gmxbatch

Release 0.1.dev50+gacbf409.d20230907

Sep 07, 2023

Contents:

gmxbatch is a Python package for GROMACS scripting. It provides a high-level, object oriented interface to common tasks in the MD simulation workflow.

Warning: We intentionally abstract away much of the internals and fine details of molecular dynamics simulations with GROMACS. While this may come handy in the rapid development of simulation scripts, the user is cautioned and required to always check the validity of the input files produced by *gmxbatch*, especially the topologies and the MDP files. The authors of this package take no responsibility whatsoever for the correctness of the simulations, all responsibility lies with the user.

CHAPTER 1

Installation

1.1 Dependencies

gmxbatch depends on the following third party packages:

- matplotlib >=3.0.0
- jinja2
- numpy >= 1.15.0

For most functionality, you will also need a working installation of GROMACS.

1.2 Conda

```
conda install -c awacha gmxbatch
```

1.3 Pip

```
pip install gmxbatch
```

1.4 From source

```
git clone https://gitlab.com/awacha/gmxbatch.git
cd gmxbatch
git submodule init
git submodule update
python setup.py install
```

CHAPTER 2

Examples

Some examples on how to use *gmxbatch*

2.1 Example 1: Construct and equilibrate a box of methanol

2.1.1 0. Prerequisites

Install *gmxbatch*.

Create a directory and put the following files in it:

- **From the source distribution of *gmxbatch* (at https://gitlab.com/awacha/gmxbatch):**
  - meoh.gro from the *test/meoh* directory
  - meoh.itp from the *test/moleculetypes/charmm* directory
- The CHARMM36m force field bundle from Alexander MacKerell's site. Download and extract it.

Change to that directory.

2.1.2 1. Initialization

In the first part of a simulation script with *gmxbatch* is the initialization. This is probably the most important part as all parameters are decided here.

Let us first import the required packages

```
>>> import gmxbatch
>>> import matplotlib.pyplot as plt  # for plotting
```

Of course we will need a GMX engine:

```
>>> engine = gmxbatch.MDEngine('gmx')  # use the default 'gmx' executable on the $PATH
```

Define the force field we want to use. We choose the CHARMM36m force field, because it has parameters for methanol through CGenFF. You should already have downloaded and extracted it to your working directory. and extract it in your working directory. You should have a directory called 'charmm36-mar2019.ff'.

The force field object in *gmxbatch* comprises three aspects: the atom and interaction definitions (through the *forcefield.itp* topology include file), default MDP options, and a list of known molecule types. Molecule types for common solvents and ions are usually found in the force field directory in corresponding .itp files. When initializing a force field object, two required parameters are the ITP file of the force field itself and a search path for ITP files containing molecule types.

At present *gmxbatch* can handle Amber, CHARMM and GROMOS force fields, but it should be straightforward to extend the code with others. We use here the just downloaded CHARMM36m force field, and add the force field directory and the current directory as search paths for molecule type ITPs.

```
>>> ff = gmxbatch.CHARMM(itp='charmm36-mar2019.ff/forcefield.itp', moltypespath=[
↪'charmm36-mar2019.ff', '.'])
```

Now we define the environment, complete with a thermostat set at 300 K with 1 ps coupling time, and an isotropic Parrinello-Rahman barostat at 1 bar with 5 ps coupling time.

```
>>> env = gmxbatch.Environment(
...            thermostat=gmxbatch.Thermostat(
...                groups=['System'],  # only one coupling group: only solvent
...                ref_temperature=300,  # K
...                tau=1,  # ps
...                algorithm='V-rescale'  # Velocity rescaling algorithm for the _
↪production_ run
...            ),
...            barostat=gmxbatch.Barostat(
...                ref_pressure=1,  # bar
...                tau=5,  # ps
...                couplingtype='isotropic',  # isotropic system
...                compressibility=4.5e-5,  # 1/bar; compressibility of water
...                algorithm='Parrinello-Rahman'  # used for the _production_ run only
...            ))
```

Now assemble the system. Take the initial state from *meoh.gro* in your working directory

```
>>> conf = gmxbatch.Coordinates('meoh.gro')
```

Assemble the system

```
>>> system = gmxbatch.System(
...                name='Methanol',
...                forcefield=ff,
...                conf=conf,
...                moleculetypes=[ff.moleculetype('MEOH', 1, 'meoh.itp')],
...                indexgroups=None)
```

Finally, create the *Simulation* instance:

```
>>> sim = gmxbatch.Simulation(
...            engine=engine,
...            system=system,
...            environment=env)
```

Sort the atoms in the coordinate set to match the topology:

```
>>> sim.sortAtoms()
```

Now the initialization part is done.

2.1.3 2. Simulation

The strength of *gmxbatch* is in its simplicity when coming to this part. Many simulation tasks involving more calls to *gmx* subprograms are represented as one function call, as you will see below.

a. Adjust box size

First we adjust the box size. We put our single methanol molecule in a cubic box with 0.5 nm minimal distance from the molecule in the center.

```
>>> sim.rebox('cubic', 0.5)
```

Under the hood, a new file is generated, and *sim.conf* is updated to show the re-boxed structure.

b. Energy minimization of a single methanol molecule

Executing MD runs is really cheap:

```
>>> results = sim.em(nsteps=10000)
```

This also updates the underlying system. Additionally, each MD run returns a *Results* object with the trajectory, the final state of the run and a number of energy terms. We will see them later.

c. Replicate the molecules

Make a small box and do the replication:

```
>>> sim.rebox('cubic', 0.1)
>>> print(sim.system.moleculetypes[0])
MoleculeType: MEOH
   kind: Solvent
   count: 1
   itp: ../moleculetypes/charmm/meoh.itp
   atoms: 6
   posres macro: POSRES_MEOH
>>> print(len(sim.system.conf))
6
>>> sim.repeat(10, 10, 10, rot=True)  # 10×10×10 molecules, each is randomly oriented
>>> print(sim.system.moleculetypes[0])
MoleculeType: MEOH
   kind: Solvent
   count: 1000
   itp: ../moleculetypes/charmm/meoh.itp
   atoms: 6
   posres macro: POSRES_MEOH
>>> print(len(sim.system.conf))
6000
```

As you see, both the topology and the coordinate set has been updated.

d. Energy minimization

Do a short energy minimization on the box.

```
>>> emresults = sim.em(nsteps=50000)
>>> f=plt.figure()  # create a new Matplotlib figure
>>> emresults.energy['potential'].plot(figure=fig)  # plot the Potential energy vs.
↪"time"
```

e. Equilibration in the NVT ensemble

Perform equilibration in the NVT ensemble:

```
>>> nvtresults = sim.nvt(100, deffnm='nvt')  # 100 ps
>>> nvtresults.energy['potential'].plot(figure=plt.figure())  # plot the potential␣
↪energy vs. the time
>>> nvtresults.energy['temperature'].plot(figure=plt.figure())  # see if the␣
↪temperature has been stabilized or not
```

f. Compress the system

The methanol molecules are still very much apart. Apply a large pressure (e.g. 1000 bar) to compress them.

```
>>> env.barostat.ref_pressure = 1000  # bar
>>> npt1000results = sim.npt(200, deffnm='npt1000bar')  # 0.2 ns
>>> npt1000results.energy['potential'].plot(figure=plt.figure())  # plot the␣
↪potential energy vs. the time
>>> npt1000results.energy['temperature'].plot(figure=plt.figure())  # plot the␣
↪temperature vs. the time
>>> npt1000results.energy['volume'].plot(figure=plt.figure())  # plot the volume vs.␣
↪the time
```

g. Equilibration in the NpT ensemble

Now we set the reference pressure back to 1 bar and re-equilibrate the system at normal atmospheric pressure.

```
>>> env.barostat.ref_pressure = 1
>>> nptresults = sim.npt(200, deffnm='npt')  # 0.2 ns
>>> nptresults.energy['potential'].plot(figure=plt.figure())  # plot the potential␣
↪energy vs. the time
>>> nptresults.energy['temperature'].plot(figure=plt.figure())  # plot the␣
↪temperature vs. the time
>>> nptresults.energy['volume'].plot(figure=plt.figure())  # plot the volume vs. the␣
↪time
```

h. Production MD

Do a short production run on the equilibrated system

```
>>> mdresults = sim.md(1000, deffnm='md')  # 1 ns
>>> mdresults.energy['potential'].plot(figure=plt.figure())  # plot the potential␣
↪energy vs. the time
>>> mdresults.energy['temperature'].plot(figure=plt.figure())  # plot the temperature␣
↪vs. the time
>>> mdresults.energy['volume'].plot(figure=plt.figure())  # plot the volume vs. the␣
↪time
>>> sim.conf.write('meoh1000final.gro')  # write the resulting state to an output file
```

CHAPTER 3

Origins of the project

This project—as probably many others—has been born from frustration. People working with computer simulation sooner or later encounter the need of automation, when faced with increasing simulation demands and projects with very similar workflows. A first solution is typically a simple shell script consisting of the sequence of commands. Some operations, especially file edits are not easy to represent in the form of command line utilities. Another challenge is decisions and conditionals, for which most shells (namely BASH, CSH and their variants) have limited and/or cumbersome facilities.

These advanced operations can be readily solved by a more complete programming language, for instance Python. Inserting Python snippets into the shell scripts (e.g. with "here documents" using the "<<" operator in BASH) mitigates the problem, but the resulting scripts tend to be large and unmaintainable. The largest advantage of BASH over Python is the straightforward way to run programs.

The autors of this project experienced that the scripts developed for their workflows tended to contain more and more Python blocks, and much of the other code was information transfer between Python and BASH. When the Python:BASH ratio reached around 1:1, we thought to turn the tables and choose Python as the main language and replace the remaining BASH constructs in a Pythonic, object oriented way. This also came with the advantage that many common operations (such as calling *grompp* or *solvate*) can be abstracted away. Of course, abstraction comes with the danger of turning parts of the workflow into black boxes and taking away / hiding important decisions from the end user. The end user of this package is therefore encouraged to always check the files produced, especially coordinate sets, topologies and molecular dynamics parameter files (.mdp).

We also intended to make this package as portable as possible. We are aware of the existence of *gmxapi*, the official Python bindings of GROMACS supplied with the newest versions. Because it directly uses the shared libraries of GROMACS, it is very fast. However, it is not easily portable between versions. In *gmxbatch* we depend on and use the command line interface by invoking *gmx* subprograms where needed. This results in a one-size-fits-(almost)-all package, where the exact program version is selected by the user. We also implemented a higher level object oriented interface, letting the user focus on the physics/chemistry instead of the exact parametrization of GROMACS commands.

CHAPTER 4

Design Philosophy

At this point this is just a jumble of decisions

1. Multi-layer approach: high level API for casual users, gmx commands are also exposed for power users
2. Auto-generating the overall topology. Topology file generation must be cheap.
3. Molecule topologies are to be given by the user (i.e. functionality of pdb2gmx is not implemented)
4. Generate files for each run separately, with a common name prefix (“deffnm”)
5. Lazy loading where possible
6. Safe default values according to the best practice
7. All responsibility lies with the user]:-P
8.

CHAPTER 5

Class hierarchy

Fig. 1: Class hierarchy of *gmxbatch*

5.1 Coordinates

Stores a conformation. Input/output from/to GRO and G96 files is implemented. A lazy loading mechanism is in place: by default when a new instance is created with a file name, the actual data set is not loaded from the file automatically, only when first accessed.

5.2 MDEngine

This class implements low level access to *gmx* subcommands. Most commands have a corresponding method of this class. Adding more commands should be a straightforward business.

5.3 Environment

This class contains information on external conditions, e.g. temperature and pressure coupling (or external pulling forces when they will be implemented).

5.4 ForceField

Contains information on atom types and the forces acting between various atom types, i.e. the information typically contained in the GROMACS force field directory (e.g. charmm27.ff). It also stores a list of supported molecule types. When constructing a *System*, molecule types are chosen from this list and the appropriate counts are set.

5.5 IndexGroups

GROMACS makes extensive use of index groups for at varios points of the simulation workflow. This class holds a list of defined index groups. At each simulation run, an index file is written with the current file name prefix.

5.6 IntermolecularInteractions

At the end of a GROMACS topology, after the *[molecules]* block intermolecular interactions can be defined in the *[intermolecular_interactions]* section. Presently bonds, angles and dihedrals can be defined by global atom numbers, function type numbers and the required parameters for the function. See the GROMACS manual for details.

5.7 MDP

5.8 MoleculeType

As by design we don't implement *pdb2gmx* functionality, the user is responsible to create the ITP files containing *[moleculetype]* blocks for all molecules she wants to use. This typically involves running *pdb2gmx* and editing the resulting *topol.top* file to contain only the *[moleculetype]* section. Position restraints should also be inserted in this file. We supply the tool *top2itp* to aid this conversion.

MoleculeType instances have the following important attributes:

- name: molecule type name
- kind: molecule kind: Solute, Solvent, Ion
- itpfile: the corresponding topology include file
- count: the number of molecules of this type in the system
- posresdefine: preprocessor macro for *grompp* which activates position restraints (e.g. POSRES)
- atoms: atom information read from the ITP file

5.9 Results

5.10 Simulation

This is the main class, encompasses all aspects of an MD simulation. Its main attributes are *system* and *environment*. Most operations (e.g. energy minimization, system replication, solvation, NVT and NpT equilibration, production MD run etc.) are exposed as methods.

5.11 System

Represents the system being simulated, consisting of the coordinate set, the topology, the force field, index groups and intermolecular interactions. However, its main responsibility is to store the information on topology and create a topology file when needed. Operations exposed in the *Simulation* class change the system automatically, i.e. after an MD run the coordinate set is updated to the final state of the run.

5.12 Trajectory

5.13 XVGFile

CHAPTER 6

Indices and tables

- genindex
- modindex
- search