_pairs** (*bool, optional, default=True*) – Writes out special pair information necessary to correctly use the OPLS fudged 1,4 interactions in HOOMD.

Notes

Force field parameters are not written to the GSD file and must be included manually in a HOOMD input script. Work on a HOOMD plugin is underway to read force field parameters from a Foyer XML file.

2.3.3 xyz

The following methods are available for reading and writing xyz files.

gmso.formats.**read_xyz**(*filename*)

Reader for xyz file format.

Read in an xyz file at the given path and return a Topology object.

Parameters
filename (*str*) – Path to .xyz file that need to be read.

Returns
top – Topology object

Return type
topology.Topology

gmso.formats.**write_xyz**(*top*, *filename*)

Writer for xyz file format.

Write a Topology object to an xyz file at the given path.

Parameters

- **top** (*topology.Topology*) – Topology object that needs to be written out.

- **filename** (*str*) – Path to file location.

2.3.4 LAMMPS DATA

The following methods are available for reading and writing LAMMPS data.

gmso.formats.**write_lammpsdata**(*top*, *filename*, *atom_style='full'*, *unit_style='real'*, *strict_potentials=False*, *strict_units=False*, *lj_cfactorsDict=None*)

Output a LAMMPS data file.

Outputs a LAMMPS data file in the 'full' atom style format. Assumes use of 'real' units. See http://lammps.sandia.gov/doc/atom_style.html for more information on atom styles.

Parameters

- **Topology** (*Topology*) – A Topology Object
- **filename** (*str*) – Path of the output file
- **atom_style** (*str, optional, default='full'*) – Defines the style of atoms to be saved in a LAMMPS data file. The following atom styles are currently supported: 'full', 'atomic', 'charge', 'molecular' see http://lammps.sandia.gov/doc/atom_style.html for more information on atom styles.
- **unit_style** (*str, optional, default='real'*) – Can be any of "real", "lj", "metal", "si", "cgs", "electron", "micro", "nano". Otherwise an error will be thrown. These are defined in _unit_style_factory. See https://docs.lammps.org/units.html for LAMMPS documentation.
- **strict_potentials** (*bool, optional, default False*) – Tells the writer how to treat conversions. If False, then check for conversions to usable potential styles found in default_parameterMaps. If True, then error if potentials are not compatible.
- **strict_units** (*bool, optional, default False*) – Tells the writer how to treat unit conversions. If False, then check for conversions to unit styles defined in _unit_style_factory. If True, then error if parameter units do not match.
- **lj_cfactorsDict** (*(None, dict), optional, default None*) – If using unit_style="lj" only, can pass a dictionary with keys of ("mass", "energy", "length", "charge"), or any combination of these, and they will be used to non- dimensionalize all values in the topology. If any key is not passed, default values will be pulled from the topology (see _default_lj_val). These are the largest: sigma, epsilon, atomtype.mass, and atomtype.charge from the topology.

Notes

See http://lammps.sandia.gov/doc/2001/data_format.html for a full description of the LAMMPS data format. This is a work in progress, as only a subset of everything LAMMPS supports is currently available. However, please raise issues as the current writer has been set up to eventually grow to support all LAMMPS styles.

Some of this function has been adopted from *mdtraj*'s support of the LAMMPSTRJ trajectory format. See https://github.com/mdtraj/mdtraj/blob/master/mdtraj/formats/lammpstrj.py for details.

2.4 External

This submodule includes functions that convert core data structures between external libraries and their internal representation.

2.4.1 mBuild

The following methods are available for converting mBuild objects to and from GMSO.

gmso.external.**from_mbuild**(*compound*, *box=None*, *search_method=<function element_by_symbol>*, *parse_label=True*, *custom_groups=None*, *infer_elements=False*)

Convert an mbuild.Compound to a gmso.Topology.

This conversion makes the following assumptions about the inputted *Compound*:

- All positional and box dimension values in compound are in nanometers.
- The hierarchical structure of the Compound will be flattened and translated to labels in GMSO Sites. The directly supported labels include *Site.group*, *Site.molecule_name*, and *Site.residue_name*.
  - *group* is determined as te second-highest level Compound and is automatically generated; * *molecule* is determined by traversing through hierarchy of the mb.Compound, starting from the particle level, until the lowest independent mb.Compound is reached (determined as an mb.Compound that does not have any bond outside its boundary);
  - *residue* is the *mb.Compound* level right above particle level.
  - *molecule* and *residue* take the format of (name, index), where the latter can be used to distinguish between molecule/residue of the same name. These two labels are only generated if parse_label=True.
- Only *Bonds* are added for each bond in the *Compound*. If *Angles* and *Dihedrals* are desired in the resulting *Topology*, they must be added separately from this function.

Parameters

- **compound** (*mbuild.Compound*) – mbuild.Compound instance that need to be converted
- **box** (*mbuild.Box, optional, default=None*) – Box information to be loaded to a gmso.Topology
- **search_method** (*function, optional, default=element_by_symbol*) – Searching method used to assign element from periodic table to particle site. The information specified in the *search_method* argument is extracted from each *Particle*'s *name* attribute. Valid functions are element_by_symbol, element_by_name, element_by_atomic_number, and element_by_mass, which can be imported from *gmso.core.element*
- **parse_label** (*bool, optional, default=True*) – Option to parse hierarchy info of the compound into system of top label, including, group, molecule and residue labels.
- **custom_groups** (*list or str, optional, default=None*) – Allows user to identify the groups assigned to each site in the topology based on the compound.name attributes found traversing down the hierarchy. Be sure to supply names such that every particle will be pass through one matching name on the way down from compound.children.

Only the first match while moving downwards will be assigned to the site. If parse_label=False, this argument does nothing.

- **infer_elements** (*bool, default=False*) – Allows the reader to try to load element info from the mbuild Particle.name instead of only from the populated Particle.element

Returns
top

Return type
gmso.Topology

`gmso.external.to_mbuild`(*topology*, *infer_hierarchy=True*)

Convert a gmso.Topology to mbuild.Compound.

Parameters

- **topology** (*gmso.Topology*) – topology instance that need to be converted
- **infer_hierarchy** (*bool, optional, default=True*) – Option to infer the hierarchy from Topology's labels

Returns
compound

Return type
mbuild.Compound

2.4.2 Parmed

Conversion methods for Parmed objects to and from GMSO.

`gmso.external.from_parmed`(*structure*, *refer_type=True*)

Convert a parmed.Structure to a gmso.Topology.

Convert a parametrized or un-parametrized parmed.Structure object to a topology.Topology. Specifically, this method maps Structure to Topology and Atom to Site. This method can only convert AtomType, BondType AngleType, DihedralType, and ImproperType.

Parameters

- **structure** (*parmed.Structure*) – parmed.Structure instance that need to be converted.
- **refer_type** (*bool, optional, default=True*) – Whether or not to transfer AtomType, BondType, AngleType, DihedralType, and ImproperType information

Returns
top

Return type
gmso.Topology

2.4.3 OpenMM

Conversion methods for OpenMM objects to and from GMSO.

gmso.external.**to_openmm**(*topology*, *openmm_object='topology'*)

Convert an untyped topology object to an untyped OpenMM modeller or topology.

This is useful if it's preferred to atom-type a system within OpenMM. See http://openmm.org for more information.

Parameters

- **topology** (*Topology* object) – An untyped topology object
- **open_mm_object** (*'topology' or 'modeller' OpenMM object, default='topology'*) – Untyped OpenMM object to convert to

Returns
open_mm_object

Return type
Untyped *topology* or *modeller* object

2.5 Installation

2.5.1 Installing with conda

Starting from GMSO version `0.3.0`, you can use conda to install GMSO in your preferred environment. This will also install the dependencies of GMSO.

```
(your-env) $ conda install -c conda-forge gmso
```

2.5.2 Installing from source conda

Dependencies of GMSO are listed in the files `environment.yml` (lightweight environment specification containing minimal dependencies) and `environment-dev.yml` (comprehensive environment specification including optional and testing packages for developers). The `gmso` or `gmso-dev` conda environments can be created with

```
$ git clone https://github.com/mosdef-hub/gmso.git
$ cd gmso
# for gmso conda environment
$ conda env create -f environment.yml
$ conda activate gmso

# for gmso-dev
$ conda env create -f environment-dev.yml
$ conda activate gmso

# install a non-editable version of gmso
$ pip install .
```

2.5.3 Install an editable version from source

Once all dependencies have been installed and the `conda` environment has been created, the `GMSO` itself can be installed.

```
$ cd gmso
$ conda activate gmso-dev # or gmso depending on your installation
$ pip install -e .
```

2.5.4 Supported Python Versions

Python 3.9-3.11 is the recommend version for users. It is the only version on which development and testing consistently takes place. Older (3.6-3.9) and newer (3.12+) versions of Python 3 are likely to work but no guarantee is made and, in addition, some dependencies may not be available for other versions. No effort is made to support Python 2 because it is considered obsolete as of early 2020.

2.5.5 Testing your installation

`GMSO` uses `py.test` to execute its unit tests. To run them, first install the `gmso-dev` environment from above as well as `gmso` itself

```
$ conda activate gmso-dev
$ pip install -e .
```

And then run the tests with the `py.test` executable:

```
$ py.test -v
```

2.5.6 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 `gmso-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
```

2.5.7 Building the documentation

`GMSO` 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 gmso-docs
$ make html
```

2.6 Using GMSO 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 GMSO with docker.

2.6.1 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.

2.6.2 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 *GMSO* installed:

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

If no command is provided to the container (as above), the container starts a `jupyter-notebook` at the (container) location `/home/anaconda/data`. To access the notebook, paste the notebook URL into a web browser on your computer. When you are finished, you can control-C to exit the notebook as usual. The docker container will exit upon notebook shutdown.

Alternatively, you can also start a Bourne shell to use python from the container's terminal:

```
$ docker run -it --mount type=bind,source=$(pwd),target="/home/anaconda/data" mosdef/
↪gmso:latest sh
~ $ source .profile
(gmso-dev) ~ $
```

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 additions 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.

2.6.3 Persisting User Volumes

If you will be using *GMSO* 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 *gmso-notebooks* in your local system, which will store all your *GMSO* 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/gmso-notebooks
$ cd /path/to/gmso-notebooks
$ docker run -it --name gmso --mount type=bind,source=$(pwd),target=/home/anaconda/data -
↪p 8888:8888 mosdef/gmso: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.

2.6.4 Running Python scripts in the container

Jupyter notebooks are a great way to explore new software and prototype code. However, when it comes time for production science, 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/
↪gmso: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 require a Jupyter notebook, but just want a Python interpreter, you can run:

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

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

```
$ docker run mosdef/gmso:latest python
```

2.6.5 Cleaning Up

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

```
$ docker container rm gmso
```

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

2.7 Contributing

Contributions are welcomed via pull requests on GitHub. Developers and/or users will review requested changes and make comments. The rest of this file will serve as a set of general guidelines for contributors.

2.7.1 Features

Implement functionality in a general and flexible fashion

GMSO is designed to be general and flexible, not limited to single chemistries, file formats, simulation engines, or simulation methods. Additions to core features should attempt to provide something that is applicable to a variety of use-cases and not targeted at only the focus area of your research. However, some specific features targeted toward a limited use case may be appropriate. Speak to the developers before writing your code and they will help you make design choices that allow flexibility.

2.7.2 Version control

We currently use the "standard" Pull Request model. Contributions should be implemented on feature branches of forks. Please try to keep the *main* branch of your fork up-to-date with the *main* branch of the main repository.

Propose a single set of related changes

Small changes are preferred over large changes. A major contribution can often be broken down into smaller PRs. Large PRs that affect many parts of the codebase can be harder to review and are more likely to cause merge conflicts.

2.7.3 Source code

Use a consistent style

It is important to have a consistent style throughout the source code. The following criteria are desired:

- Lines wrapped to 80 characters
- Lines are indented with spaces
- Lines do not end with whitespace
- For other details, refer to PEP8

To help with the above, there are tools such as flake8 and Black.

Document code with comments

All public-facing functions should have docstrings using the numpy style. This includes concise paragraph-style description of what the class or function does, relevant limitations and known issues, and descriptions of arguments. Internal functions can have simple one-liner docstrings.

2.7.4 Tests

Write unit tests

All new functionality in GMSO should be tested with automatic unit tests that execute in a few seconds. These tests should attempt to cover all options that the user can select. All or most of the added lines of source code should be covered by unit test(s). We currently use pytest, which can be executed simply by calling *pytest* from the root directory of the package.

INDEX

N

O

P

R

T

U

V

W