gmso

Release 0.12.0

Matt Thompson, Alex Yang, Ray Matsumoto, Parashara Shamapra

Jan 30, 2024

CONTENTS

This is the documentation for `GMSO`, the General Molecular Simulation Object. It is a part of the MoSDeF, the Molecular Simulation Design Framework.

CHAPTER

ONE

DESIGN PRINCIPLES

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!

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.

CHAPTER

TWO

DATA STRUCTURES IN GMSO

Following data structures are available within GMSO.

2.1 Core Classes

2.1.1 Topology

2.1.2 SubTopology

2.1.3 Atom

2.1.4 Bond

2.1.5 Angle

2.1.6 Dihedral

2.1.7 Improper

2.2 Potential Classes

2.2.1 AtomType

2.2.2 BondType

2.2.3 AngleType

2.2.4 DihedralType

2.2.5 ImproperType

2.2.6 PairPotentialType

2.3 ForceField

CHAPTER THREE

FORMATS

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

3.1 GROMACS

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

3.2 GSD

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

3.3 xyz

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

3.4 LAMMPS DATA

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

CHAPTER FOUR

EXTERNAL

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

4.1 mBuild

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

4.2 Parmed

Conversion methods for Parmed objects to and from `GMSO`.

4.3 OpenMM

Conversion methods for OpenMM objects to and from `GMSO`.

CHAPTER

FIVE

INSTALLATION

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
```

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 .
```

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 .
```

5.4 Supported Python Versions

Python 3.8-3.11 is the recommend version for users. It is the only version on which development and testing consistently takes place. Older (3.6-3.7) 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.

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
```

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
```

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
```

CHAPTER

SIX

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.

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.

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.

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.

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
```

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.

CHAPTER

SEVEN

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.

7.1 Features

7.1.1 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.

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 *master* branch of your fork up-to-date with the *master* branch of the main repository.

7.2.1 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.

7.3 Source code

7.3.1 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.

7.3.2 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.

7.4 Tests

7.4.1 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.