MDpow Documentation

Release 0.8.0+91.g120c5dd

Ian Kenney, Bogdan Iorga, and Oliver Beckstein

Sep 07, 2023

Contents

Release 0.8.0+91.g120c5dd

Date Sep 07, 2023

MDPOW is a python package that automates the calculation of solvation free energies via molecular dynamics (MD) simulations. In particular, it facilitates the computation of partition coeffcients. Currently implemented:

- *water-octanol* partition coefficient (P_{OW})
- *water-cyclohexane* partition coefficient (P_{CW})
- *water-toluene* partition coefficient (P_{TW})

The package is being actively developed and incorporates recent ideas and advances. If something appears unclear or just wrong, then please ask questions on the MDPOW Issue Tracker.

CHAPTER 1

MD engine

Calculations are performed with the Gromacs molecular dynamics (MD) software package[1]. MDPOW is tested with

- Gromacs 4.6.5
- Gromacs 2018.6
- Gromacs 2020.6
- Gromacs 2021.1
- Gromacs 2022.4
- Gromacs 2023.1

but versions 5.x, 2016.x, and 2019.x should also work. It should be possible to use any of these Gromacs versions without further adjustments, thanks to the underlying GromacsWrapper library[1].

Nevertheless, you should *always* check the topology and runinput (mdp) files for the version of Gromacs that you are using.

[1] The package is built on top of the GromacsWrapper framework (which is automatically installed).

CHAPTER 2

Force fields

Currently

- OPLS-AA
- CHARMM/CGenFF
- AMBER/GAFF

are supported. In principle it is possible to add force fields sets by changing the `GMXLIB` environment variable and providing appropriate template files but this is currently untested.

A number of different *water models* are supporte (see *mdpow.forcefields.GROMACS_WATER_MODELS*).

See also:

mdpow.forcefields

CHAPTER 3

Required input

As *input*, the user only needs to provide a structure file (PDB or GRO) and a Gromacs ITP file containing the parametrization of the small molecule (e.g. from LigandBook or ParamChem).

CHAPTER 4

Version information

MDPOW uses semantic versioning with the release number consisting of a triplet *MAJOR.MINOR.PATCH*. *PATCH* releases are bug fixes or updates to docs or meta data only and do not introduce new features or change the API. Within a *MAJOR* release, the user API is stable except during the development cycles with MAJOR = 0 where the API may also change (rarely) between MINOR releases. *MINOR* releases can introduce new functionality or deprecate old ones.

The version information can be accessed from the attribute *mdpow.__version__*.

mdpow.**__version__ = '0.8.0+91.g120c5dd.dirty'**
str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

CHAPTER 5

Limitations and known issues

For current issues and open feature requests please look through the MDPOW Issue Tracker. Some of the major open issues are:

- GROMACS versions < 2021 can silently produce incorrect free energy estimates because exclusions are not properly accounted for for solutes larger than the rlist cutoff when the `couple-intramol = no` decoupling is used (as it is in all of MDPOW), see https://gitlab.com/gromacs/gromacs/-/issues/3403. MDPOW does not detect this situation and does not offer a workaround (namely doing separate vacuum simulations and use `couple-intramol = yes`). GROMACS 2021 at least fails when the failure condition occurs (see https://gitlab.com/gromacs/gromacs/-/merge_requests/861).
- Only free energy calculations of neutral solutes are supported; the workflow also does not include addition of ions (see issue #97).
- Mixed solvents (octanol and water) are only supported with the included template topology files for GROMACS >= 2018.
- Adding new solvents requires modifying the MDPOW code; instead it should be configurable.

5.1 Quick installation instructions for *MDPOW*

MDPOW is compatible with Python >=3.8 and tested on Ubuntu and Mac OS.

We recommend that you install MDPOW in a virtual environment.

5.1.1 Installation for users

Most users should follow these instructions.

We use the Anaconda distribution with the `conda` command to manage dependencies (see installing the Anaconda distribution for details on how to set up `conda`).

To run MDPOW you will also need to install a compatible version of GROMACS.

5.1.1.1 Conda environment with pre-requisites

Make a conda environment with the latest packages for Python 3.8 or higher with the name *mdpow*; this installs the larger dependencies that are pre-requisites for MDPOW:

```
conda create -c conda-forge -n mdpow  numpy scipy matplotlib seaborn mdanalysis␣
↪pyyaml alchemlyb pandas gromacswrapper rdkit
conda activate mdpow
```

Install MDPOW with `pip`:

```
pip install mdpow
```

5.1.1.2 Installation from source

Instead of the pip-installation of MDPOW, you can also install from source:

```
conda activate mdpow
git clone https://github.com/Becksteinlab/MDPOW.git
pip install ./MDPOW
```

(Older releases are available but outdated; use the latest source for now.)

5.1.1.3 Checking that the installation worked

Check that you can run the `mdpow-*` commandline tools:

```
mdpow-equilibrium --help
```

should print a whole bunch of messages. If your GROMACS installation cannot be found, make sure you `source GMXRC` or load the appopriate modules or whatever else you have to do so that the `gmx` command is found. See https://gromacswrapper.readthedocs.io for more details.

Check that you can import the module:

```
python
>>> import mdpow
>>> help(mdpow)
```

In case of problems file an issue at https://github.com/Becksteinlab/MDPOW/issues

5.1.2 Developer installation

A development install is useful while hacking away on the code:

```
cd MDPOW
pip install -e  .
```

5.2 `mdpow` — Computing the octanol/water partitioning coefficient

The `mdpow` module helps in setting up and analyzing absolute free energy calculations of small molecules by molecular dynamics (MD) simulations. By computing the hydration free energy and the solvation free energy in octanol

one can compute the octanol/water partitioning coefficient, an important quantity that is used to characterize drug-like compounds.

The MD simulations can be performed with all recent versions of Gromacs, starting with Gromacs 4.6.x.

5.2.1 How to use the module

Before you can start you will need

- a coordinate file for the small molecule
- a Gromacs OPLS/AA topology (itp) file
- an installation of Gromacs in a *supported version*.

5.2.1.1 Basic work flow

You will typically calculate two solvation free energies (free energy of transfer of the solute from the liquid into the vacuum phase):

1. solvent = *water*
    1. set up a short equilibrium simulation of the molecule in a *water* box (and run the MD simulation);
    2. set up a free energy perturbation calculation of the ligand in water , which will yield the hydration free energy;
2. solvent = *octanol*
    1. set up a short equilibrium simulation of the molecule in a *octanol* box (and run the MD simulation);
    2. set up a free energy perturbation calculation of the ligand in octanol , which will yield the solvation free energy in octanol;
3. run these simulations on a cluster;
4. analyze the output and combine the free energies to arrive at an estimate of the octanol-water partition coefficient;
5. plot results using `mdpow.analysis.plot_exp_vs_comp()`.

5.2.1.2 Customized submission scripts for queuing systems

One can also generate run scripts for various queuing systems; check the documentation for `gromacs.qsub` and in particular the section on writing queuing system templates . You will have to

- add a template script to your private GromacsWrapper template directory (`~/.gromacswrapper/qscripts`); in this example we call it `my_script.sge`;
- add the keyword *qscript* to the *`mdpow.equil.Simulation.MD()`* and *`mdpow.fep.Gsolv.setup()`* invocations; e.g. as

    qscript = ['my_script.sge', 'local.sh']

- submit the generated queuing system script to your queuing system, e.g.

```
cd Equilibrium/water
qsub my_script.sh
```

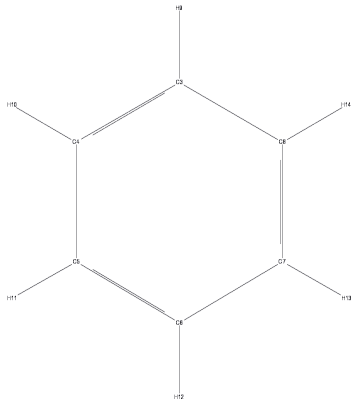
5.2.2 The mdpow scripts

Some tasks are simplified by using scripts, which are installed in a bin directory (or the directory pointed to by `python setup.py --install-scripts`). See *The mdpow-* scripts* for details and *Tutorial: Using the mdpow scripts to compute log POW of benzene* for an example. The scripts essentially execute the steps shown in *Tutorial: Manual session: 1-octanol as a solute* so in order to gain a better understanding of MDpow it is suggested to look at both tutorials.

5.2.3 Tutorial: Using the mdpow scripts to compute log P_{OW} of benzene

The most straightforward use of MDpow is through the Python scripts described in *The mdpow-* scripts*. This tutorial shows how to use them to calculate log P_{OW} for benzene.

The `examples/benzene` directory of the distribution contains a Gromacs OPLS/AA topology for benzene (`benzene.itp`) and a starting structure (`benzene.pdb`) together with a *run input configuration file* for the mdpow scripts (`benzene.yml`) in YAML format.



1. Make a directory `benzene` and collect the files in the directory.
2. Move into the parent directory of `benzene`; in this tutorial all files will be generated under `benzene` but the configuration file has recorded the location of the input files as `benzene/filename`. (For more on to make best use of file names in the configuration file see *Advanced: Separating input from output data*.)
3. Set up Gromacs (e.g. by sourcing `GMXRC`).

   Note: Gromacs must be set up so that all Gromacs commands (such as **`mdrun`** or **`grompp`** can be found on the shell's `PATH`). If this is not the case then the following will fail. Look at the **log file** (named **mdpow.log**) for error messages.)

4. Generate the input files for a equilibrium simulation of benzene in **water** (TIP4P) and run the simulation:

   ```
   mdpow-equilibrium --solvent water benzene/benzene.yml
   ```

   The example file provided will only set up very short simulations and it will also directly call **`mdrun`** to run the simulations. This is configured via the `runlocal = True` parameter in the `MD_relaxed` and `MD_NPT` sections in the `benzene.yml` file.

   In fact, the following steps are carried out:

   1. generate a system topology (section `Setup`)
   2. solvate the compound (section `Setup`)

3. energy minimise the system (section `Setup`)
4. run a short relaxation with a time step of 0.1 fs in order to remove any steric clashes; this step was found to enable much more robust simulations, especially when using octanol as a solvent. Section `MD_relaxed` controls this step.
5. run a equilibrium MD simulation at constant pressure and temperature using a time step of 2 fs. Section `MD_NPT` controls this step.

5. Generate the input files for a equilibrium simulation of benzene in **octanol** (OPLS/AA parameters) and run the simulation:

```
mdpow-equilibrium --solvent octanol benzene/benzene.yml
```

   The steps are analogous to the ones described under 4.

6. Use the last frame of the equilibrium simulation as a starting point for the free energy perturbation (FEP) calculations. This step is controlled by the `FEP` section in the run input configuration file.

   The example only runs very short windows (`runtime: 25` ps) but it will run *all* 21 individual simulations sequentially (`runlocal: True`). Thus it is recommended to run this example on a fast multi-core workstation and at least Gromacs 4.5.x, which has thread support for **`mdrun`**.

   The FEP windows for benzene in **water** are generated and run by

```
mdpow-fep --solvent water benzene/benzene.yml
```

   (In order to run the FEP windows on a cluster see *Advanced: Generating queuing system scripts*.)

7. The FEP windows for benzene in **octanol** are generated and run by

```
mdpow-fep --solvent octanol benzene/benzene.yml
```

8. Analyse the simulation output with *mdpow-pow*. It collects the raw data from the FEP simulations and computes the free energies of solvation using thermodynamic integration (TI) together with error estimates. log P_{OW} is calculated from the difference of the octanol and water solvation free energies:

```
mdpow-pow benzene
```

   `benzene` is the directory name under which the FEP simulations are stored. By default, results are *appended* to the files `energies.txt` and `pow.txt`.

   See also:

   The output formats are explained with examples in *Output data file formats*.

5.2.3.1 Advanced: Generating queuing system scripts

(To be written)

See also:

`gromacs.qsub` for the reference on how queuing system script templates are handled and processed

5.2.3.2 Advanced: Separating input from output data

Make another directory under which simulation data will be stored; in this tutorial it will be called `WORK` and we assume that all directories reside in `~/Projects/POW`. We will end up with a directory layout under `~/Projects/POW` like this:

```
benzene/
       benzene.itp
       benzene.pdb
       benzene.yml
WORK/
     benzene/
             water/
             octanol/
```

Edit `benzene.yml` to put in *absolute paths* to the input files: In the `setup` section use absolute paths to the itp and pdb files of benzene:

```
setup:
  name: "benzene"
  molecule:  "BNZ"
  structure: "~/Projects/POW/benzene/benzene.pdb"
  itp: "~/Projects/POW/benzene/benzene.itp"
```

With absolute paths defined, it is easy to generate all other files under `WORK/benzene` (the directory name "benzene" is the *name* entry from the `setup` section of the configuration file):

```
cd WORK
mdpow-equilibrium --solvent water ../benzene/benzene.yml
mdpow-equilibrium --solvent octanol ../benzene/benzene.yml
mdpow-fep --solvent water ../benzene/benzene.yml
mdpow-fep --solvent octanol ../benzene/benzene.yml
```

Finally, calculate log P_{OW}:

```
cd WORK
mdpow-pow benzene
```

5.2.4 Tutorial: Manual session: 1-octanol as a solute

In the following interactive python session we use octanol as an example for a solute; all files are present in the package so one can work through the example immediately.

Before starting **`python`** (preferrably ipython) make sure that the Gromacs 4.6.x tools can be found, e.g. `which grompp` should show you the path to **`grompp`**.

5.2.4.1 Water

Equilibrium simulation

Make a directory `octanol` and copy the `octanol.itp` and `octanol.gro` file into it. Launch **`ipython`** from this directory and type:

```
import mdpow.equil
S = mdpow.equil.WaterSimulation(molecule="OcOH")
S.topology(itp="octanol.itp")
S.solvate(struct="octanol.gro")
S.energy_minimize()
S.MD_relaxed()
# run the simulation in the MD_relaxed/ directory
```

(continues on next page)

(continued from previous page)

```
S.MD(runtime=50, qscript=['my_script.sge', 'local.sh'])   # only run for 50 ps in this␣
↪tutorial
S.save("water.simulation")                                # save setup for later␣
↪(analysis stage)
```

Background (`Ctrl-Z`) or quit (`Ctrl-D`) python and run the simulations in the `MD_relaxed` and `MD_NPT` subdirectory. You can modify the `local.sh` script to your ends or use *qscript* to generate queuing system scripts.

Note: Here we only run 50 ps equilibrium MD for testing. For production this should be substantially longer, maybe even 50 ns if you want to extract thermodynamic data.

Hydration free energy

Reopen the python session and set up a *Ghyd* object:

```
import mdpow.fep
gwat = mdpow.fep.Ghyd(molecule="OcOH", top="Equilibrium/water/top/system.top.template
↪", struct="Equilibrium/water/MD_NPT/md.pdb", ndx="Equilibrium/water/solvation/main.
↪ndx", runtime=100)
```

Alternatively, one can save some typing if we continue the last session and use the *mdpow.equil.Simulation* object (which we can re-load from its saved state file from disk):

```
import mdpow.equil
S = mdpow.equil.WaterSimulation(filename="water.simulation")  # only needed when quit
gwat = mdpow.fep.Ghyd(simulation=S, runtime=100)
```

This generates all the input files under `FEP/water`.

Note: Here we only run 100 ps per window for testing. For production this should be rather something like 5-10 ns (the default is 5 ns).

Then set up all input files:

```
gwat.setup(qscript=['my_script.sge', 'local.sh'])
```

(The details of the FEP runs can be customized by setting some keywords (such as *lambda_vdw*, *lamda_coulomb*, see *mdpow.fep.Gsolv* for details) or by deriving a new class from the *mdpow.fep.Ghyd* base class but this is not covered in this tutorial.)

5.2.4.2 Octanol

Equilibrium simulation

Almost identical to the water case:

```
O = mdpow.equil.OctanolSimulation(molecule="OcOH")
O.topology(itp="octanol.itp")
O.solvate(struct="octanol.gro")
```

(continues on next page)

(continued from previous page)

```
O.energy_minimize()
O.MD_relaxed()
O.MD(runtime=50, qscript=['my_script.sge', 'local.sh'])     # only run for 50 ps in␣
↪this tutorial
O.save()
```

Note: Here we only run 50 ps equilibrium MD for testing. For production this should be substantially longer, maybe even 50 ns if you want to extract thermodynamic data.

Octanol solvation free energy

Almost identical setup as in the water case:

```
goct = mdpow.fep.Goct(simulation=O, runtime=100)
goct.setup(qscript=['my_script.sge', 'local.sh'])
```

This generates all the input files under `FEP/octanol`.

Note: Here we only run 100 ps per window for testing. For production this should be rather something like 5-10 ns (the default is 5 ns)

5.2.4.3 Running the FEP simulations

The files are under the `FEP/water` and `FEP/octanol` directories in separate sub directories.

Either run job arrays that should have been generated from the `my_script.sge` template

```
qsub Coul_my_script.sge
qsub VDW_my_script.sge
```

Or run each job in its own directory. Note that **`mdrun`** should be called with at least the following options

```
mdrun -deffnm $DEFFNM  -dgdl
```

where DEFFNM is typically "md"; see the run `local.sh` script in each direcory for hints on what needs to be done.

5.2.4.4 Analyze output and log P_{OW} calculation

For the water and octanol FEPs do

```
gwat.collect()
gwat.analyze()

goct.collect()
goct.analyze()
```

The analyze step reports the estimate for the free energy difference.

Calculate the free energy for transferring the solute from water to octanol and *octanol-water partition coefficient* log P_OW

```
mdpow.fep.pOW(gwat, goct)
```

(see *mdpow.fep.pOW()* for details and definitions).

All individual results can also accessed as a dictionary

```
gwat.results.DeltaA
```

Free energy of transfer from water to octanol:

```
goct.results.DeltaA.Gibbs - gwat.results.DeltaA.Gibbs
```

The individual components are

Gibbs total free energy difference of transfer from solvent to vacuum at the Ben-Naim standard state (i.e. 1M solution/1M gas phase) in kJ/mol;

DeltaG0 = (G_solv - G_vac)

We calculate the Gibbs free energy (at constant pressure P) by using the *NPT* ensemble for all MD simulations.

coulomb contribution of the de-charging process to DeltaG

vdw contribution of the de-coupling process to DeltaG

To plot the data (de-charging and de-coupling):

```
import pylab
gwat.plot()
pylab.figure()
goct.plot()
```

For comparison to experimental values see `mdpow.analysis`.

5.2.4.5 Error analysis

The data points are the (time) **average <G>** of G = dV/dl over each window. The **error bars** s_G are the error of the mean <G>. They are computed from the auto-correlation time of the fluctuations and the standard deviation (see Frenkel and Smit, p526 and `numkit.timeseries.tcorrel()`):

```
s_G**2 = 2*tc*acf(0)/T
```

where tc is the decay time of the ACF of <(G-<G>)**2> (assumed to follow f(t) = exp(-t/tc) and hence calculated from the integral of the ACF to its first root); T is the total runtime.

Errors on the energies are calculated via the propagation of the errors s_G through the thermodynamic integration and the subsequent thermodynamic sums (see `numkit.integration.simps_error()` `numkit.observables.QuantityWithError` for details).

- If the graphs do not look smooth or the errors are large then a longer *runtime* is definitely required. It might also be necessary to add additional lambda values in regions where the function changes rapidly.
- The errors on the Coulomb and VDW free energies should be of similar magnitude because there is no point in being very accurate in one if the other is inaccurate.
- For water the "canonical" lambda schedule produces errors <0.5 kJ/mol (or sometimes much better) in the Coulomb and VDW free energy components.

- For octanol the errors on the coulomb dis-charging free energy can become large (up to 4 kJ/mol) and thus completely swamp the final estimate. Additional lambdas 0.125 and 0.375 should improve the precision of the calculations.

5.3 The `mdpow-*` scripts

A number of python scripts are installed together with the `mdpow` package. They simplify some common tasks (especially at the analysis stage) but they make some assumptions about directory layout and filenames. If one uses defaults for all directory and filename options then it should "just work".

In particular, a directory hierarchy such as the following is assumed:

```
moleculename/
     water.simulation
     octanol.simulation
     Equilibrium/
           water/
           octanol/
     FEP/
           water/
                 Ghyd.fep
                 Coulomb/
                 VDW/
           octanol/
                 Goct.fep
                 Coulomb/
                 VDW/
```

moleculename is, for instance, "benzene" or "amantadine"; in the run input file (see *Equilibrium simulations*) is the value of the variable `name` in the `[setup]` section.

5.3.1 Run input file

The `mdpow-*` scripts are the **commandline interface** (CLI) to the Python API functionality in the `mdpow` package. Whereas the Python API requires passing of parameters to classes and functions and therefore exposes the full flexibility of MDPOW, the CLI works with a narrower set of options which are collected in a **run input file**. This run input file or *RUNFILE* is named `runinput.yml` by default.

A template *RUNFILE* can be generated with **`mdpow-get-runinput`**

```
mdpow-get-runinput runinput.yml
```

which will copy the default run input file bundled with MDPOW and put it in the current directory under the name `runinput.yml`.

For an example of a *RUNFILE* see `benzene.yml`. The comments in the file serve as the documentation.

The run input file uses YAML syntax (and is parsed by `yaml`).

Note: It is recommended to use absolute paths to file names.

Note: It is recommended to enclose all strings in the input file in quotes, especially if they can be interpreted as numbers. For example, a name "005" would be interpreted as the number 5 unless explicitly quoted.

5.3.2 Equilibrium simulations

The **mdpow-equilibrium** script

- sets up equilibrium MD simulations for the solvents (e.g., *water* or *octanol*)
- runs **energy minimization**, **MD_relaxed**, and **MD_NPT** protocols; the user can choose if she wants to launch **mdrun** herself (e.g. on a cluster) or let the script do it locally on the workstation

The script runs essentially the same steps as described in the tutorial *Tutorial: Manual session: 1-octanol as a solute* but it gathers all required parameters from the *run input file* and it allows one to stop and continue and the protocol transparently.

It requires as at least Gromacs 4.6.5 ready to run (check that all commands can be found). The required **input** is

1. a run configuration file (*runinput.yml*);
2. a structure file (PDB or GRO) for the compound
3. a Gromacs ITP file for the compound (OPLS/AA force field)

The script keeps track of the stages of the simulation protocol (in the state files `water.simulation`, `octanol.simulation` etc) and allows the user to **restart from the last completed stage**. For instance, one can use the script to set up a simulation, then run the simulation on a cluster, transfer back the generated files, and start **mdpow-equilibrium** again with the exact same input to finish the protocol. Since Gromacs 4.5 it is also possible to interrupt a running **mdrun** process (e.g. with `Control-c`) and then resume the simulation at the last saved trajectory checkpoint by running **mdpow-equilibrium** again.

If in doubt, just try running **mdpow-equilibrium** running again and let it figure out the best course of action. Look at the log file to see what has been done. Lines reading "Fast forwarding: *stage*" indicate that results from *stage* are available.

Usage of the command:

> **mdpow-equilibrium** [options] *RUNFILE*
>
> Set up (and possibly run) the equilibration equilibrium simulation for one compound and one solvent. All parameters except the solvent are specified in the *RUNFILE*.
>
> Arguments:
>
> **RUNFILE**
> The runfile `runinput.yml` with all configuration parameters.
>
> Options:
>
> **-h, --help**
> show this help message and exit
>
> **-S** <NAME>**, --solvent**=<NAME>
> solvent <NAME> for compound, can be 'water', 'octanol', 'cyclohexane' [water]
>
> **-d** <DIRECTORY>**, --dirname**=<DIRECTORY>
> generate files and directories in <DIRECTORY>, which is created if it does not already exist. The default is to use the molecule *name* from the run input file.

5.3.3 FEP simulations

The **mdpow-fep** script sets up (and can also run) the free energy perturbation calculations for one compound and one solvent. It uses the results from *mdpow-equilibrium* together with the run input file.

You will require:

1. at least Gromacs 4.6.5 ready to run (check that all commands can be found)
2. the results from a previous complete run of **mdpow-equilibrium**

Usage of the command:

mdpow-fep [options] *RUNFILE*

Arguments:

RUNFILE
The runfile `runinput.yml` with all configuration parameters.

Options:

-h, --help
show this help message and exit

-S <NAME>, **--solvent**=<NAME>
solvent <NAME> for compound, can be 'water' or 'octanol' [water]

-d <DIRECTORY>, **--dirname**=<DIRECTORY>
generate files and directories in <DIRECTORY>, which is created if it does not already exist. The default is to use the molecule *name* from the run input file.

5.3.4 Running analysis

5.3.4.1 Solvation free energy

The **mdpow-solvationenergy** calculatest the solvation free energy. It

- collects data from FEP simulations (converting EDR files to XVG.bz2 when necessary)
- calculates desolvation free energies for *solvent* –> vacuum via thermodynamic integration (TI)
- plots dV/dlambda graphs
- appends results to `energies.txt` (when the default names are chosen), see *Output data file formats*.

Usage of the command:

mdpow-solvationenergy [options] DIRECTORY [DIRECTORY ...]

Run the free energy analysis for a solvent in <DIRECTORY>/FEP and return DeltaG.

DIRECTORY should contain all the files resulting from running `mdpow.fep.Ghyd.setup()` (or the corresponding `Goct.setup()` or `Gcyclohexane.setup()` and the results of the MD FEP simulations. It relies on the canonical naming scheme (basically: just use the defaults as in the tutorial).

The dV/dlambda plots can be produced automatically (`--plotfile auto`). If multiple DIRECTORY arguments are provided then one has to choose the "auto" option (or "none").

The total solvation free energy is calculated as

$$\Delta G^* = -(\Delta G_{\mathrm{coul}} + \Delta G_{\mathrm{vdw}})$$

Note that the standard state refers to the "Ben-Naim" standard state of transferring 1 M of compound in the gas phase to 1 M in the aqueous phase.

Results are *appended* to a data file with **Output file format**:

```
.                 ---------- kJ/mol ---
molecule solvent  DeltaG*  coulomb  vdw
```

All observables are quoted with an error estimate, which is derived from the correlation time and error propagation through Simpson's rule (see *mdpow.fep.Gsolv()*). It ultimately comes from the error on <dV/dlambda>, which is estimated as the error to determine the average.

molecule molecule name as used in the itp

DeltaG* solvation free energy vacuum –> solvent, in kJ/mol

coulomb discharging contribution to the DeltaG*

vdw decoupling contribution to the DeltaG*

directory folder in which the simulations were stored

positional arguments:

DIRECTORY [DIRECTORY ...]
directory or directories which contain all the files resulting from running *mdpow.fep.Ghyd.setup()*

optional arguments:

-h, --help
show this help message and exit

--plotfile FILE
plot dV/dlambda to FILE; use png or pdf suffix to determine the file type. *'auto'* generates a pdf file *DIRECTORY*/dVdl*_molname_solvent*.pdf and *none* disables it The plot function is only available for mdpow estimator,and is disabled when using alchemlyb estimators. (default: none)

--solvent NAME, **-S** NAME
solvent *NAME* for compound, 'water', 'octanol', or 'cyclohexane' (default: water)

-o FILE, **--outfile** FILE
append one-line results summary to *FILE* (default: gsolv.txt)

-e FILE, **--energies** FILE
append solvation free energies to FILE (default: energies.txt)

--estimator {mdpow,alchemlyb}
choose the estimator to be used, *alchemlyb* or *mdpow* estimators (default: *alchemlyb*)

--method {TI,MBAR,BAR}
choose the method to calculate free energy (default: *MBAR*)

--force
force rereading all data (default: False)

--SI
enable statistical inefficiency (SI) analysis. Statistical inefficiency analysis is performed by default when usingalchemlyb estimators and is always disabled when using mdpow estimator. (default: True)

--no-SI
disable statistical inefficiency analysis. Statistical inefficiency analysis is performed by default when usingalchemlyb estimators and is disabled when using mdpow estimator. (default: False)

-s N, **--stride** N
use every *N*-th datapoint from the original dV/dlambda data. (default: 1)

--start START
start point for the data used from the original dV/dlambda data. (default: 0)

--stop STOP
stop point for the data used from the original dV/dlambda data. (default: None)

--ignore-corrupted
skip lines in the md.xvg files that cannot be parsed. (default: False)

> **Warning:** Other lines in the file might have been corrupted in such a way that they appear correct but are in fact wrong. WRONG RESULTS CAN OCCUR! USE AT YOUR OWN RISK

It is recommended to simply rerun the affected simulation(s).

5.3.4.2 Partition coefficients

The **mdpow-pow** script

- collects data from FEP simulations
- calculates desolvation free energies for octanol –> vacuum and water –> vacuum via thermodynamic integration (TI)
- calculates transfer free energy water –> octanol
- calculates log P_{OW}
- plots dV/dlambda graphs
- appends results to `pow.txt` and `energies.txt` (when the default names are chosen), see *Output data file formats*.

An equivalent script **mdpow-pcw** for the water-cyclohexane partition coefficient is also included.

Usage of the command:

mdpow-pow [options] *DIRECTORY* [*DIRECTORY* ...]

Run the free energy analysis for water and octanol in *DIRECTORY*/FEP and return the octanol-water partition coefficient log P_{OW}.

DIRECTORY should contain all the files resulting from running *mdpow.fep.Goct.setup()* and *mdpow.fep.Goct.setup()* and the results of the MD FEP simulations. It relies on the canonical naming scheme (basically: just use the defaults as in the tutorial).

The dV/dlambda plots can be produced automatically (`--plotfile auto`). If multiple *DIRECTORY* arguments are provided then one has to choose the "auto" option (or "none").

positional arguments:

DIRECTORY [DIRECTORY ...]
One or more directories that contain the state pickle files (`water.simulation`, `octanol.simulation`) for the solvation free energy calculations in water and octanol. These directory or directories should contain all the files resulting from running *mdpow.fep.Ghyd.setup()* and *mdpow.fep.Goct.setup()* and the results of the MD FEP simulations.

optional arguments:

-h, --help
show this help message and exit

--plotfile FILE
plot dV/dlambda to FILE; use png or pdf suffix to determine the file type. *'auto'* generates a pdf file *DIRECTORY*/dVdl_*molname*_pow.pdf and *none* disables it The plot function is only available for mdpow estimator,and is disabled when using alchemlyb estimators. (default: none)

-o FILE, **--outfile** FILE
append one-line results summary to *FILE* (default: pow.txt)

-e FILE, **--energies** FILE
append solvation free energies to FILE (default: energies.txt)

--estimator {mdpow,alchemlyb}
choose the estimator to be used, *alchemlyb* or *mdpow* estimators (default: *alchemlyb*)

--method {TI,MBAR,BAR}
choose the method to calculate free energy (default: *MBAR*)

--force
force rereading all data (default: False)

--SI
enable statistical inefficiency (SI) analysis. Statistical inefficiency analysis is performed by default when usingalchemlyb estimators and is always disabled when using mdpow estimator. (default: True)

--no-SI
disable statistical inefficiency analysis. Statistical inefficiency analysis is performed by default when usingalchemlyb estimators and is disabled when using mdpow estimator. (default: False)

-s N, **--stride** N
use every *N*-th datapoint from the original dV/dlambda data. (default: 1)

--start START
start point for the data used from the original dV/dlambda data. (default: 0)

--stop STOP
stop point for the data used from the original dV/dlambda data. (default: None)

--ignore-corrupted
skip lines in the md.xvg files that cannot be parsed. (default: False)

> **Warning:** Other lines in the file might have been corrupted in such a way that they appear correct but are in fact wrong. WRONG RESULTS CAN OCCUR! USE AT YOUR OWN RISK

It is recommended to simply rerun the affected simulation(s).

5.3.4.3 Output data file formats

Results are *appended* to data files.

Note: Energies are always output in **kJ/mol**.

POW summary file

The `pow.txt` output file summarises the results from the water and octanol solvation calculations. Its name set with the option *mdpow-pow -o*. It contains fixed column data in the following order and all energies are in **kJ/mol**. See the *Table of computed water-octanol transfer energies and logPow* as an example.

itp_name

molecule identifier from the itp file

DeltaG0

transfer free energy from water to octanol (difference between **DeltaG0** for octanol and water), in kJ/mol; >0: partitions into water, <0: partitions into octanol,

errDG0

error on **DeltaG0**; errors are calculated through propagation of errors from the errors on the means <dV/dlambda>

logPOW

log P_{OW}, base-10 logarithm of the octanol-water partition coefficient; >0: partitions into octanol, <0: partitions preferrably into water

errlogP

error on **logPow**

directory

directory under which all data files are stored; by default this is the *name* of the molecule and hence it can be used to identify the compound.

Table 1: Computed log P_{OW} and water-to-octanol transfer energies (in kJ/mol).

itp_name	DeltaG0	errDG0	logPow	errlogP	directory
BNZ	-12.87	0.43	+2.24	0.07	benzene
OC9	-16.24	1.12	+2.83	0.20	octanol
URE	+4.66	1.13	-0.81	0.20	urea
902	+9.35	1.06	-1.63	0.18	water_TIP4P

Energy file

The `energy.txt` output file collects all computed energy terms together with the results also found in the summary file `pow.txt`. Its name is set with the option *mdpow-pow -e*. It contains fixed column data in the following order and all energies are in **kJ/mol**. As an example see *Table of Solvation Energies* for the same compounds as above.

molecule

molecule identifier from the itp file

solvent

solvent name (water, octanol)

DeltaG0

solvation free energy difference in kJ/mol (Ben-Naim standard state, i.e. 1M gas/1M solution)

errDG0

error on **DeltaG0**, calculated through propagation of errors from the errors on the mean <dV/dlambda>

coulomb

Coulomb (discharging) contribution to the solvation free energy **DeltaG0**

errCoul

error on **coulomb**

VDW

Van der Waals/Lennard-Jones (decoupling) contribution to **DeltaG0**

errVDW

error on **VDW**

w2oct

transfer free energy from water to octanol (difference between **DeltaG0** for octanol and water)

errw2oct

error on **w2oct**

logPOW

log P_{OW}

errlogP

error on **logPow**

directory

directory under which all data files are stored; by default this is the *name* of the molecule and hence it can be used to identify the compound.

Table 2: Solvation free energies for compounds in water and octanol (in kJ/mol).

molecule	solvent	DeltaG0	errDG0	coulomb	errCoul	VDW	errVDW	w2oct	errw2oct	logPOW	errlogP	directory
BNZ	water	-2.97	0.21	+7.65	0.07	-4.68	0.20	-12.87	0.43	+2.24	0.07	benzene
BNZ	octanol	-15.84	0.37	+1.40	0.19	+14.44	0.32	-12.87	0.43	+2.24	0.07	benzene
OC9	water	-16.03	0.32	+27.35	0.09	-11.32	0.31	-16.24	1.12	+2.83	0.20	octanol
OC9	octanol	-32.28	1.08	+11.32	0.92	+20.96	0.56	-16.24	1.12	+2.83	0.20	octanol
URE	water	-53.52	0.17	+56.94	0.11	-3.41	0.14	+4.66	1.13	-0.81	0.20	urea
URE	octanol	-48.86	1.12	+35.75	1.09	+13.11	0.25	+4.66	1.13	-0.81	0.20	urea
902	water	-25.46	0.11	+34.93	0.10	-9.48	0.06	+9.35	1.06	-1.63	0.18	water_TIP4P
902	octanol	-16.11	1.05	+21.16	1.05	-5.05	0.09	+9.35	1.06	-1.63	0.18	water_TIP4P

5.3.5 House-keeping scripts

A number of scripts are provided to complete simple tasks; they can be used to check that all required files are present or they can help in fixing small problems without one having to write Python code to do this oneself (e.g. by manipulating the checlpoint files). They generally make the same assumptions about file system layout as the other mdpow scripts.

5.3.5.1 Checking if the simulation is complete

Run **mdpow-check** in order to check if all necessary files are available.

Usage of the command:

> **mdpow-check** [options] *DIRECTORY* [*DIRECTORY* ...]
>
> Check status of the progress of the project in *DIRECTORY*.
>
> Output is only written to the status file (`status.txt`). A quick way to find problems is to do a

```
cat status.txt | gawk -F '|' '$2 !~ /OK/ {print $0}'
```

> Options:
>
> **-h, --help**
> show this help message and exit
>
> **-o** <FILE>, **--outfile**=<FILE>
> write status results to *FILE* [`status.txt`]

5.3.5.2 Changing paths in `water.simulation` and `octanol.simulation`

It can become necessary to recreate the *solvent*`.simulation` status/checkpoint files in order to change paths, e.g. when one moved the directories or transferred all files to a different file system.

Typically, one would execute the **mdpow-rebuild-simulation** command in the parent directory of *moleculename*.

Usage of the command:

> **mdpow-rebuild-simulation** [options] *DIRECTORY* [*DIRECTORY* ...]
>
> Re-create the `water.simulation` or `octanol.simulation` file with adjusted paths (now rooted at *DIRECTORY*).
>
> Options:
>
> **-h, --help**
> show this help message and exit
>
> **--solvent**=<NAME>
> rebuild file for 'water', 'octanol', or 'all' [all]

5.3.5.3 Re-building `Ghyd.fep` and `Goct.fep`

It can become necessary to recreate the *name*`.fep` status/checkpoint files (e.g. if the files became corrupted due to a software error or in order to change paths).

Typically, one would execute the **mdpow-rebuild-fep** command in the parent directory of *moleculename*.

Usage of the command:

mdpow-rebuild-fep [options] *DIRECTORY* [*DIRECTORY* ...]

Re-create the `Goct.fep` or `Ghyd.fep` file using the appropriate equilibrium simulation file under *DIRECTORY*/FEP.

This should only be necessary when the fep file was destroyed due to a software error or when the files are transferred to a different file system and some of the paths stored in the *name*`.fep` files have to be changed.

Options:

-h, --help
show this help message and exit

--solvent=<NAME>
rebuild fep for 'water', 'octanol', or 'all' [all]

--setup=<LIST>
Re-generate queuing system scripts with appropriate paths: runs `fep.Gsolv.setup()` with argument *qscript=[LIST]* after fixing Gsolv.

`LIST` should contain a comma-separated list of queing system templates. For example: `'icsn_8pd.sge,icsn_2pd.sge,local.sh'`. It is an error if `--setup` is set without a `LIST`.

5.4 mdpow.equil — Setting up and running equilibrium MD

The *mdpow.equil* module facilitates the setup of equilibrium molecular dynamics simulations of a compound molecule in a simulation box of water or other solvent such as octanol.

It requires as input

- the itp file for the compound
- a coordinate (structure) file (in pdb or gro format)

By default it uses the *OPLS/AA* forcefield and the *TIP4P* water model.

Warning: Other forcefields than OPLS/AA are currently not officially supported; it is not hard to do but requires tedious changes to a few paths in template scripts.

class mdpow.equil.**Simulation**(*molecule=None*, *forcefield: Union[mdpow.forcefields.Forcefield, str] = 'OPLS-AA'*, ***kwargs*)
Simple MD simulation of a single compound molecule in water.

Typical use

```
S = Simulation(molecule='DRUG')
S.topology(itp='drug.itp')
S.solvate(struct='DRUG-H.pdb')
S.energy_minimize()
S.MD_relaxed()
S.MD()
```

Note: The OPLS/AA force field and the TIP4P water molecule is the default; changing this is possible but will require provision of customized itp, mdp and template top files at various stages.

Set up Simulation instance.

The *molecule* of the compound molecule should be supplied. Existing files (which have been generated in previous runs) can also be supplied.

Keywords

molecule Identifier for the compound molecule. This is the same as the entry in the `[ molecule ]` section of the itp file. ["DRUG"]

filename If provided and *molecule* is `None` then load the instance from the pickle file *filename*, which was generated with *save()*.

dirname base directory; all other directories are created under it

forcefield A `Forcefield`, or the string name of a packaged forcefield: 'OPLS-AA', 'CHARMM' or 'AMBER'.

solvent 'water' or 'octanol' or 'cyclohexane' or 'wetoctanol' or 'toluene'

solventmodel `None` chooses the default (e.g, the *mdpow.forcefields.Forcefield* default water model for `solvent == "water"`. Other options are the models defined in *mdpow.forcefields.GROMACS_WATER_MODELS*. At the moment, there are no alternative parameterizations included for other solvents.

mdp dict with keys corresponding to the stages `energy_minimize`, `MD_restrained`, `MD_relaxed`, `MD_NPT` and values *mdp* file names (if no entry then the package defaults are used)

distance minimum distance between solute and closest box face

kwargs advanced keywords for short-circuiting; see *mdpow.equil.Simulation.filekeys*.

Changed in version 0.9.0: *forcefield* may now be either a `Forcefield` or the string name of a builtin forcefield.

MD (***kwargs*)

Short NPT MD simulation.

See documentation of `gromacs.setup.MD()` for details such as *runtime* or specific queuing system options. The following keywords can not be changed: *top*, *mdp*, *ndx*, *mainselection*.

Note: If the system crashes (with LINCS errors), try initial equilibration with timestep *dt* = 0.0001 ps (0.1 fs instead of 2 fs) and *runtime* = 5 ps as done in *MD_relaxed()*

Keywords

struct starting conformation; by default, the *struct* is the last frame from the position restraints run, or, if this file cannot be found (e.g. because *Simulation.MD_restrained()* was not run) it falls back to the relaxed and then the solvated system.

mdp MDP run parameter file for Gromacs

runtime total run time in ps

qscript list of queuing system scripts to prepare; available values are in `gromacs.config.templates` or you can provide your own filename(s) in the current directory (see `gromacs.qsub` for the format of the templates)

qname name of the job as shown in the queuing system

startdir **advanced uses**: path of the directory on a remote system, which will be hard-coded into the queuing system script(s); see `gromacs.setup.MD()` and `gromacs.manager.Manager`

MD_NPT(***kwargs*)

Short NPT MD simulation.

See documentation of `gromacs.setup.MD()` for details such as *runtime* or specific queuing system options. The following keywords can not be changed: *top*, *mdp*, *ndx*, *mainselection*.

Note: If the system crashes (with LINCS errors), try initial equilibration with timestep *dt* = 0.0001 ps (0.1 fs instead of 2 fs) and *runtime* = 5 ps as done in *MD_relaxed()*

Keywords

struct starting conformation; by default, the *struct* is the last frame from the position restraints run, or, if this file cannot be found (e.g. because *Simulation.MD_restrained()* was not run) it falls back to the relaxed and then the solvated system.

mdp MDP run parameter file for Gromacs

runtime total run time in ps

qscript list of queuing system scripts to prepare; available values are in `gromacs.config.templates` or you can provide your own filename(s) in the current directory (see `gromacs.qsub` for the format of the templates)

qname name of the job as shown in the queuing system

startdir **advanced uses**: path of the directory on a remote system, which will be hard-coded into the queuing system script(s); see `gromacs.setup.MD()` and `gromacs.manager.Manager`

MD_relaxed(***kwargs*)

Short MD simulation with *timestep* = 0.1 fs to relax strain.

Energy minimization does not always remove all problems and LINCS constraint errors occur. A very short *runtime* = 5 ps MD with very short integration time step *dt* tends to solve these problems.

Keywords

struct starting coordinates (typically guessed)

mdp MDP run parameter file for Gromacs

qscript list of queuing system submission scripts; probably a good idea to always include the default "local.sh" even if you have your own ["local.sh"]

qname name of the job as shown in the queuing system

startdir **advanced uses**: path of the directory on a remote system, which will be hard-coded into the queuing system script(s); see `gromacs.setup.MD()` and `gromacs.manager.Manager`

MD_restrained(***kwargs*)

Short MD simulation with position restraints on compound.

See documentation of `gromacs.setup.MD_restrained()` for details. The following keywords can not be changed: top, mdp, ndx, mainselection

Note: Position restraints are activated with `-DPOSRES` directives for `gromacs.grompp()`. Hence this will only work if the compound itp file does indeed contain a `[ posres ]` section that is protected by a `#ifdef POSRES` clause.

Keywords

struct starting coordinates (leave empty for inspired guess of file name)

mdp MDP run parameter file for Gromacs

qscript list of queuing system submission scripts; probably a good idea to always include the default "local.sh" even if you have your own ["local.sh"]

qname name of the job as shown in the queuing system

startdir **advanced uses**: path of the directory on a remote system, which will be hard-coded into the queuing system script(s); see `gromacs.setup.MD()` and `gromacs.manager.Manager`

coordinate_structures = ('solvated', 'energy_minimized', 'MD_relaxed', 'MD_restrained'
Coordinate files of the full system in increasing order of advancement of the protocol; the later the better. The values are keys into `Simulation.files`.

energy_minimize(***kwargs*)
Energy minimize the solvated structure on the local machine.

kwargs are passed to `gromacs.setup.energ_minimize()` but if *solvate()* step has been carried out previously all the defaults should just work.

filekeys = ('topology', 'processed_topology', 'structure', 'solvated', 'ndx', 'energy_
Keyword arguments to pre-set some file names; they are keys in `Simulation.files`.

get_last_checkpoint()
Returns the checkpoint of the most advanced step in the protocol. Relies on md.gro being present from previous simulation, assumes that checkpoint is then present.

get_last_structure()
Returns the coordinates of the most advanced step in the protocol.

load(*filename=None*)
Re-instantiate class from pickled file.

make_paths_relative(*prefix='.'*)
Hack to be able to copy directories around: prune basedir from paths.

Warning: This is not guaranteed to work for all paths. In particular, check `mdpow.equil.Simulation.dirs.includes` and adjust manually if necessary.

mdp_defaults = {'MD_NPT': 'NPT_opls.mdp', 'MD_relaxed': 'NPT_opls.mdp', 'MD_restraine
Default Gromacs *MDP* run parameter files for the different stages. (All are part of the package and are found with *mdpow.config.get_template()*.)

processed_topology(***kwargs*)
Create a portable topology file from the topology and the solvated system.

protocols = ('MD_NPT', 'MD_NPT_run', 'MD_relaxed', 'MD_relaxed_run', 'MD_restrained',
Check list of all methods that can be run as an independent protocol; see also `Simulation.get_protocol()` and `restart.Journal`

save (*filename=None*)
Save instance to a pickle file.

The default filename is the name of the file that was last loaded from or saved to.

solvate (*struct=None*, ***kwargs*)
Solvate structure *struct* in a box of solvent.

The solvent is determined with the *solvent* keyword to the constructor.

Keywords

struct pdb or gro coordinate file (if not supplied, the value is used that was supplied to the constructor of *Simulation*)

distance minimum distance between solute and the closes box face; the default depends on the solvent but can be set explicitly here, too.

bt any box type understood by `gromacs.editconf()` (`-bt`):

- "triclinic" is a triclinic box,
- "cubic" is a rectangular box with all sides equal;
- "dodecahedron" represents a rhombic dodecahedron;
- "octahedron" is a truncated octahedron.

The default is "dodecahedron".

kwargs All other arguments are passed on to `gromacs.setup.solvate()`, but set to sensible default values. *top* and *water* are always fixed.

topology (*itp='drug.itp'*, *prm=None*, ***kwargs*)
Generate a topology for compound *molecule*.

Keywords

itp Gromacs itp file; will be copied to topology dir and included in topology

prm Gromacs prm file; if given, will be copied to topology dir and included in topology

dirname name of the topology directory ["top"]

kwargs see source for *top_template*, *topol*

class mdpow.equil.**WaterSimulation** (*molecule=None*, *forcefield: Union[mdpow.forcefields.Forcefield, str] = 'OPLS-AA'*, ***kwargs*)
Equilibrium MD of a solute in a box of water.

Set up Simulation instance.

The *molecule* of the compound molecule should be supplied. Existing files (which have been generated in previous runs) can also be supplied.

Keywords

molecule Identifier for the compound molecule. This is the same as the entry in the `[ molecule ]` section of the itp file. ["DRUG"]

filename If provided and *molecule* is `None` then load the instance from the pickle file *filename*, which was generated with *save()*.

dirname base directory; all other directories are created under it

forcefield A `Forcefield`, or the string name of a packaged forcefield: 'OPLS-AA', 'CHARMM' or 'AMBER'.

solvent 'water' or 'octanol' or 'cyclohexane' or 'wetoctanol' or 'toluene'

solventmodel `None` chooses the default (e.g, the *mdpow.forcefields.Forcefield* default water model for `solvent == "water"`. Other options are the models defined in *mdpow.forcefields.GROMACS_WATER_MODELS*. At the moment, there are no alternative parameterizations included for other solvents.

mdp dict with keys corresponding to the stages `energy_minimize`, `MD_restrained`, `MD_relaxed`, `MD_NPT` and values *mdp* file names (if no entry then the package defaults are used)

distance minimum distance between solute and closest box face

kwargs advanced keywords for short-circuiting; see *mdpow.equil.Simulation.filekeys*.

Changed in version 0.9.0: *forcefield* may now be either a `Forcefield` or the string name of a builtin forcefield.

class mdpow.equil.**OctanolSimulation**(*molecule=None*, *forcefield: Union[mdpow.forcefields.Forcefield, str] = 'OPLS-AA'*, ***kwargs*)

Equilibrium MD of a solute in a box of octanol.

Set up Simulation instance.

The *molecule* of the compound molecule should be supplied. Existing files (which have been generated in previous runs) can also be supplied.

Keywords

molecule Identifier for the compound molecule. This is the same as the entry in the `[ molecule ]` section of the itp file. ["DRUG"]

filename If provided and *molecule* is `None` then load the instance from the pickle file *filename*, which was generated with *save()*.

dirname base directory; all other directories are created under it

forcefield A `Forcefield`, or the string name of a packaged forcefield: 'OPLS-AA', 'CHARMM' or 'AMBER'.

solvent 'water' or 'octanol' or 'cyclohexane' or 'wetoctanol' or 'toluene'

solventmodel `None` chooses the default (e.g, the *mdpow.forcefields.Forcefield* default water model for `solvent == "water"`. Other options are the models defined in *mdpow.forcefields.GROMACS_WATER_MODELS*. At the moment, there are no alternative parameterizations included for other solvents.

mdp dict with keys corresponding to the stages `energy_minimize`, `MD_restrained`, `MD_relaxed`, `MD_NPT` and values *mdp* file names (if no entry then the package defaults are used)

distance minimum distance between solute and closest box face

kwargs advanced keywords for short-circuiting; see *mdpow.equil.Simulation.filekeys*.

Changed in version 0.9.0: *forcefield* may now be either a `Forcefield` or the string name of a builtin forcefield.

mdpow.equil.**DIST = {'cyclohexane': 1.5, 'octanol': 1.5, 'toluene': 1.5, 'water': 1.0,**

minimum distance between solute and box surface (in nm)

5.5 mdpow.fep – Calculate free energy of solvation

Set up and run free energy perturbation (FEP) calculations to calculate the free energy of hydration of a solute in a solvent box. The protocol follows the works of D. Mobley (Free Energy Tutorial) and M. Shirts, and uses Gromacs 4.0.x.

Required Input:

- topology
- equilibrated structure of the solvated molecule

See the docs for *Gsolv* for details on what is calculated.

5.5.1 Differences to published protocols

Some notes on how the approach encoded here differs from what others (notabley Mobley) did:

- We use Gromacs 4.x and use the new decoupling feature

```
couple-intramol = no
```

  which indicates that "intra-molecular interactions remain". It should (as I understand it) allow us to only calculate the free energy changes in solution so that we do not have to do an extra calculation in vacuo. http://www.mail-archive.com/gmx-users@gromacs.org/msg18803.html

  Mobley does an extra discharging calculation in vacuo and calculates

$$\Delta A = \Delta A_{\mathrm{coul}}(\mathrm{vac}) - (\Delta A_{\mathrm{coul}}(\mathrm{sol}) + \Delta A_{\mathrm{vdw}}(\mathrm{sol}))$$

  (but also annihilates the interactions on the solute, corresponding to `couple-intramol = yes`) whereas we do

$$\Delta A = -(\Delta A_{\mathrm{coul}}(\mathrm{sol}) + \Delta A_{\mathrm{vdw}}(\mathrm{sol}))$$

- simulations are run as NPT (but make sure to use Langevin dynamics for temperature control)

5.5.2 Example

see `mdpow`

5.5.3 User reference

Simulation setup and analysis of all FEP simulations is encapsulated by a *mdpow.fep.Gsolv* object. For the hydration free energy there is a special class *Ghyd* and for the solvation free energy in octanol there is *Goct*. See the description of *Gsolv* for methods common to both.

class mdpow.fep.**Gsolv**(*molecule=None*, *top=None*, *struct=None*, *method='BAR'*, ***kwargs*)

Simulations to calculate and analyze the solvation free energy.

ΔA is computed from the decharging and the decoupling step. With our choice of `lambda=0` being the fully interacting and `lambda=1` the non-interacting state, it is computed as

$$\Delta A = -(\Delta A_{\mathrm{coul}} + \Delta A_{\mathrm{vdw}})$$

With this protocol, the concentration in the liquid and in the gas phase is the same. (Under the assumption of ideal solution/ideal gas behaviour this directly relates to the Ben-Naim 1M/1M standard state.)

(We neglect the negligible correction $-kT \ln V_x/V_{\mathrm{sim}} = -kT \ln(1 - v_s/V_{\mathrm{sim}})$ where V_x is the volume of the system without the solute but the same number of water molecules as in the fully interacting case [see Michael Shirts' Thesis, p82].)

Typical work flow:

```
G = Gsolv(simulation='drug.simulation')           # continue from :mod:`mdpow.
↪equil`
G.setup(qscript=['my_template.sge', 'local.sh'])  # my_template.sge is user␣
↪supplied
G.qsub()    # run SGE job arrays as generated from my_template.sge
G.analyze()
G.plot()
```

See `gromacs.qsub` for notes on how to write templates for queuing system scripts (in particular queuing system templates).

Set up Gsolv from input files or a equilibrium simulation.

Arguments

molecule name of the molecule for which the hydration free energy is to be computed (as in the gromacs topology) [REQUIRED]

top topology [REQUIRED]

struct solvated and equilibrated input structure [REQUIRED]

ndx index file

dirname directory to work under ['FEP/*solvent*']

solvent name of the solvent (only used for path names); will not affect the simulations because the input coordinates and topologies are determined by either *simulation* or *molecule*, *top*, and *struct*. If *solvent* is not provided then it is set to `solvent_default`.

lambda_coulomb list of lambdas for discharging: q+vdw –> vdw

lambda_vdw list of lambdas for decoupling: vdw –> none

runtime simulation time per window in ps [5000]

temperature temperature in Kelvin of the simulation [300.0]

qscript template or list of templates for queuing system scripts (see `gromacs.config.templates` for details) [local.sh]

includes include directories

simulation Instead of providing the required arguments, obtain the input files from a *mdpow.equil.Simulation* instance.

method "TI" for thermodynamic integration or "BAR" for Bennett acceptance ratio; using "BAR" in Gromacs also writes TI data so this is the default.

mdp MDP file name (if no entry then the package defaults are used)

filename Instead of providing the required arguments, load from pickle file. If either *simulation* or *molecule*, *top*, and *struct* are provided then simply set the attribute `filename` of the default checkpoint file to *filename*.

basedir Prepend *basedir* to all filenames; `None` disables [.]

permissive Set to `True` if you want to read past corrupt data in output xvg files (see `gromacs.formats.XVG` for details); note that *permissive*=''True'' can lead to **wrong results*. Overrides the value set in a loaded pickle file. [`False`]

stride collect every *stride* data line, see *Gsolv.collect()* [1]

start Start frame of data analyzed in every fep window.

stop Stop frame of data analyzed in every fep window.

SI Set to `True` if you want to perform statistical inefficiency to preprocess the data.

kwargs other undocumented arguments (see source for the moment)

Note: Only a subset of the FEP parameters can currently be set with keywords (lambdas); other parameters such as the soft cores are currently fixed. For advanced use: pass a dict with keys "coulomb" and "vdw" and values of *FEPschedule* in the keyword *schedules*. The *lambdas* will override these settings (which typically come from a cfg). This method allows setting all parameters.

---

analyze (*force=False*, *stride=None*, *autosave=True*, *ncorrel=25000*)

Extract dV/dl from output and calculate dG by TI.

Thermodynamic integration (TI) is performed on the individual component window calculation (typically the Coulomb and the VDW part, ΔA_{coul} and ΔA_{vdW}). ΔA_{coul} is the free energy component of discharging the molecule and ΔA_{vdW} of decoupling (switching off LJ interactions with the environment). The free energy components must be interpreted in this way because we defined `lambda=0` as interaction switched on and `lambda=1` as switched off.

$$\Delta A^* = -(\Delta A_{\mathrm{coul}} + \Delta A_{\mathrm{vdw}})$$

Data are stored in *Gsolv.results*.

The dV/dlambda graphs are integrated with the composite Simpson's rule (and if the number of datapoints are even, the first interval is evaluated with the trapezoidal rule); see `scipy.integrate.simps()` for details). Note that this implementation of Simpson's rule does not require equidistant spacing on the lambda axis.

For the Coulomb part using Simpson's rule has been shown to produce more accurate results than the trapezoidal rule [Jorge2010].

Errors are estimated from the errors of the individual <dV/dlambda>:

1. The error of the mean <dV/dlambda> is calculated via the decay time of the fluctuation around the mean. `ncorrel` is the max number of samples that is going to be used to calculate the autocorrelation time via a FFT. See `numkit.timeseries.tcorrel()`.
2. The error on the integral is calculated analytically via propagation of errors through Simpson's rule (with the approximation that all spacings are assumed to be equal; taken as the average over all spacings as implemented in `numkit.integration.simps_error()`).

Note: For the Coulomb part, which typically only contains about 5 lambdas, it is recommended to have a odd number of lambda values to fully benefit from the higher accuracy of the integration scheme.

Keywords

force reload raw data even though it is already loaded

stride read data every *stride* lines, `None` uses the class default

autosave save to the pickle file when results have been computed

ncorrel aim for <= 25,000 samples for t_correl

Notes

The error on the mean of the data ϵ_y, taking the correlation time into account, is calculated according to [FrenkelSmit2002] p526:

$$\epsilon_y = \sqrt{2\tau_c \mathrm{acf}(0)/T}$$

where $\mathrm{acf}()$ is the autocorrelation function of the fluctuations around the mean, $y - \langle y \rangle$, τ_c is the correlation time, and T the total length of the simulation

analyze_alchemlyb(*SI=True*, *start=0*, *stop=None*, *stride=None*, *force=False*, *autosave=True*)
Compute the free energy from the simulation data with alchemlyb.

Unlike *analyze()*, the MBAR estimator is available (in addition to TI). Note that SI should be enabled for meaningful error estimates.

arraylabel(*component*)
Batch submission script name for a job array.

collect(*stride=None*, *autosave=True*, *autocompress=True*)
Collect dV/dl from output.

Keywords

stride read data every *stride* lines, `None` uses the class default

autosave immediately save the class pickle fep file

autocompress compress the xvg file with bzip2 (saves >70% of space)

collect_alchemlyb(*SI=True*, *start=0*, *stop=None*, *stride=None*, *autosave=True*, *autocompress=True*)
Collect the data files using alchemlyb.

Unlike *collect()*, this method can subsample with the statistical inefficiency (parameter *SI*).

compress_dgdl_xvg()
Compress *all* dgdl xvg files with bzip2.

Note: After running this method you might want to run *collect()* to ensure that the results in `results.xvg` point to the *compressed* files. Otherwise `IOError` might occur which fail to find a *md.xvg* file.

---

contains_corrupted_xvgs()
Check if any of the source datafiles had reported corrupted lines.

Returns `True` if any of the xvg dgdl files were produced with the permissive=True flag and had skipped lines.

For debugging purposes, the number of corrupted lines are stored in `Gsolv._corrupted` as dicts of dicts with the component as primary and the lambda as secondary key.

convert_edr()
Convert EDR files to compressed XVG files.

dgdl_edr(**args*)
Return filename of the dgdl EDR file.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Returns path to EDR

Raises `IOError` with error code ENOENT if no file could be found

dgdl_total_edr(**args*, ***kwargs*)
Return filename of the combined dgdl EDR file.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Keywords

total_edr_name Name of the user defined total edr file.

Returns path to total EDR

dgdl_tpr(**args*)
Return filename of the dgdl TPR file.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Returns path to TPR

Raises `IOError` with error code ENOENT if no file could be found

dgdl_xvg(**args*)
Return filename of the dgdl XVG file.

Recognizes uncompressed, gzipped (gz), and bzip2ed (bz2) files.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Returns Checks if a compressed file exists and returns the appropriate filename.

Raises `IOError` with error code ENOENT if no file could be found

estimators = {'BAR': {'estimator': <class 'alchemlyb.estimators.bar_.BAR'>, 'extract'
Estimators in alchemlyb

fep_dirs()
Generator for all simulation sub directories

frombase(**args*)
Return path with `Gsolv.basedir` prefixed.

get_protocol(*protocol*)
Return method for *protocol*.

- If *protocol* is a real method of the class then the method is returned.
- If *protocol* is a registered protocol name but no method of the name exists (i.e. *protocol* is a "dummy protocol") then a wrapper function is returned. The wrapper has the signature

  dummy_protocol(*func*, **args*, ***kwargs*)
  Runs *func* with the arguments and keywords between calls to `Journal.start()` and `Journal.completed()`, with the stage set to *protocol*.
- Raises a `ValueError` if the *protocol* is not registered (i.e. not found in `Journalled.protocols`).

has_dVdl()
Check if dV/dl data have already been collected.

Returns `True` if the dV/dl data have bee aquired (*Gsolv.collect()*) for all FEP simulations.

label(*component*)
Simple label for component, e.g. for use in filenames.

load(*filename=None*)
Re-instantiate class from pickled file.

If no *filename* was supplied then the filename is taken from the attribute `filename`.

Changed in version 0.7.1: Can read pickle files with either Python 2.7 or 3.x, regardless of the Python version that created the pickle.

logger_DeltaA0()
Print the free energy contributions (errors in parentheses).

mdp_default = 'bar_opls.mdp'
Default Gromacs *MDP* run parameter file for FEP. (The file is part of the package and is found with *mdpow.config.get_template()*.)

plot(***kwargs*)
Plot the TI data with error bars.

Run *mdpow.fep.Gsolv.analyze()* first.

All *kwargs* are passed on to `pylab.errorbar()`.

Returns The axes of the subplot.

protocols = ['setup', 'fep_run']
Check list of all methods that can be run as an independent protocol; see also `Simulation.get_protocol()` and `restart.Journal`

qsub(*script=None*)
Submit a batch script locally.

If *script* == `None` then take the first script (works well if only one template was provided).

results = None
Results from the analysis

save(*filename=None*)
Save instance to a pickle file.

The default filename is the name of the file that was last loaded from or saved to. Also sets the attribute `filename` to the absolute path of the saved file.

scripts = None
Generated run scripts

setup(***kwargs*)
Prepare the input files for all Gromacs runs.

Keywords

qscript (List of) template(s) for batch submission scripts; if not set then the templates are used that were supplied to the constructor.

kwargs Most kwargs are passed on to `gromacs.setup.MD()` although some are set to values that are required for the FEP functionality.

mdrun_opts list of options to **mdrun**; `-dhdl` is always added to this list as it is required for the thermodynamic integration calculations

includes list of directories where Gromacs input files can be found

The following keywords cannot be changed: *dirname*, *jobname*, *prefix*, *runtime*, *deffnm*. The last two can be specified when constructing *Gsolv*.

See also:

`gromacs.setup.MD()` and `gromacs.qsub.generate_submit_scripts()`

Changed in version 0.6.0: Gromacs now uses option `-dhdl` instead of `-dgdl`.

summary()
Return a string that summarizes the energetics.

Each energy component is followed by its error.

Format:

```
.                  ------ kJ/mol -----
molecule solvent  total  coulomb  vdw
```

tasklabel(*component*, *lmbda*)
Batch submission script name for a single task job.

wdir(*component*, *lmbda*)
Return rooted path to the work directory for *component* and *lmbda*.

(Constructed from *frombase()* and *wname()*.)

wname(*component*, *lmbda*)
Return name of the window directory itself.

Typically something like `VDW/0000`, `VDW/0500`, ..., `Coulomb/1000`

write_DeltaA0(*filename*, *mode='w'*)
Write free energy components to a file.

Arguments

filename name of the text file

mode 'w' for overwrite or 'a' for append ['w']

Format:: . — kJ/mol —— molecule solvent total coulomb vdw

class mdpow.fep.**Ghyd**(*molecule=None*, *top=None*, *struct=None*, *method='BAR'*, ***kwargs*)
Sets up and analyses MD to obtain the hydration free energy of a solute.

Set up Gsolv from input files or a equilibrium simulation.

Arguments

molecule name of the molecule for which the hydration free energy is to be computed (as in the gromacs topology) [REQUIRED]

top topology [REQUIRED]

struct solvated and equilibrated input structure [REQUIRED]

ndx index file

dirname directory to work under ['FEP/*solvent*']

solvent name of the solvent (only used for path names); will not affect the simulations because the input coordinates and topologies are determined by either *simulation* or *molecule*, *top*, and *struct*. If *solvent* is not provided then it is set to `solvent_default`.

lambda_coulomb list of lambdas for discharging: q+vdw –> vdw

lambda_vdw list of lambdas for decoupling: vdw –> none

runtime simulation time per window in ps [5000]

temperature temperature in Kelvin of the simulation [300.0]

qscript template or list of templates for queuing system scripts (see `gromacs.config.templates` for details) [local.sh]

includes include directories

simulation Instead of providing the required arguments, obtain the input files from a *`mdpow.equil.Simulation`* instance.

method "TI" for thermodynamic integration or "BAR" for Bennett acceptance ratio; using "BAR" in Gromacs also writes TI data so this is the default.

mdp MDP file name (if no entry then the package defaults are used)

filename Instead of providing the required arguments, load from pickle file. If either *simulation* or *molecule*, *top*, and *struct* are provided then simply set the attribute `filename` of the default checkpoint file to *filename*.

basedir Prepend *basedir* to all filenames; `None` disables [.]

permissive Set to `True` if you want to read past corrupt data in output xvg files (see `gromacs.formats.XVG` for details); note that *permissive*=*‘‘True‘‘ can lead to **wrong results**.* Overrides the value set in a loaded pickle file. [`False`]

stride collect every *stride* data line, see *`Gsolv.collect()`* [1]

start Start frame of data analyzed in every fep window.

stop Stop frame of data analyzed in every fep window.

SI Set to `True` if you want to perform statistical inefficiency to preprocess the data.

kwargs other undocumented arguments (see source for the moment)

Note: Only a subset of the FEP parameters can currently be set with keywords (lambdas); other parameters such as the soft cores are currently fixed. For advanced use: pass a dict with keys "coulomb" and "vdw" and values of *`FEPschedule`* in the keyword *schedules*. The *lambdas* will override these settings (which typically come from a cfg). This method allows setting all parameters.

`analyze` (*force=False*, *stride=None*, *autosave=True*, *ncorrel=25000*)

Extract dV/dl from output and calculate dG by TI.

Thermodynamic integration (TI) is performed on the individual component window calculation (typically the Coulomb and the VDW part, ΔA_{coul} and ΔA_{vdW}). ΔA_{coul} is the free energy component of discharging the molecule and ΔA_{vdW} of decoupling (switching off LJ interactions with the environment). The free energy components must be interpreted in this way because we defined `lambda=0` as interaction switched on and `lambda=1` as switched off.

$$\Delta A^* = -(\Delta A_{\mathrm{coul}} + \Delta A_{\mathrm{vdw}})$$

Data are stored in *`Gsolv.results`*.

The dV/dlambda graphs are integrated with the composite Simpson's rule (and if the number of datapoints are even, the first interval is evaluated with the trapezoidal rule); see `scipy.integrate.simps()` for details). Note that this implementation of Simpson's rule does not require equidistant spacing on the lambda axis.

For the Coulomb part using Simpson's rule has been shown to produce more accurate results than the trapezoidal rule [Jorge2010].

Errors are estimated from the errors of the individual <dV/dlambda>:

1. The error of the mean <dV/dlambda> is calculated via the decay time of the fluctuation around the mean. `ncorrel` is the max number of samples that is going to be used to calculate the autocorrelation time via a FFT. See `numkit.timeseries.tcorrel()`.
2. The error on the integral is calculated analytically via propagation of errors through Simpson's rule (with the approximation that all spacings are assumed to be equal; taken as the average over all spacings as implemented in `numkit.integration.simps_error()`).

Note: For the Coulomb part, which typically only contains about 5 lambdas, it is recommended to have a odd number of lambda values to fully benefit from the higher accuracy of the integration scheme.

Keywords

force reload raw data even though it is already loaded

stride read data every *stride* lines, `None` uses the class default

autosave save to the pickle file when results have been computed

ncorrel aim for <= 25,000 samples for t_correl

Notes

The error on the mean of the data ϵ_y, taking the correlation time into account, is calculated according to [FrenkelSmit2002] p526:

$$\epsilon_y = \sqrt{2\tau_c \mathrm{acf}(0)/T}$$

where $\mathrm{acf}()$ is the autocorrelation function of the fluctuations around the mean, $y - \langle y \rangle$, τ_c is the correlation time, and T the total length of the simulation

analyze_alchemlyb(*SI=True*, *start=0*, *stop=None*, *stride=None*, *force=False*, *autosave=True*)
Compute the free energy from the simulation data with alchemlyb.

Unlike *`analyze()`*, the MBAR estimator is available (in addition to TI). Note that SI should be enabled for meaningful error estimates.

arraylabel(*component*)
Batch submission script name for a job array.

collect(*stride=None*, *autosave=True*, *autocompress=True*)
Collect dV/dl from output.

Keywords

stride read data every *stride* lines, `None` uses the class default

autosave immediately save the class pickle fep file

autocompress compress the xvg file with bzip2 (saves >70% of space)

collect_alchemlyb(*SI=True*, *start=0*, *stop=None*, *stride=None*, *autosave=True*, *autocompress=True*)
Collect the data files using alchemlyb.

Unlike *collect()*, this method can subsample with the statistical inefficiency (parameter *SI*).

compress_dgdl_xvg()
Compress *all* dgdl xvg files with bzip2.

Note: After running this method you might want to run *collect()* to ensure that the results in `results.xvg` point to the *compressed* files. Otherwise `IOError` might occur which fail to find a *md.xvg* file.

---

contains_corrupted_xvgs()
Check if any of the source datafiles had reported corrupted lines.

Returns `True` if any of the xvg dgdl files were produced with the permissive=True flag and had skipped lines.

For debugging purposes, the number of corrupted lines are stored in `Gsolv._corrupted` as dicts of dicts with the component as primary and the lambda as secondary key.

convert_edr()
Convert EDR files to compressed XVG files.

dgdl_edr(**args*)
Return filename of the dgdl EDR file.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Returns path to EDR

Raises `IOError` with error code ENOENT if no file could be found

dgdl_total_edr(**args*, ***kwargs*)
Return filename of the combined dgdl EDR file.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Keywords

total_edr_name Name of the user defined total edr file.

Returns path to total EDR

dgdl_tpr(**args*)
Return filename of the dgdl TPR file.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Returns path to TPR

Raises `IOError` with error code ENOENT if no file could be found

dgdl_xvg(**args*)
Return filename of the dgdl XVG file.

Recognizes uncompressed, gzipped (gz), and bzip2ed (bz2) files.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Returns Checks if a compressed file exists and returns the appropriate filename.

Raises `IOError` with error code ENOENT if no file could be found

fep_dirs()
Generator for all simulation sub directories

frombase(**args*)
Return path with `Gsolv.basedir` prefixed.

get_protocol(*protocol*)
Return method for *protocol*.

- If *protocol* is a real method of the class then the method is returned.
- If *protocol* is a registered protocol name but no method of the name exists (i.e. *protocol* is a "dummy protocol") then a wrapper function is returned. The wrapper has the signature

  dummy_protocol(*func*, **args*, ***kwargs*)
  Runs *func* with the arguments and keywords between calls to `Journal.start()` and `Journal.completed()`, with the stage set to *protocol*.
- Raises a `ValueError` if the *protocol* is not registered (i.e. not found in `Journalled.protocols`).

has_dVdl()
Check if dV/dl data have already been collected.

Returns `True` if the dV/dl data have bee aquired (*`Gsolv.collect()`*) for all FEP simulations.

label(*component*)
Simple label for component, e.g. for use in filenames.

load(*filename=None*)
Re-instantiate class from pickled file.

If no *filename* was supplied then the filename is taken from the attribute `filename`.

Changed in version 0.7.1: Can read pickle files with either Python 2.7 or 3.x, regardless of the Python version that created the pickle.

logger_DeltaA0()
Print the free energy contributions (errors in parentheses).

plot(***kwargs*)
Plot the TI data with error bars.

Run *`mdpow.fep.Gsolv.analyze()`* first.

All *kwargs* are passed on to `pylab.errorbar()`.

Returns The axes of the subplot.

qsub(*script=None*)
Submit a batch script locally.

If *script* == `None` then take the first script (works well if only one template was provided).

save(*filename=None*)
Save instance to a pickle file.

The default filename is the name of the file that was last loaded from or saved to. Also sets the attribute `filename` to the absolute path of the saved file.

setup(***kwargs*)
Prepare the input files for all Gromacs runs.

Keywords

qscript (List of) template(s) for batch submission scripts; if not set then the templates are used that were supplied to the constructor.

kwargs Most kwargs are passed on to `gromacs.setup.MD()` although some are set to values that are required for the FEP functionality.

mdrun_opts list of options to **mdrun**; `-dhdl` is always added to this list as it is required for the thermodynamic integration calculations

includes list of directories where Gromacs input files can be found

The following keywords cannot be changed: *dirname*, *jobname*, *prefix*, *runtime*, *deffnm*. The last two can be specified when constructing *Gsolv*.

See also:

`gromacs.setup.MD()` and `gromacs.qsub.generate_submit_scripts()`

Changed in version 0.6.0: Gromacs now uses option `-dhdl` instead of `-dgdl`.

summary()
Return a string that summarizes the energetics.

Each energy component is followed by its error.

Format:

```
.                  ------ kJ/mol -----
molecule solvent  total  coulomb  vdw
```

tasklabel(*component*, *lmbda*)
Batch submission script name for a single task job.

wdir(*component*, *lmbda*)
Return rooted path to the work directory for *component* and *lmbda*.

(Constructed from *frombase()* and *wname()*.)

wname(*component*, *lmbda*)
Return name of the window directory itself.

Typically something like `VDW/0000`, `VDW/0500`, ..., `Coulomb/1000`

write_DeltaA0(*filename*, *mode='w'*)
Write free energy components to a file.

Arguments

filename name of the text file

mode 'w' for overwrite or 'a' for append ['w']

Format:: . —— kJ/mol —— molecule solvent total coulomb vdw

class mdpow.fep.**Goct**(*molecule=None*, *top=None*, *struct=None*, *method='BAR'*, ***kwargs*)
Sets up and analyses MD to obtain the solvation free energy of a solute in octanol.

The *coulomb* lambda schedule is enhanced compared to water as the initial part of the dV/dl curve is quite sensitive. By adding two additional points we hope to reduce the overall error on the dis-charging free energy.

Set up Gsolv from input files or a equilibrium simulation.

Arguments

molecule name of the molecule for which the hydration free energy is to be computed (as in the gromacs topology) [REQUIRED]

top topology [REQUIRED]

struct solvated and equilibrated input structure [REQUIRED]

ndx index file

dirname directory to work under ['FEP/*solvent*']

solvent name of the solvent (only used for path names); will not affect the simulations because the input coordinates and topologies are determined by either *simulation* or *molecule*, *top*, and *struct*. If *solvent* is not provided then it is set to `solvent_default`.

lambda_coulomb list of lambdas for discharging: q+vdw –> vdw

lambda_vdw list of lambdas for decoupling: vdw –> none

runtime simulation time per window in ps [5000]

temperature temperature in Kelvin of the simulation [300.0]

qscript template or list of templates for queuing system scripts (see `gromacs.config.templates` for details) [local.sh]

includes include directories

simulation Instead of providing the required arguments, obtain the input files from a *`mdpow.equil.Simulation`* instance.

method "TI" for thermodynamic integration or "BAR" for Bennett acceptance ratio; using "BAR" in Gromacs also writes TI data so this is the default.

mdp MDP file name (if no entry then the package defaults are used)

filename Instead of providing the required arguments, load from pickle file. If either *simulation* or *molecule*, *top*, and *struct* are provided then simply set the attribute `filename` of the default checkpoint file to *filename*.

basedir Prepend *basedir* to all filenames; `None` disables [.]

permissive Set to `True` if you want to read past corrupt data in output xvg files (see `gromacs.formats.XVG` for details); note that *permissive*=''True'' can lead to **wrong results*. Overrides the value set in a loaded pickle file. [`False`]

stride collect every *stride* data line, see *`Gsolv.collect()`* [1]

start Start frame of data analyzed in every fep window.

stop Stop frame of data analyzed in every fep window.

SI Set to `True` if you want to perform statistical inefficiency to preprocess the data.

kwargs other undocumented arguments (see source for the moment)

Note: Only a subset of the FEP parameters can currently be set with keywords (lambdas); other parameters such as the soft cores are currently fixed. For advanced use: pass a dict with keys "coulomb" and "vdw" and

values of `FEPschedule` in the keyword *schedules*. The *lambdas* will override these settings (which typically come from a cfg). This method allows setting all parameters.

---

analyze (*force=False*, *stride=None*, *autosave=True*, *ncorrel=25000*)

Extract dV/dl from output and calculate dG by TI.

Thermodynamic integration (TI) is performed on the individual component window calculation (typically the Coulomb and the VDW part, ΔA_{coul} and ΔA_{vdW}). ΔA_{coul} is the free energy component of discharging the molecule and ΔA_{vdW} of decoupling (switching off LJ interactions with the environment). The free energy components must be interpreted in this way because we defined `lambda=0` as interaction switched on and `lambda=1` as switched off.

$$\Delta A^* = -(\Delta A_{\mathrm{coul}} + \Delta A_{\mathrm{vdw}})$$

Data are stored in `Gsolv.results`.

The dV/dlambda graphs are integrated with the composite Simpson's rule (and if the number of datapoints are even, the first interval is evaluated with the trapezoidal rule); see `scipy.integrate.simps()` for details). Note that this implementation of Simpson's rule does not require equidistant spacing on the lambda axis.

For the Coulomb part using Simpson's rule has been shown to produce more accurate results than the trapezoidal rule [Jorge2010].

Errors are estimated from the errors of the individual <dV/dlambda>:

1. The error of the mean <dV/dlambda> is calculated via the decay time of the fluctuation around the mean. `ncorrel` is the max number of samples that is going to be used to calculate the autocorrelation time via a FFT. See `numkit.timeseries.tcorrel()`.
2. The error on the integral is calculated analytically via propagation of errors through Simpson's rule (with the approximation that all spacings are assumed to be equal; taken as the average over all spacings as implemented in `numkit.integration.simps_error()`).

Note: For the Coulomb part, which typically only contains about 5 lambdas, it is recommended to have a odd number of lambda values to fully benefit from the higher accuracy of the integration scheme.

Keywords

force reload raw data even though it is already loaded

stride read data every *stride* lines, `None` uses the class default

autosave save to the pickle file when results have been computed

ncorrel aim for <= 25,000 samples for t_correl

Notes

The error on the mean of the data ϵ_y, taking the correlation time into account, is calculated according to [FrenkelSmit2002] p526:

$$\epsilon_y = \sqrt{2\tau_c \mathrm{acf}(0)/T}$$

where $\mathrm{acf}()$ is the autocorrelation function of the fluctuations around the mean, $y - \langle y \rangle$, τ_c is the correlation time, and T the total length of the simulation

analyze_alchemlyb(*SI=True*, *start=0*, *stop=None*, *stride=None*, *force=False*, *autosave=True*)
Compute the free energy from the simulation data with alchemlyb.

Unlike *analyze()*, the MBAR estimator is available (in addition to TI). Note that SI should be enabled for meaningful error estimates.

arraylabel(*component*)
Batch submission script name for a job array.

collect(*stride=None*, *autosave=True*, *autocompress=True*)
Collect dV/dl from output.

Keywords

stride read data every *stride* lines, `None` uses the class default

autosave immediately save the class pickle fep file

autocompress compress the xvg file with bzip2 (saves >70% of space)

collect_alchemlyb(*SI=True*, *start=0*, *stop=None*, *stride=None*, *autosave=True*, *autocompress=True*)
Collect the data files using alchemlyb.

Unlike *collect()*, this method can subsample with the statistical inefficiency (parameter *SI*).

compress_dgdl_xvg()
Compress *all* dgdl xvg files with bzip2.

Note: After running this method you might want to run *collect()* to ensure that the results in `results.xvg` point to the *compressed* files. Otherwise `IOError` might occur which fail to find a *md.xvg* file.

---

contains_corrupted_xvgs()
Check if any of the source datafiles had reported corrupted lines.

Returns `True` if any of the xvg dgdl files were produced with the permissive=True flag and had skipped lines.

For debugging purposes, the number of corrupted lines are stored in `Gsolv._corrupted` as dicts of dicts with the component as primary and the lambda as secondary key.

convert_edr()
Convert EDR files to compressed XVG files.

dgdl_edr(**args*)
Return filename of the dgdl EDR file.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Returns path to EDR

Raises `IOError` with error code ENOENT if no file could be found

dgdl_total_edr(**args*, ***kwargs*)
Return filename of the combined dgdl EDR file.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Keywords

total_edr_name Name of the user defined total edr file.

Returns path to total EDR

dgdl_tpr(**args*)
Return filename of the dgdl TPR file.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Returns path to TPR

Raises `IOError` with error code ENOENT if no file could be found

dgdl_xvg(**args*)
Return filename of the dgdl XVG file.

Recognizes uncompressed, gzipped (gz), and bzip2ed (bz2) files.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Returns Checks if a compressed file exists and returns the appropriate filename.

Raises `IOError` with error code ENOENT if no file could be found

fep_dirs()
Generator for all simulation sub directories

frombase(**args*)
Return path with `Gsolv.basedir` prefixed.

get_protocol(*protocol*)
Return method for *protocol*.

- If *protocol* is a real method of the class then the method is returned.
- If *protocol* is a registered protocol name but no method of the name exists (i.e. *protocol* is a "dummy protocol") then a wrapper function is returned. The wrapper has the signature

  dummy_protocol(*func*, **args*, ***kwargs*)
  Runs *func* with the arguments and keywords between calls to `Journal.start()` and `Journal.completed()`, with the stage set to *protocol*.
- Raises a `ValueError` if the *protocol* is not registered (i.e. not found in `Journalled.protocols`).

has_dVdl()
Check if dV/dl data have already been collected.

Returns `True` if the dV/dl data have bee aquired (*`Gsolv.collect()`*) for all FEP simulations.

label(*component*)
Simple label for component, e.g. for use in filenames.

load(*filename=None*)
Re-instantiate class from pickled file.

If no *filename* was supplied then the filename is taken from the attribute `filename`.

Changed in version 0.7.1: Can read pickle files with either Python 2.7 or 3.x, regardless of the Python version that created the pickle.

logger_DeltaA0()
Print the free energy contributions (errors in parentheses).

plot(***kwargs*)
Plot the TI data with error bars.

Run *mdpow.fep.Gsolv.analyze()* first.

All *kwargs* are passed on to `pylab.errorbar()`.

Returns The axes of the subplot.

qsub(*script=None*)
Submit a batch script locally.

If *script* == `None` then take the first script (works well if only one template was provided).

save(*filename=None*)
Save instance to a pickle file.

The default filename is the name of the file that was last loaded from or saved to. Also sets the attribute `filename` to the absolute path of the saved file.

setup(***kwargs*)
Prepare the input files for all Gromacs runs.

Keywords

qscript (List of) template(s) for batch submission scripts; if not set then the templates are used that were supplied to the constructor.

kwargs Most kwargs are passed on to `gromacs.setup.MD()` although some are set to values that are required for the FEP functionality.

mdrun_opts list of options to **mdrun**; `-dhdl` is always added to this list as it is required for the thermodynamic integration calculations

includes list of directories where Gromacs input files can be found

The following keywords cannot be changed: *dirname*, *jobname*, *prefix*, *runtime*, *deffnm*. The last two can be specified when constructing *Gsolv*.

See also:

`gromacs.setup.MD()` and `gromacs.qsub.generate_submit_scripts()`

Changed in version 0.6.0: Gromacs now uses option `-dhdl` instead of `-dgdl`.

summary()
Return a string that summarizes the energetics.

Each energy component is followed by its error.

Format:

```
.                  ------ kJ/mol -----
molecule solvent  total  coulomb  vdw
```

tasklabel(*component*, *lmbda*)
Batch submission script name for a single task job.

wdir(*component*, *lmbda*)
Return rooted path to the work directory for *component* and *lmbda*.

(Constructed from *frombase()* and *wname()*.)

wname (*component*, *lmbda*)
Return name of the window directory itself.

Typically something like `VDW/0000`, `VDW/0500`, ..., `Coulomb/1000`

write_DeltaA0 (*filename*, *mode='w'*)
Write free energy components to a file.

Arguments

filename name of the text file

mode 'w' for overwrite or 'a' for append ['w']

Format:: . —— kJ/mol —— molecule solvent total coulomb vdw

class mdpow.fep.**Gcyclo** (*molecule=None*, *top=None*, *struct=None*, *method='BAR'*, ***kwargs*)
Set up Gsolv from input files or a equilibrium simulation.

Arguments

molecule name of the molecule for which the hydration free energy is to be computed (as in the gromacs topology) [REQUIRED]

top topology [REQUIRED]

struct solvated and equilibrated input structure [REQUIRED]

ndx index file

dirname directory to work under ['FEP/*solvent*']

solvent name of the solvent (only used for path names); will not affect the simulations because the input coordinates and topologies are determined by either *simulation* or *molecule*, *top*, and *struct*. If *solvent* is not provided then it is set to `solvent_default`.

lambda_coulomb list of lambdas for discharging: q+vdw –> vdw

lambda_vdw list of lambdas for decoupling: vdw –> none

runtime simulation time per window in ps [5000]

temperature temperature in Kelvin of the simulation [300.0]

qscript template or list of templates for queuing system scripts (see `gromacs.config.templates` for details) [local.sh]

includes include directories

simulation Instead of providing the required arguments, obtain the input files from a *mdpow.equil.Simulation* instance.

method "TI" for thermodynamic integration or "BAR" for Bennett acceptance ratio; using "BAR" in Gromacs also writes TI data so this is the default.

mdp MDP file name (if no entry then the package defaults are used)

filename Instead of providing the required arguments, load from pickle file. If either *simulation* or *molecule*, *top*, and *struct* are provided then simply set the attribute `filename` of the default checkpoint file to *filename*.

basedir Prepend *basedir* to all filenames; `None` disables [.]

permissive Set to `True` if you want to read past corrupt data in output xvg files (see `gromacs.formats.XVG` for details); note that *permissive*=''True'' can lead to **wrong results*. Overrides the value set in a loaded pickle file. [`False`]

stride collect every *stride* data line, see `Gsolv.collect()` [1]

start Start frame of data analyzed in every fep window.

stop Stop frame of data analyzed in every fep window.

SI Set to `True` if you want to perform statistical inefficiency to preprocess the data.

kwargs other undocumented arguments (see source for the moment)

Note: Only a subset of the FEP parameters can currently be set with keywords (lambdas); other parameters such as the soft cores are currently fixed. For advanced use: pass a dict with keys "coulomb" and "vdw" and values of `FEPschedule` in the keyword *schedules*. The *lambdas* will override these settings (which typically come from a cfg). This method allows setting all parameters.

---

analyze (*force=False*, *stride=None*, *autosave=True*, *ncorrel=25000*)

Extract dV/dl from output and calculate dG by TI.

Thermodynamic integration (TI) is performed on the individual component window calculation (typically the Coulomb and the VDW part, ΔA_{coul} and ΔA_{vdW}). ΔA_{coul} is the free energy component of discharging the molecule and ΔA_{vdW} of decoupling (switching off LJ interactions with the environment). The free energy components must be interpreted in this way because we defined `lambda=0` as interaction switched on and `lambda=1` as switched off.

$$\Delta A^* = -(\Delta A_{\mathrm{coul}} + \Delta A_{\mathrm{vdw}})$$

Data are stored in `Gsolv.results`.

The dV/dlambda graphs are integrated with the composite Simpson's rule (and if the number of datapoints are even, the first interval is evaluated with the trapezoidal rule); see `scipy.integrate.simps()` for details). Note that this implementation of Simpson's rule does not require equidistant spacing on the lambda axis.

For the Coulomb part using Simpson's rule has been shown to produce more accurate results than the trapezoidal rule [Jorge2010].

Errors are estimated from the errors of the individual <dV/dlambda>:

1. The error of the mean <dV/dlambda> is calculated via the decay time of the fluctuation around the mean. `ncorrel` is the max number of samples that is going to be used to calculate the autocorrelation time via a FFT. See `numkit.timeseries.tcorrel()`.
2. The error on the integral is calculated analytically via propagation of errors through Simpson's rule (with the approximation that all spacings are assumed to be equal; taken as the average over all spacings as implemented in `numkit.integration.simps_error()`).

Note: For the Coulomb part, which typically only contains about 5 lambdas, it is recommended to have a odd number of lambda values to fully benefit from the higher accuracy of the integration scheme.

Keywords

force reload raw data even though it is already loaded

stride read data every *stride* lines, `None` uses the class default

autosave save to the pickle file when results have been computed

ncorrel aim for <= 25,000 samples for t_correl

Notes

The error on the mean of the data ϵ_y, taking the correlation time into account, is calculated according to [FrenkelSmit2002] p526:

$$\epsilon_y = \sqrt{2\tau_c \mathrm{acf}(0)/T}$$

where $\mathrm{acf}()$ is the autocorrelation function of the fluctuations around the mean, $y - \langle y \rangle$, τ_c is the correlation time, and T the total length of the simulation

analyze_alchemlyb(*SI=True*, *start=0*, *stop=None*, *stride=None*, *force=False*, *autosave=True*)
Compute the free energy from the simulation data with alchemlyb.

Unlike *analyze()*, the MBAR estimator is available (in addition to TI). Note that SI should be enabled for meaningful error estimates.

arraylabel(*component*)
Batch submission script name for a job array.

collect(*stride=None*, *autosave=True*, *autocompress=True*)
Collect dV/dl from output.

Keywords

stride read data every *stride* lines, `None` uses the class default

autosave immediately save the class pickle fep file

autocompress compress the xvg file with bzip2 (saves >70% of space)

collect_alchemlyb(*SI=True*, *start=0*, *stop=None*, *stride=None*, *autosave=True*, *autocompress=True*)
Collect the data files using alchemlyb.

Unlike *collect()*, this method can subsample with the statistical inefficiency (parameter *SI*).

compress_dgdl_xvg()
Compress *all* dgdl xvg files with bzip2.

Note: After running this method you might want to run *collect()* to ensure that the results in `results.xvg` point to the *compressed* files. Otherwise `IOError` might occur which fail to find a *md.xvg* file.

---

contains_corrupted_xvgs()
Check if any of the source datafiles had reported corrupted lines.

Returns `True` if any of the xvg dgdl files were produced with the permissive=True flag and had skipped lines.

For debugging purposes, the number of corrupted lines are stored in `Gsolv._corrupted` as dicts of dicts with the component as primary and the lambda as secondary key.

convert_edr()
Convert EDR files to compressed XVG files.

dgdl_edr(**args*)
Return filename of the dgdl EDR file.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Returns path to EDR

Raises `IOError` with error code ENOENT if no file could be found

dgdl_total_edr(**args*, ***kwargs*)
Return filename of the combined dgdl EDR file.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Keywords

total_edr_name Name of the user defined total edr file.

Returns path to total EDR

dgdl_tpr(**args*)
Return filename of the dgdl TPR file.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Returns path to TPR

Raises `IOError` with error code ENOENT if no file could be found

dgdl_xvg(**args*)
Return filename of the dgdl XVG file.

Recognizes uncompressed, gzipped (gz), and bzip2ed (bz2) files.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Returns Checks if a compressed file exists and returns the appropriate filename.

Raises `IOError` with error code ENOENT if no file could be found

fep_dirs()
Generator for all simulation sub directories

frombase(**args*)
Return path with `Gsolv.basedir` prefixed.

get_protocol(*protocol*)
Return method for *protocol*.

- If *protocol* is a real method of the class then the method is returned.
- If *protocol* is a registered protocol name but no method of the name exists (i.e. *protocol* is a "dummy protocol") then a wrapper function is returned. The wrapper has the signature

  dummy_protocol(*func*, **args*, ***kwargs*)
  Runs *func* with the arguments and keywords between calls to `Journal.start()` and `Journal.completed()`, with the stage set to *protocol*.
- Raises a `ValueError` if the *protocol* is not registered (i.e. not found in `Journalled.protocols`).

has_dVdl()
Check if dV/dl data have already been collected.

Returns `True` if the dV/dl data have bee aquired (*`Gsolv.collect()`*) for all FEP simulations.

label(*component*)
Simple label for component, e.g. for use in filenames.

load(*filename=None*)
Re-instantiate class from pickled file.

If no *filename* was supplied then the filename is taken from the attribute `filename`.

Changed in version 0.7.1: Can read pickle files with either Python 2.7 or 3.x, regardless of the Python version that created the pickle.

logger_DeltaA0()
Print the free energy contributions (errors in parentheses).

plot(***kwargs*)
Plot the TI data with error bars.

Run *mdpow.fep.Gsolv.analyze()* first.

All *kwargs* are passed on to `pylab.errorbar()`.

Returns The axes of the subplot.

qsub(*script=None*)
Submit a batch script locally.

If *script* == `None` then take the first script (works well if only one template was provided).

save(*filename=None*)
Save instance to a pickle file.

The default filename is the name of the file that was last loaded from or saved to. Also sets the attribute `filename` to the absolute path of the saved file.

setup(***kwargs*)
Prepare the input files for all Gromacs runs.

Keywords

qscript (List of) template(s) for batch submission scripts; if not set then the templates are used that were supplied to the constructor.

kwargs Most kwargs are passed on to `gromacs.setup.MD()` although some are set to values that are required for the FEP functionality.

mdrun_opts list of options to **mdrun**; `-dhdl` is always added to this list as it is required for the thermodynamic integration calculations

includes list of directories where Gromacs input files can be found

The following keywords cannot be changed: *dirname*, *jobname*, *prefix*, *runtime*, *deffnm*. The last two can be specified when constructing *Gsolv*.

See also:

`gromacs.setup.MD()` and `gromacs.qsub.generate_submit_scripts()`

Changed in version 0.6.0: Gromacs now uses option `-dhdl` instead of `-dgdl`.

summary()
Return a string that summarizes the energetics.

Each energy component is followed by its error.

Format:

```
.                  ------ kJ/mol -----
molecule solvent  total  coulomb  vdw
```

tasklabel (*component*, *lmbda*)
Batch submission script name for a single task job.

wdir (*component*, *lmbda*)
Return rooted path to the work directory for *component* and *lmbda*.

(Constructed from *frombase()* and *wname()*.)

wname (*component*, *lmbda*)
Return name of the window directory itself.

Typically something like `VDW/0000`, `VDW/0500`, ..., `Coulomb/1000`

write_DeltaA0 (*filename*, *mode='w'*)
Write free energy components to a file.

Arguments

filename name of the text file

mode 'w' for overwrite or 'a' for append ['w']

Format:: . —— kJ/mol —— molecule solvent total coulomb vdw

class mdpow.fep.**Gtol** (*molecule=None*, *top=None*, *struct=None*, *method='BAR'*, ***kwargs*)
Sets up and analyses MD to obtain the solvation free energy of a solute in toluene.

Set up Gsolv from input files or a equilibrium simulation.

Arguments

molecule name of the molecule for which the hydration free energy is to be computed (as in the gromacs topology) [REQUIRED]

top topology [REQUIRED]

struct solvated and equilibrated input structure [REQUIRED]

ndx index file

dirname directory to work under ['FEP/*solvent*']

solvent name of the solvent (only used for path names); will not affect the simulations because the input coordinates and topologies are determined by either *simulation* or *molecule*, *top*, and *struct*. If *solvent* is not provided then it is set to `solvent_default`.

lambda_coulomb list of lambdas for discharging: q+vdw –> vdw

lambda_vdw list of lambdas for decoupling: vdw –> none

runtime simulation time per window in ps [5000]

temperature temperature in Kelvin of the simulation [300.0]

qscript template or list of templates for queuing system scripts (see *gromacs.config.templates* for details) [local.sh]

includes include directories

simulation Instead of providing the required arguments, obtain the input files from a *mdpow.equil.Simulation* instance.

method "TI" for thermodynamic integration or "BAR" for Bennett acceptance ratio; using "BAR" in Gromacs also writes TI data so this is the default.

mdp MDP file name (if no entry then the package defaults are used)

filename Instead of providing the required arguments, load from pickle file. If either *simulation* or *molecule*, *top*, and *struct* are provided then simply set the attribute `filename` of the default checkpoint file to *filename*.

basedir Prepend *basedir* to all filenames; `None` disables [.]

permissive Set to `True` if you want to read past corrupt data in output xvg files (see `gromacs.formats.XVG` for details); note that *permissive*=*‘‘True‘‘ can lead to **wrong results**.* Overrides the value set in a loaded pickle file. [`False`]

stride collect every *stride* data line, see *Gsolv.collect()* [1]

start Start frame of data analyzed in every fep window.

stop Stop frame of data analyzed in every fep window.

SI Set to `True` if you want to perform statistical inefficiency to preprocess the data.

kwargs other undocumented arguments (see source for the moment)

Note: Only a subset of the FEP parameters can currently be set with keywords (lambdas); other parameters such as the soft cores are currently fixed. For advanced use: pass a dict with keys "coulomb" and "vdw" and values of *FEPschedule* in the keyword *schedules*. The *lambdas* will override these settings (which typically come from a cfg). This method allows setting all parameters.

---

analyze (*force=False*, *stride=None*, *autosave=True*, *ncorrel=25000*)

Extract dV/dl from output and calculate dG by TI.

Thermodynamic integration (TI) is performed on the individual component window calculation (typically the Coulomb and the VDW part, ΔA_{coul} and ΔA_{vdW}). ΔA_{coul} is the free energy component of discharging the molecule and ΔA_{vdW} of decoupling (switching off LJ interactions with the environment). The free energy components must be interpreted in this way because we defined `lambda=0` as interaction switched on and `lambda=1` as switched off.

$$\Delta A^* = -(\Delta A_{\mathrm{coul}} + \Delta A_{\mathrm{vdw}})$$

Data are stored in *Gsolv.results*.

The dV/dlambda graphs are integrated with the composite Simpson's rule (and if the number of datapoints are even, the first interval is evaluated with the trapezoidal rule); see `scipy.integrate.simps()` for details). Note that this implementation of Simpson's rule does not require equidistant spacing on the lambda axis.

For the Coulomb part using Simpson's rule has been shown to produce more accurate results than the trapezoidal rule [Jorge2010].

Errors are estimated from the errors of the individual <dV/dlambda>:

1. The error of the mean <dV/dlambda> is calculated via the decay time of the fluctuation around the mean. `ncorrel` is the max number of samples that is going to be used to calculate the autocorrelation time via a FFT. See `numkit.timeseries.tcorrel()`.
2. The error on the integral is calculated analytically via propagation of errors through Simpson's rule (with the approximation that all spacings are assumed to be equal; taken as the average over all spacings as implemented in `numkit.integration.simps_error()`).

Note: For the Coulomb part, which typically only contains about 5 lambdas, it is recommended to have a odd number of lambda values to fully benefit from the higher accuracy of the integration scheme.

Keywords

force reload raw data even though it is already loaded

stride read data every *stride* lines, `None` uses the class default

autosave save to the pickle file when results have been computed

ncorrel aim for <= 25,000 samples for t_correl

Notes

The error on the mean of the data ϵ_y, taking the correlation time into account, is calculated according to [FrenkelSmit2002] p526:

$$\epsilon_y = \sqrt{2\tau_c \mathrm{acf}(0)/T}$$

where $\mathrm{acf}()$ is the autocorrelation function of the fluctuations around the mean, $y - \langle y \rangle$, τ_c is the correlation time, and T the total length of the simulation

analyze_alchemlyb(*SI=True*, *start=0*, *stop=None*, *stride=None*, *force=False*, *autosave=True*)
Compute the free energy from the simulation data with alchemlyb.

Unlike *analyze()*, the MBAR estimator is available (in addition to TI). Note that SI should be enabled for meaningful error estimates.

arraylabel(*component*)
Batch submission script name for a job array.

collect(*stride=None*, *autosave=True*, *autocompress=True*)
Collect dV/dl from output.

Keywords

stride read data every *stride* lines, `None` uses the class default

autosave immediately save the class pickle fep file

autocompress compress the xvg file with bzip2 (saves >70% of space)

collect_alchemlyb(*SI=True*, *start=0*, *stop=None*, *stride=None*, *autosave=True*, *autocompress=True*)
Collect the data files using alchemlyb.

Unlike *collect()*, this method can subsample with the statistical inefficiency (parameter *SI*).

compress_dgdl_xvg()
Compress *all* dgdl xvg files with bzip2.

Note: After running this method you might want to run *collect()* to ensure that the results in `results.xvg` point to the *compressed* files. Otherwise `IOError` might occur which fail to find a *md.xvg* file.

contains_corrupted_xvgs()
Check if any of the source datafiles had reported corrupted lines.

Returns `True` if any of the xvg dgdl files were produced with the permissive=True flag and had skipped lines.

For debugging purposes, the number of corrupted lines are stored in `Gsolv._corrupted` as dicts of dicts with the component as primary and the lambda as secondary key.

convert_edr()
Convert EDR files to compressed XVG files.

dgdl_edr(**args*)
Return filename of the dgdl EDR file.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Returns path to EDR

Raises `IOError` with error code ENOENT if no file could be found

dgdl_total_edr(**args*, ***kwargs*)
Return filename of the combined dgdl EDR file.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Keywords

total_edr_name Name of the user defined total edr file.

Returns path to total EDR

dgdl_tpr(**args*)
Return filename of the dgdl TPR file.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Returns path to TPR

Raises `IOError` with error code ENOENT if no file could be found

dgdl_xvg(**args*)
Return filename of the dgdl XVG file.

Recognizes uncompressed, gzipped (gz), and bzip2ed (bz2) files.

Arguments

args joins the arguments into a path and adds the default filename for the dvdl file

Returns Checks if a compressed file exists and returns the appropriate filename.

Raises `IOError` with error code ENOENT if no file could be found

fep_dirs()
Generator for all simulation sub directories

frombase(**args*)
Return path with `Gsolv.basedir` prefixed.

get_protocol(*protocol*)
Return method for *protocol*.

- If *protocol* is a real method of the class then the method is returned.

- If *protocol* is a registered protocol name but no method of the name exists (i.e. *protocol* is a "dummy protocol") then a wrapper function is returned. The wrapper has the signature

  dummy_protocol (*func*, **args*, ***kwargs*)
  Runs *func* with the arguments and keywords between calls to `Journal.start()` and `Journal.completed()`, with the stage set to *protocol*.

- Raises a `ValueError` if the *protocol* is not registered (i.e. not found in `Journalled.protocols`).

has_dVdl()
Check if dV/dl data have already been collected.

Returns `True` if the dV/dl data have bee aquired (*`Gsolv.collect()`*) for all FEP simulations.

label (*component*)
Simple label for component, e.g. for use in filenames.

load (*filename=None*)
Re-instantiate class from pickled file.

If no *filename* was supplied then the filename is taken from the attribute `filename`.

Changed in version 0.7.1: Can read pickle files with either Python 2.7 or 3.x, regardless of the Python version that created the pickle.

logger_DeltaA0()
Print the free energy contributions (errors in parentheses).

plot (***kwargs*)
Plot the TI data with error bars.

Run *`mdpow.fep.Gsolv.analyze()`* first.

All *kwargs* are passed on to `pylab.errorbar()`.

Returns The axes of the subplot.

qsub (*script=None*)
Submit a batch script locally.

If *script* == `None` then take the first script (works well if only one template was provided).

save (*filename=None*)
Save instance to a pickle file.

The default filename is the name of the file that was last loaded from or saved to. Also sets the attribute `filename` to the absolute path of the saved file.

setup (***kwargs*)
Prepare the input files for all Gromacs runs.

Keywords

qscript (List of) template(s) for batch submission scripts; if not set then the templates are used that were supplied to the constructor.

kwargs Most kwargs are passed on to `gromacs.setup.MD()` although some are set to values that are required for the FEP functionality.

mdrun_opts list of options to **mdrun**; `-dhdl` is always added to this list as it is required for the thermodynamic integration calculations

includes list of directories where Gromacs input files can be found

The following keywords cannot be changed: *dirname*, *jobname*, *prefix*, *runtime*, *deffnm*. The last two can be specified when constructing `Gsolv`.

See also:

`gromacs.setup.MD()` and `gromacs.qsub.generate_submit_scripts()`

Changed in version 0.6.0: Gromacs now uses option `-dhdl` instead of `-dgdl`.

summary()
Return a string that summarizes the energetics.

Each energy component is followed by its error.

Format:

```
.                  ------ kJ/mol -----
molecule solvent  total  coulomb  vdw
```

tasklabel(*component*, *lmbda*)
Batch submission script name for a single task job.

wdir(*component*, *lmbda*)
Return rooted path to the work directory for *component* and *lmbda*.

(Constructed from `frombase()` and `wname()`.)

wname(*component*, *lmbda*)
Return name of the window directory itself.

Typically something like `VDW/0000`, `VDW/0500`, ..., `Coulomb/1000`

write_DeltaA0(*filename*, *mode='w'*)
Write free energy components to a file.

Arguments

filename name of the text file

mode 'w' for overwrite or 'a' for append ['w']

Format:: . —— kJ/mol —— molecule solvent total coulomb vdw

mdpow.fep.**pOW**(*G1*, *G2*, ***kwargs*)
Compute water-octanol partition coefficient from two `Gsolv` objects.

transfer free energy from water into octanol:

```
DeltaDeltaG0 = DeltaG0_oct - DeltaG0_water
```

octanol/water partition coefficient:

```
log P_oct/wat =  log [X]_oct/[X]_wat
```

Arguments

G1, G2 *G1* and *G2* should be a `Ghyd` and a `Goct` instance, but order does not matter

force force rereading of data files even if some data were already stored [False]

stride analyze every *stride*-th datapoint in the dV/dlambda files

start Start frame of data analyzed in every fep window.

stop Stop frame of data analyzed in every fep window.

SI Set to `True` if you want to perform statistical inefficiency to preprocess the data.

estimator Set to `alchemlyb` if you want to use alchemlyb estimators, or `mdpow` if you want the default TI method.

method Use *TI*, *BAR* or *MBAR* method in *alchemlyb*, or *TI* in *mdpow*.

Returns (transfer free energy, log10 of the octanol/water partition coefficient = log Pow)

`mdpow.fep.`**`pCW`**(*G1*, *G2*, ***kwargs*)

Compute water-cyclohexane partition coefficient from two *Gsolv* objects.

transfer free energy from water into cyclohexane:

```
DeltaDeltaG0 = DeltaG0_cyclohexane - DeltaG0_water
```

cyclohexane/water partition coefficient:

```
log P_CW =  log [X]_cyclohexane/[X]_water
```

Arguments

G1, G2 *G1* and *G2* should be a *Ghyd* and a *Gcyclo* instance, but order does not matter

force force rereading of data files even if some data were already stored [False]

stride analyze every *stride*-th datapoint in the dV/dlambda files

start Start frame of data analyzed in every fep window.

stop Stop frame of data analyzed in every fep window.

SI Set to `True` if you want to perform statistical inefficiency to preprocess the data.

estimator Set to `alchemlyb` if you want to use alchemlyb estimators, or `mdpow` if you want the default TI method.

method Use *TI*, *BAR* or *MBAR* method in *alchemlyb*, or *TI* in *mdpow*.

Returns (transfer free energy, log10 of the cyclohexane/water partition coefficient = log Pcw)

`mdpow.fep.`**`pTW`**(*G1*, *G2*, ***kwargs*)

Compute water-toluene partition coefficient from two *Gsolv* objects.

transfer free energy from water into toluene:

```
DeltaDeltaG0 = DeltaG0_tol - DeltaG0_water
```

toluene/water partition coefficient:

```
log P_tol/wat =  log [X]_tol/[X]_wat
```

Arguments

G1, G2 *G1* and *G2* should be a *Ghyd* and a *Gtol* instance, but order does not matter

force force rereading of data files even if some data were already stored [False]

stride analyze every *stride*-th datapoint in the dV/dlambda files

start Start frame of data analyzed in every fep window.

stop Stop frame of data analyzed in every fep window.

SI Set to `True` if you want to perform statistical inefficiency to preprocess the data.

estimator Set to `alchemlyb` if you want to use alchemlyb estimators, or `mdpow` if you want the default TI method.

method Use *TI*, *BAR* or *MBAR* method in *alchemlyb*, or *TI* in *mdpow*.

Returns (transfer free energy, log10 of the toluene/water partition coefficient = log Ptw)

5.5.4 Developer notes

A user really only needs to access classes derived from *mdpow.fep.Gsolv*; all other classes and functions are auxiliary and only of interest to developers.

Additional objects that support *mdpow.fep.Gsolv*.

class mdpow.fep.**FEPschedule**

Describe mdp parameter choices as key - value pairs.

The FEP schedule can be loaded from a configuration parser with the static method *FEPschedule.load()*.

See the example runinput file for an example. It contains the sections:

```
[FEP_schedule_Coulomb]
name = Coulomb
description = dis-charging vdw+q --> vdw
label = Coul
couple_lambda0 = vdw-q
couple_lambda1 = vdw
# soft core alpha: linear scaling for coulomb
sc_alpha = 0
lambdas = 0, 0.25, 0.5, 0.75, 1.0


[FEP_schedule_VDW]
name = vdw
description = decoupling vdw --> none
label = VDW
couple_lambda0 = vdw
couple_lambda1 = none
# recommended values for soft cores (Mobley, Shirts et al)
sc_alpha = 0.5
sc_power = 1
sc_sigma = 0.3
lambdas = 0.0, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.
↪9, 0.95, 1
```

static load(*cfg*, *section*)

Initialize a *FEPschedule* from the *section* in the configuration *cfg*

mdp_dict

Dict of key-values that can be set in a mdp file.

mdpow.fep.**molar_to_nm3**(*c*)

Convert a concentration in Molar to nm|^-3|.

mdpow.fep.**kcal_to_kJ**(*x*)

Convert a energy in kcal to kJ.

mdpow.fep.**kJ_to_kcal**(*x*)
Convert a energy in kJ to kcal.

mdpow.fep.**N_AVOGADRO**
Avogadro's constant N_A in mol^{-1} (NA NIST value).

mdpow.fep.**kBOLTZ**
Boltzmann's constant k_B in kJ mol^{-1} (kB NIST value).

5.5.4.1 TODO

- run minimization, NVT-equil, NPT-equil prior to production (probably use preprocessed topology from `grompp -pp` for portability)

  See Free Energy Tutorial.

5.6 Analysis

New in version 0.8.0.

The `mdpow.analysis` module contains tools for analyzing whole sets (ensembles) of FEP simulations and a framework to write new analysis tools [Lescoulie2021].

5.6.1 Analysis tools

MDPOW analysis tools are based on the *Ensemble Analysis Framework*. They generally take as an input the top level directory of a complete FEP run and then collect individual simulations and run a specific analysis over all FEP simulations. They then make data available (typically, as a `pandas.DataFrame`).

5.6.1.1 Solvation Shell Analysis

Analyzes the number of solvent molecules within given distances of the solute. The solvation tool is built with the framwework.

New in version 0.8.0.

class mdpow.analysis.solvation.**SolvationAnalysis**(*solute: mdpow.analysis.ensemble.EnsembleAtomGroup*, *solvent: mdpow.analysis.ensemble.EnsembleAtomGroup*, *distances: List[float]*)
Measures the number of solvent molecules withing the given distances in an *Ensemble* .

Parameters

solute An `EnsembleAtom` containing the solute used to measure distance.

solvent An `EnsembleAtom` containing the solvents counted in by the distance measurement. Each solvent atom is counted by the distance calculation.

distances Array like of the cutoff distances around the solute measured in Angstroms.

The data is returned in a `pandas.DataFrame` with observations sorted by distance, solvent, interaction, lambda, time.

Example

Typical Workflow:

```
ens = Ensemble(dirname='Mol')
solvent = ens.select_atoms('resname SOL and name OW')
solute = ens.select_atoms('resname UNK')

solv_dist = SolvationAnalysis(solute, solvent, [1.2, 2.4]).run(stop=10)
```

run (*start=None*, *stop=None*, *step=None*)
Runs `_single_universe()` on each system or `_single_frame()` on each frame in the system.

First iterates through keys of ensemble, then runs `_single_universe()` or `_single_frame()`.

5.6.1.2 Dihedral Analysis

Analyzes selected dihedral angles over a multi-system simulation. Built using the framwework to run over a collection of systems contained in an *Ensemble*.

New in version 0.8.0.

class mdpow.analysis.dihedral.**DihedralAnalysis**(*dihedral_groups: List[mdpow.analysis.ensemble.EnsembleAtomGroup]*)
Analyzes dihedral angles of selections from a single *Ensemble*.

Keywords

dihedral_groups list of *EnsembleAtomGroup* with four atoms selected on each. All selections must be from the same *Ensemble*.

Data is returned in a `pandas.DataFrame` with observations sorted by selection, solvent, interaction, lambda, time.

Example

Typical Workflow:

```
ens = Ensemble(dirname='Mol')

dihedral1 = Ens.select_atoms('name C1 or name C2 or name C3 or name C4')
dihedral2 = Ens.select_atoms('name C5 or name C8 or name C10 or name C12')

dih_run = DihedralAnalysis([dihedral1, dihedral2]).run(start=0, stop=10, step=1)
```

run (*start=None*, *stop=None*, *step=None*)
Runs `_single_universe()` on each system or `_single_frame()` on each frame in the system.

First iterates through keys of ensemble, then runs `_single_universe()` or `_single_frame()`.

5.6.2 Ensemble Analysis Framework

The *Ensemble Analysis framework* [Lescoulie2021] allows for the construction analysis tools that work with whole sets (ensembles) of FEP simulations. They generally follow (and can use) standard MDAnalysis analysis classes.

The *Ensemble Objects* and *Ensemble Analysis base class* sections assume a basic understanding of object oriented programming in Python and are for users who wish to construct their own analyses. The code in the *Analysis tools* serves as example implementations and is described in more detail in [Lescoulie2021].

5.6.2.1 Ensemble Analysis base class

New in version 0.8.0.

The Analysis modules help in the implementation analyses of MDPOW simulations. To simplify the process of analyzing a collection of systems generated by a free energy simulation the objects in *Ensemble Objects* allow for a molecule directory's systems to be loaded into MDAnalysis Universes and be analyzed as a group.

EnsembleAnalysis is a class inspired by the AnalysisBase from MDAnalysis which iterates over the systems in the ensemble or the frames in the systems. It sets up iterations between universes or universe frames allowing for analysis to be run on either whole systems or the frames of those systems. This allows for users to easily run analyses on MDPOW simulations.

NotImplementedError will detect whether _single_universe() or _single_frame() should be implemented, based on which is defined in the *EnsembleAnalysis*. Only one of the two methods should be defined for an *EnsembleAnalysis*. For verbose functionality, the analysis may show two iteration bars, where only one of which will actually be iterated, while the other will load to completion instantaneously, showing the system that is being worked on.

class mdpow.analysis.ensemble.**EnsembleAnalysis**(*ensemble=None*)

Base class for running multi-system analyses

The class is designed based on the AnalysisBase class in MDAnalysis and is a template for creating multi-universe multi-frame analyses using the *Ensemble* object

Keywords

ensemble The *Ensemble* being analyzed in the class

Example Analysis

Dihedral Analysis Demonstration:

```
class DihedralAnalysis(mdpow.ensemble.EnsembleAnalysis):
    def __init__(self, DihedralEnsembleGroup):
        super(DihedralAnalysis, self).__init__(DihedralEnsembleGroup.ensemble)

        self._sel = DihedralEnsembleGroup

    def _prepare_ensemble(self):
        self.result_dict = {}
        for s in ['water', 'octanol']:
            self.result_dict[s] = {'Coulomb': {},
                                   'VDW': {}}
        for key in self._sel.group_keys():
            self.result_dict[key[0]][key[1]][key[2]] = None

    def _prepare_universe(self):
        self.angle_dict = {'angle': None,
                           'time': None}
        self.angles = []

    def _single_frame(self):
```

(continues on next page)

(continued from previous page)

```
        angle = calc_dihedrals(self._sel[self._key].positions[0], self._sel[self._
↪key].positions[1],
                               self._sel[self._key].positions[2], self._sel[self._
↪key].positions[3])
        self.angles.append(angle)

    def _conclude_universe(self):
        self.angle_dict['time'] = self.times
        self.angle_dict['angle'] = self.angles
        self.result_dict[self._key[0]][self._key[1]][self._key[2]] = self.angle_
↪dict

    def _conclude_ensemble(self):
        self.results = pd.DataFrame(data=self.result_dict)

D = DihedralAnalysis.run(start=0 stop=10, step=1)
```

_prepare_ensemble()
For establishing data structures used in running analysis on the entire ensemble.

Data structures will not be overwritten upon moving to next system in ensemble.

_prepare_universe()
For establishing data structures used in running analysis on each trajectory in ensemble

Data structures will be overwritten between upon after each trajectory has been run

_single_universe()
Calculations on a single `MDAnalysis.Universe` object.

Run on each `MDAnalysis.Universe` in the *Ensemble* during when *run()* in called. `NotImplementedError` will detect whether *_single_universe()* or *_single_frame()* should be implemented, based on which is defined in the *EnsembleAnalysis*.

_single_frame()
Calculate data from a single frame of trajectory.

Called on each frame for each `MDAnalysis.Universe` in the *Ensemble*.

`NotImplementedError` will detect whether *_single_universe()* or *_single_frame()* should be implemented, based on which is defined in the *EnsembleAnalysis*.

_conclude_ensemble()
Run after all trajectories in ensemble are finished

_conclude_universe()
Run after each trajectory is finished

static check_groups_from_common_ensemble(*groups: List[mdpow.analysis.ensemble.EnsembleAtomGroup]*)
Checks if inputted list of *EnsembleAtomGroup* originate from the same *Ensemble*

Checks every *EnsembleAtomGroup* in list to determine if their `ensemble()` references the same object in memory. If two *EnsembleAtomGroup* object don't have a common *Ensemble* `ValueError` is raised.

run(*start=None*, *stop=None*, *step=None*)
Runs *_single_universe()* on each system or *_single_frame()* on each frame in the system.

First iterates through keys of ensemble, then runs `_single_universe()` or *_single_frame()*.

5.6.2.2 Ensemble Objects

New in version 0.8.0.

Ensemble

The *Ensemble* object is a collection of `MDAnalysis.Universe` objects. It is intended to store the set of systems generated by running **`mdpow-fep`**.

The *Ensemble* object works by storing the systems in a dictionary and extending the functionality of an `MDAnalysis.Universe` to a collection of universes. It when given a directory finds the simulation files, reads then loads them into a dictionary. The object can be indexed the same as a dictionary, and has methods analogous the the `MDAnalysis.Universe` object. The main one being *select_atoms()* which returns a *EnsembleAtomGroup* . An *Ensemble* in its current form can also be built by manually adding and popping universes into an empty instance.

class mdpow.analysis.ensemble.**Ensemble**(*dirname=None*, *solvents=('octanol', 'water')*, *topology_paths=None*, *interactions=('Coulomb', 'VDW')*, ***universe_kwargs*)

Collection of related `MDAnalysis.Universe` objects.

Stores systems produced by running mdpow-fep organized by solvent, interaction, and lambda.

Given a mdpow simulation directory will load the MD simulation files with the directory structure as keys.

Keywords

dirname Molecule Simulation directory. Loads simulation files present in lambda directories into the new instance. With this method for generating an *Ensemble* the lambda directories are explored and *_load_universe_from_dir()* searches for .gro, .gro.b2z, .gro.gz, and .tpr files for topology, and .xtc files for trajectory. It will default to using the tpr file available.

solvents Solvents from directory given to the new instance. Default `solvents=('water', 'octanol')`

topology_paths Specifies topologies used in loading simulated systems. Given with a dictionary with keys-value pair for each solvent and its respective topology path.

interactions Interactions from directory given to the instance. Default `interactions=('Coulomb', 'VDW')`

universe_kwargs Keywords arguments for loading `MDAnalysis.Universe` objects from MDPOW files in `dirname` argument directory when creating an *Ensemble* .

Examples

Typical workflow for MDPOW directory:

```
ens = Ensemble(dirname='molecule')
```

Typical workflow for adding universes individually:

```
ens = Ensemble()
u = mda.Universe(md.gro', 'md.xtc')
ens.add_system(u)
```

Topology paths can be specified when defining the _ensemble by giving the paths to each solvent topology in a dictionary with the topology_paths argument:

```
ens = Ensemble(dirname='molecule', topology_paths={'water': water_path,
                                                   'octanol': octanol_path}
```

Interactions can also be specified when initializing the with a list using the interactions argument:

```
ens = Ensemble(dirname='molecule', interactions=['Coulomb']
```

New in version 0.8.0.

_build_ensemble()
Finds simulation files genderated by MDPOW and attempts to build `MDAnalysis.Universe` in the lambda directories.

Run if `dirname` argument is given when initializing the class. First enters FEP directory, then traverses solvent and interaction directories to search lambda directories for system files.

static _load_universe_from_dir(*solv_dir=None*, ***universe_kwargs*) → Optional[MDAnalysis.core.universe.Universe]
Loads system simulation files in directory into an `MDAnalysis.Universe`

If multiple topologies are found it will default to using the .tpr file. If more than one trajectory is present they will be sorted alphabetically and passed into the `MDAnalysis.Universe` This method is run automatically by `_build_ensemble()` when initializing the class using the `dirname` argument.

add_system(*key*, *universe: MDAnalysis.core.universe.Universe*)
Adds system from universe object for trajectory and topology files

Existing mda.Universe object or trajectory and topology path. Ensure that paths are set to absolute when creating the universe.

keys()
Returns list of system keys

pop(*key*)
Removes and returns system at specified key.

Logs if KeyError is raised.

select_atoms(**args*, ***kwargs*)
Returns *EnsembleAtomGroup* containing selections from the *Ensemble*

Uses the same selection commands as MDAnalysis, and has the same keys as the *Ensemble*

select_systems(*keys=None*, *solvents=None*, *interactions=None*, *lambdas=None*, *lambda_range=None*)
Select specific subset of systems and returns them in an Ensemble.

This can be accomplished in two ways, by specific keys, or by specifying the desired system attributes solvents, interactions and lambdas. All arguments are stored in list form.

Keywords

keys System keys from *Ensemble* to be returned.

solvents Solvents from *Ensemble* to be returned.

interactions Interactions from *Ensemble* to be returned

lambdas Specific lambdas to be returned

lambda_range Range of lambda to be returned

Examples

Specific key workflow example:

```
Ens = Ensemble(dirname='Mol')
w_v_0_25 = Ens.select_systems(keys=[('water', 'VDW', '0000'),
                                    ('water', 'VDW', '0025')]
```

For the system attributes workflow there are two ways of selecting lambdas, the :param lambdas: keyword saves specific lambdas or the :param lambda_range: which saves the lambdas that fall within the given range.

Specific lambdas example:

```
Ens = Ensemble(dirname='Mol')
w_v_0_25 = Ens.select_systems(solvents=['water'], interactions=['VDW'],
                              lambdas=['0000', '0025'])
```

Range of lambdas example:

```
Ens = Ensemble(dirname='Mol')
w_v = Ens.select_systems(solvents=['water'], interactions=['VDW'],
                         lambda_range=[0, 1])
```

EnsembleAtomGroup

The *EnsembleAtomGroup* is created by running the on an *Ensemble* *select_atoms()*. It stores MDAnalysis.AtomGroup selections of the groups generated by running select atom on individual universes in a dictionary with the same key structure as the parent *Ensemble* class. It returns a copy of the parent *Ensemble* object when the *ensemble()* is run.

class mdpow.analysis.ensemble.**EnsembleAtomGroup**(*group_dict: dict*, *ensemble: mdpow.analysis.ensemble.Ensemble*)

Group for storing selections from *Ensemble* objects made using the *select_atoms()* method.

EnsembleAtomGroup is not set up for manual initialization, they should be obtained by selecting atoms from an existing object.

ensemble
Returns the ensemble of the EnsembleAtomGroup

keys()
List of keys to specific atom groups in the system

positions(*keys=None*)
Returns the positions of the keys of the selected atoms.

If no keys are specified positions for all keys are returned

select_atoms(**args*, ***kwargs*)
Returns *EnsembleAtomGroup* containing selections from the *EnsembleAtomGroup*

Uses the same selection commands as MDAnalysis, and has the same keys as *EnsembleAtomGroup*

5.6.3 References

5.7 Workflows

New in version 0.9.0.

MDPOW analysis workflows provide automation functions for use with analysis *Analysis tools* based on the *Ensemble Analysis Framework*. Existing workflow modules, *dihedrals*, include functions for use with *DihedralAnalysis*.

5.7.1 Workflows Base

New in version 0.9.0.

5.7.1.1 mdpow.workflows.base — Automated workflow base functions

To analyze multiple MDPOW projects, provide *project_paths()* with the top-level directory containing all MDPOW projects' simulation data to obtain a pandas.DataFrame containing the project information and paths. Then, *automated_project_analysis()* takes as input the aforementioned pandas.DataFrame and runs the specified *EnsembleAnalysis* for all MDPOW projects under the top-level directory provided to *project_paths()*.

See also:

registry

mdpow.workflows.base.**project_paths**(*parent_directory=None*, *csv=None*, *csv_save_dir=None*)

Takes a top directory containing MDPOW projects and determines the molname, resname, and path, of each MDPOW project within.

Optionally takes a .csv file containing *molname*, *resname*, and *paths*, in that order.

Keywords

parent_directory the path for the location of the top directory under which the subdirectories of MDPOW simulation data exist, additionally creates a 'project_paths.csv' file for user manipulation of metadata and for future reference

csv .csv file containing the molecule names, resnames, and paths, in that order, for the MDPOW simulation data to be iterated over must contain header of the form: *molecule,resname,path*

csv_save_dir optionally provided directory to save .csv file, otherwise, data will be saved in current working directory

Returns

project_paths pandas.DataFrame containing MDPOW project metadata

Example

Typical Workflow:

```
project_paths = project_paths(parent_directory='/foo/bar/MDPOW_projects')
automated_project_analysis(project_paths)
```

or:

```
project_paths = project_paths(csv='/foo/bar/MDPOW.csv')
automated_project_analysis(project_paths)
```

mdpow.workflows.base.**automated_project_analysis**(*project_paths*, *ensemble_analysis*, ***kwargs*)

Takes a `pandas.DataFrame` created by *`project_paths()`* and iteratively runs the specified *`EnsembleAnalysis`* for each of the projects by running the associated automated workflow in each project directory returned by *`project_paths()`*.

Compatibility with more automated analyses in development.

Keywords

project_paths `pandas.DataFrame` that provides paths to MDPOW projects

ensemble_analysis name of the *`EnsembleAnalysis`* that corresponds to the desired automated workflow module

kwargs keyword arguments for the supported automated workflows, see the *`registry`* for all available workflows and their call signatures

Example

A typical workflow is the automated dihedral analysis from *`mdpow.workflows.dihedrals`*, which applies the *ensemble analysis* `DihedralAnalysis` to each project. The *`registry`* contains this automated workflow under the key *"DihedralAnalysis"* and so the automated execution for all *project_paths* (obtained via *`project_paths()`*) is performed by passing the specific key to *`automated_project_analysis()`*:

```
project_paths = project_paths(parent_directory='/foo/bar/MDPOW_projects')
automated_project_analysis(project_paths, ensemble_analysis='DihedralAnalysis',␣
↪**kwargs)
```

mdpow.workflows.base.**guess_elements**(*atoms*, *rtol=0.001*)

guess elements for atoms from masses

Given masses, we perform a reverse lookup on `MDAnalysis.topology.tables.masses` to find the corresponding element. Only atoms where the standard MDAnalysis guesser finds elements with masses contradicting the topology masses are corrected.

Note: This function *requires* correct masses to be present. No sanity checks because MDPOW always uses TPR files that contain correct masses.

Arguments

atoms MDAnalysis AtomGroup *with masses defined*

Keywords

rtol relative tolerance for a match (as used in `numpy.isclose()`); atol=1e-6 is at a fixed value, which means that "zero" is only recognized for values =< 1e-6

Note: In order to reliably match GROMACS masses, *rtol* should be at least 1e-3.

Returns

elements array of guessed element symbols, in same order as *atoms*

Example

As an example we guess masses and then set the elements for all atoms:

```
elements = guess_elements(atoms)
atoms.add_TopologyAttr("elements", elements)
```

5.7.2 Workflows Registry

New in version 0.9.0.

5.7.2.1 `mdpow.workflows.registry` — Registry of currently supported automated workflows

The *mdpow.workflows.registry* module hosts a dictionary with keys that correspond to an *EnsembleAnalysis* for which exists a corresponding automated workflow.

Table 3: Currently supported automated workflows.

key/keyword: EnsembleAnalysis	value: <workflow module>.<top-level automated analysis function>
DihedralAnalysis	*dihedrals.automated_dihedral_analysis*

mdpow.workflows.registry.**registry = {'DihedralAnalysis': <function automated_dihedral_anal**

In the *registry*, each entry corresponds to an *EnsembleAnalysis* for which exists a corresponding automated workflow.

Intended for use with *mdpow.workflows.base* to specify which *EnsembleAnalysis* should run iteratively over the provided project data directory.

To include a new automated workflow for use with *mdpow.workflows.base*, create a key that is the name of the corresponding *EnsembleAnalysis*, with the value defined as <*workflow module*>.<*top-level automated analysis function*>.

The available automated workflows (key-value pairs) are listed in the following table *Currently supported automated workflows.*

See also:

base

5.7.3 Automated Dihedral Analysis

New in version 0.9.0.

5.7.3.1 `mdpow.workflows.dihedrals` — Automation for `DihedralAnalysis`

dihedrals module provides functions for automated workflows that encompass *DihedralAnalysis*. See each function for requirements and examples.

Most functions can be used as standalone, individually, or in combination depending on the desired results. Details of the completely automated workflow are discussed under *automated_dihedral_analysis()*.

Atom indices obtained by *get_atom_indices()* are 0-based, atom index labels on the molecule in the plots are 0-based, but atom *names* in plots and file names are 1-based.

mdpow.workflows.dihedrals.**SOLVENTS_DEFAULT = ('water', 'octanol')**
Default solvents are water and octanol:

- must match solvents used in project directory
- one or two solvents can be specified
- current solvents supported,

See also:

mdpow.forcefields

mdpow.workflows.dihedrals.**INTERACTIONS_DEFAULT = ('Coulomb', 'VDW')**
Default interactions set to Coulomb and VDW:

- default values should not be changed
- order should not be changed

mdpow.workflows.dihedrals.**SMARTS_DEFAULT = [!#1]~[!$(*#*)&!D1]-!@[!$(*#*)&!D1]~[!#1]**
Default SMARTS string to identify relevant dihedral atom groups:

- `[!#1]` : any atom, not Hydrogen
- `~` : any bond
- `[!$(*#*)&!D1]` : any atom that is not part of linear triple bond and not atom with 1 explicit bond
- `-!@` : single bond that is not ring bond
- `[!$(*#*)&!D1]-!@[!$(*#*)&!D1]` : the central portion selects two atoms that are not involved in a triple bond and are not terminal, that are connected by a single, non-ring bond
- `[!#1]~` or `~[!#1]` : the first and last portion specify any bond, to any atom that is not hydrogen

mdpow.workflows.dihedrals.**PLOT_WIDTH_DEFAULT = 190**
Plot width (*plot_pdf_width*) should be provided in millimeters (mm), and is converted to pixels (px) for use with `cairosvg`.

conversion factor: 1 mm = 3.7795275591 px default value: 190 mm = 718.110236229 pixels

mdpow.workflows.dihedrals.**automated_dihedral_analysis**(*dirname*, *resname*, *figdir=None*, *df_save_dir=None*, *molname=None*, *SMARTS='[!#1]~[!$(*#*)&!D1]-!@[!$(*#*)&!D1]~[!#1]'*, *plot_pdf_width=190*, *dataframe=None*, *padding=45*, *width=0.9*, *solvents=('water', 'octanol')*, *interactions=('Coulomb', 'VDW')*, *start=None*, *stop=None*, *step=None*)
Runs *DihedralAnalysis* for a single MDPOW project and creates violin plots of dihedral angle frequencies for each relevant dihedral atom group.

For one MDPOW project, automatically determines all relevant dihedral atom groups in the molecule, runs *DihedralAnalysis* for each group, pads the dihedral angles to maintain periodicity, creates violin plots of dihedral angle frequencies (KDEs), and saves publication quality PDF figures for each group, separately.

Optionally saves all pre-padded *DihedralAnalysis* results as a single pandas.DataFrame in *df_save_dir* provided.

Keywords

dirname Molecule Simulation directory. Loads simulation files present in lambda directories into the new instance. With this method for generating an *Ensemble* the lambda directories are explored and *_load_universe_from_dir()* searches for .gro, .gro.bz2, .gro.gz, and .tpr files for topology, and .xtc files for trajectory. It will default to using the tpr file available.

figdir path to the location to save figures (REQUIRED but marked as a kwarg for technical reasons; will be changed in #244)

resname *resname* for the molecule as defined in the topology and trajectory

df_save_dir optional, path to the location to save results pandas.DataFrame

molname molecule name to be used for labelling plots, if different from *resname*

SMARTS The default SMARTS string is described in detail under *SMARTS_DEFAULT*.

plot_pdf_width The default value for width of plot output is described in detail under *PLOT_WIDTH_DEFAULT*.

dataframe optional, if *DihedralAnalysis* was done prior, then results pandas.DataFrame can be input to utilize angle padding and violin plotting functionality

padding value in degrees default: 45

See also:

periodic_angle_padding()

width width of the violin element (>1 overlaps) default: 0.9

See also:

dihedral_violins()

solvents The default solvents are documented under *SOLVENTS_DEFAULT*. Normally takes a two-tuple, but analysis is compatible with single solvent selections. Single solvent analyses will result in a figure with fully filled violins for the single solvent.

interactions The default interactions are documented under *INTERACTIONS_DEFAULT*.

start, stop, step arguments passed to *run()*, as parameters for iterating through the trajectories of the current ensemble

See also:

EnsembleAnalysis

Example

Typical Workflow:

```
import dihedrals

dihedrals.automated_dihedral_analysis(dirname='/foo/bar/MDPOW_project_data',
                                      figdir='/foo/bar/MDPOW_figure_directory',
                                      resname='UNK', molname='benzene',
                                      padding=45, width=0.9,
                                      solvents=('water','octanol'),
                                      interactions=('Coulomb','VDW'),
                                      start=0, stop=100, step=10)
```

mdpow.workflows.dihedrals.**build_universe**(*dirname*, *solvents=('water', 'octanol')*)

Builds `Universe` from the `./Coulomb/0000` topology and trajectory of the project for the first solvent specified.

Output used by *`rdkit_conversion()`* and *`get_atom_indices()`* to obtain atom indices for each dihedral atom group.

Keywords

dirname

Molecule Simulation directory. Loads simulation files present in lambda directories into the new instance. With this method for generating an *`Ensemble`* the lambda directories are explored and *`_load_universe_from_dir()`* searches for .gro, .gro.bz2, .gro.gz, and .tpr files for topology, and .xtc files for trajectory. It will default to using the tpr file available.

solvents The default solvents are documented under *`SOLVENTS_DEFAULT`*. Normally takes a two-tuple, but analysis is compatible with single solvent selections. Single solvent analyses will result in a figure with fully filled violins for the single solvent.

Returns

u `Universe` object

mdpow.workflows.dihedrals.**rdkit_conversion**(*u*, *resname*)

Converts the solute, *resname*, of the `Universe` to `rdkit.Chem.rdchem.Mol` object for use with a SMARTS selection string to identify dihedral atom groups.

Accepts `Universe` object made with *`build_universe()`* and a *resname* as input. Uses *resname* to select the solute for conversion by `RDKitConverter` to `rdkit.Chem.rdchem.Mol`, and will add element attributes for Hydrogen if not listed in the topology, using `MDAnalysis.topology.guessers.guess_atom_element()`.

Keywords

u `Universe` object

resname *resname* for the molecule as defined in the topology and trajectory

Returns

tuple(mol, solute) function call returns tuple, see below

mol `rdkit.Chem.rdchem.Mol` object converted from *solute*

solute the `MDAnalysis` *AtomGroup* for the solute

mdpow.workflows.dihedrals.**get_atom_indices**(*mol*, *SMARTS='[!#1]~[!$(*#*)&!D1]-!@[!$(*#*)&!D1]~[!#1]'*)

Uses a SMARTS selection string to identify atom indices for relevant dihedral atom groups.

Requires a `rdkit.Chem.rdchem.Mol` object as input for the *SMARTS_DEFAULT* kwarg to match patterns to and identify relevant dihedral atom groups.

Keywords

mol `rdkit.Chem.rdchem.Mol` object converted from *solute*

SMARTS The default SMARTS string is described in detail under *SMARTS_DEFAULT*.

Returns

atom_indices tuple of tuples of indices for each dihedral atom group

mdpow.workflows.dihedrals.**get_bond_indices**(*mol*, *atom_indices*)

From the `rdkit.Chem.rdchem.Mol` object, uses *atom_indices* to determine the indices of the bonds between those atoms for each dihedral atom group.

Keywords

mol `rdkit.Chem.rdchem.Mol` object converted from *solute*

atom_indices tuple of tuples of indices for each dihedral atom group

Returns

bond_indices tuple of tuples of indices for the bonds in each dihedral atom group

mdpow.workflows.dihedrals.**get_dihedral_groups**(*solute*, *atom_indices*)

Uses the 0-based *atom_indices* of the relevant dihedral atom groups determined by *get_atom_indices()* and returns the 1-based index names for each atom in each group.

Requires the *atom_indices* from *get_atom_indices()* to index the *solute* specified by `select_atoms()` and return an array of the names of each atom within its respective dihedral atom group as identified by the SMARTS selection string.

Keywords

solute the `MDAnalysis` *AtomGroup* for the solute

atom_indices tuple of tuples of indices for each dihedral atom group

Returns

dihedral_groups list of `numpy.array()` for atom names in each dihedral atom group

mdpow.workflows.dihedrals.**get_paired_indices**(*atom_indices*, *bond_indices*, *dihedral_groups*)

Combines *atom_indices* and *bond_indices* in tuples to be paired with their respective dihedral atom groups.

A dictionary is created with key-value pairs as follows: *atom_indices* and *bond_indices* are joined in a tuple as the value, with the key being the respective member of *dihedral_groups* to facilitate highlighting the relevant dihedral atom group when generating violin plots. As an example, *'C1-N2-O3-S4': ((0, 1, 2, 3), (0, 1, 2))*, would be one key-value pair in the dictionary.

Keywords

atom_indices tuple of tuples of indices for each dihedral atom group

bond_indices tuple of tuples of indices for the bonds in each dihedral atom group

dihedral_groups list of `numpy.array()` for atom names in each dihedral atom group

Returns

name_index_pairs dictionary with key-value pair for dihedral atom group, atom indices, and bond indices

mdpow.workflows.dihedrals.**dihedral_groups_ensemble**(*dirname*, *atom_indices*, *solvents=('water', 'octanol')*, *interactions=('Coulomb', 'VDW')*, *start=None*, *stop=None*, *step=None*)

Creates one *Ensemble* for the MDPOW project and runs *DihedralAnalysis* for each dihedral atom group identified by the SMARTS selection string.

See also:

automated_dihedral_analysis(), *DihedralAnalysis*

Keywords

dirname Molecule Simulation directory. Loads simulation files present in lambda directories into the new instance. With this method for generating an *Ensemble* the lambda directories are explored and *_load_universe_from_dir()* searches for .gro, .gro.bz2, .gro.gz, and .tpr files for topology, and .xtc files for trajectory. It will default to using the tpr file available.

atom_indices tuples of atom indices for dihedral atom groups

See also:

get_atom_indices(), *SMARTS_DEFAULT*

solvents The default solvents are documented under *SOLVENTS_DEFAULT*. Normally takes a two-tuple, but analysis is compatible with single solvent selections. Single solvent analyses will result in a figure with fully filled violins for the single solvent.

interactions The default interactions are documented under *INTERACTIONS_DEFAULT*.

start, stop, step arguments passed to *run()*, as parameters for iterating through the trajectories of the current ensemble

See also:

EnsembleAnalysis

Returns

df `pandas.DataFrame` of *DihedralAnalysis* results, including all dihedral atom groups for molecule of current project

mdpow.workflows.dihedrals.**save_df**(*df*, *df_save_dir*, *resname*, *molname=None*)

Takes a `pandas.DataFrame` of results from *DihedralAnalysis* as input before padding the angles to optionaly save the raw data.

Optionally saves results before padding the angles for periodicity and plotting dihedral angle frequencies as KDE violins with *dihedral_violins()*. Given a parent directory, creates subdirectory for molecule, saves fully sampled, unpadded results `pandas.DataFrame` as a compressed csv file, default: .csv.bz2.

Keywords

df pandas.DataFrame of *DihedralAnalysis* results, including all dihedral atom groups for molecule of current project

df_save_dir optional, path to the location to save results pandas.DataFrame

resname *resname* for the molecule as defined in the topology and trajectory

molname molecule name to be used for labelling plots, if different from *resname*

mdpow.workflows.dihedrals.**periodic_angle_padding**(*df*, *padding=45*)

Pads the angles from the results DataFrame to maintain periodicity in the violin plots.

Takes a pandas.DataFrame of results from *DihedralAnalysis* or *dihedral_groups_ensemble()* as input and pads the angles to maintain periodicity for properly plotting dihedral angle frequencies as KDE violins with *dihedral_violins()* and *plot_dihedral_violins()*. Creates two new pandas.DataFrame based on the *padding* value specified, pads the angle values, concatenates all three pandas.DataFrame, maintaining original data and adding padded values, and returns new augmented pandas.DataFrame.

Keywords

df pandas.DataFrame of *DihedralAnalysis* results, including all dihedral atom groups for molecule of current project

padding value in degrees to specify angle augmentation threshold default: 45

Returns

df_aug augmented results pandas.DataFrame containing padded dihedral angles as specified by *padding*

mdpow.workflows.dihedrals.**dihedral_violins**(*df*, *width=0.9*, *solvents=('water', 'octanol')*, *plot_title=None*)

Plots kernel density estimates (KDE) of dihedral angle frequencies for one dihedral atom group as violin plots, using as input the augmented pandas.DataFrame from *periodic_angle_padding()*.

Output is converted to SVG by *build_svg()* and final output is saved as PDF by *plot_dihedral_violins()*

Keywords

df augmented results pandas.DataFrame from *periodic_angle_padding()*

width width of the violin element (>1 overlaps) default: 0.9

solvents The default solvents are documented under *SOLVENTS_DEFAULT*. Normally takes a two-tuple, but analysis is compatible with single solvent selections. Single solvent analyses will result in a figure with fully filled violins for the single solvent.

plot_title generated by *build_svg()* using *molname*, *dihedral_groups*, *atom_indices*, and *interactions* in this order and format: f'{molname}, {name[0]} {a} | ''{col_name}'

Returns

g returns a seaborn.FacetGrid object containing a violin plot of the kernel density estimates (KDE) of the dihedral angle frequencies for each dihedral atom group identified by *SMARTS_DEFAULT*

mdpow.workflows.dihedrals.**build_svg**(*mol*, *molname*, *name_index_pairs*, *atom_group_selection*, *solvents=('water', 'octanol')*, *width=0.9*)

Converts and combines figure components into an SVG object to be converted and saved as a publication quality PDF.

Keywords

mol `rdkit.Chem.rdchem.Mol` object converted from *solute*

molname molecule name to be used for labelling plots, if different from *resname* (in this case, carried over from an upstream

decision between the two)

name_index_pairs dictionary with key-value pair for dihedral atom group, atom indices, and bond indices

See also:

get_paired_indices()

atom_group_selection *name* of each section in the *groupby* series of atom group selections

See also:

plot_dihedral_violins()

solvents The default solvents are documented under *SOLVENTS_DEFAULT*. Normally takes a two-tuple, but analysis is compatible with single solvent selections. Single solvent analyses will result in a figure with fully filled violins for the single solvent.

width width of the violin element (>1 overlaps) default: 0.9

Returns

fig `svgutils` SVG figure object

mdpow.workflows.dihedrals.**plot_dihedral_violins**(*df*, *resname*, *mol*, *name_index_pairs*, *figdir=None*, *molname=None*, *width=0.9*, *plot_pdf_width=190*, *solvents=('water', 'octanol')*)

Coordinates plotting and saving figures for all dihedral atom groups.

Makes a subdirectory for the current project within the specified *figdir* using *resname* or *molname* as title and saves production quality PDFs for each dihedral atom group separately.

See also:

automated_dihedral_analysis(), *dihedral_violins()*, *build_svg()*

Keywords

df augmented results `pandas.DataFrame` from *periodic_angle_padding()*

resname *resname* for the molecule as defined in the topology and trajectory

mol `rdkit.Chem.rdchem.Mol` object converted from *solute*

name_index_pairs dictionary with key-value pair for dihedral atom group, atom indices, and bond indices

See also:

get_paired_indices()

figdir path to the location to save figures (REQUIRED but marked as a kwarg for technical reasons; will be changed in #244)

molname molecule name to be used for labelling plots, if different from *resname*

width width of the violin element (>1 overlaps) default: 0.9

See also:

dihedral_violins()

plot_pdf_width The default value for width of plot output is described in detail under *PLOT_WIDTH_DEFAULT*.

solvents The default solvents are documented under *SOLVENTS_DEFAULT*. Normally takes a two-tuple, but analysis is compatible with single solvent selections. Single solvent analyses will result in a figure with fully filled violins for the single solvent.

5.8 Helper modules

The code described here is only relevant for developers.

5.8.1 `mdpow.config` – Configuration for MDPOW

The config module provides configurable options for the whole package; eventually it might grow into a more sophisticated configuration system but right now it mostly serves to define which gromacs tools and other scripts are offered in the package and where template files are located. If the user wants to change anything they will still have to do it here in source until a better mechanism with a global configuration file has been implemented.

5.8.1.1 Force field

By default, MDPOW uses a collection of OPLS/AA force field files based on the Gromacs 4.5.3 distribution, with the following differences:

- For ions we use the new alkali and halide ion parameters from Table 2 in [Jensen2006] which had shown some small improvements in the paper. They should only be used with the TIP4P water model.
- OPLS/AA parameters for 1-octanol were added. These parameters were validated against experimental data by computing the density (neat), hydration free energy and logP (the latter being a self consistency check).

The force field files are found in the directory pointed to by the environment variable `GMXLIB`. By default, *mdpow.config* sets `GMXLIB` to *includedir* unless `GMXLIB` has already been set. This mechanism allows the user to override the choice of location of force field.

At the moment, only OPLS/AA is tested with MDPOW although in principle it is possible to use other force fields by supplying appropriately customized template files.

References

5.8.1.2 Location of template files

Template variables list files in the package that can be used as templates such as run input files. Because the package can be a zipped egg we actually have to unwrap these files at this stage but this is completely transparent to the user.

mdpow.config.**templates = {'NPT_amber.mdp': '/home/docs/checkouts/readthedocs.org/user_bui**
POW comes with a number of templates for run input files and queuing system scripts. They are provided as a convenience and examples but **WITHOUT ANY GUARANTEE FOR CORRECTNESS OR SUITABILITY FOR ANY PURPOSE**.

All template filenames are stored in `gromacs.config.templates`. Templates have to be extracted from the GromacsWrapper python egg file because they are used by external code: find the actual file locations from this variable.

Gromacs mdp templates

These are supplied as examples and there is *NO GUARANTEE THAT THEY PRODUCE SENSIBLE OUTPUT* — check for yourself! Note that only existing parameter names can be modified with `gromacs.cbook.edit_mdp()` at the moment; if in doubt add the parameter with its gromacs default value (or empty values) and modify later with `edit_mdp()`.

The safest bet is to use one of the `mdout.mdp` files produced by `gromacs.grompp()` as a template as this mdp contains all parameters that are legal in the current version of Gromacs.

mdpow.config.**topfiles = {'1cyclo.gro': '/home/docs/checkouts/readthedocs.org/user_builds/**
List of all topology files that are included in the package. (includes force field files under `top/oplsaa.ff`)

mdpow.config.**includedir = '/home/docs/checkouts/readthedocs.org/user_builds/mdpow/checkout**
The package's include directory for `gromacs.grompp()`; the environment variable `GMXLIB` is set to *`includedir`* so that the bundled version of the force field is picked up.

mdpow.config.**defaults = {'runinput': '/home/docs/checkouts/readthedocs.org/user_builds/md**
Locations of default run input files and configurations.

5.8.1.3 Functions

The following functions can be used to access configuration data.

mdpow.config.**get_template**(*t*)
Find template file *t* and return its real path.

t can be a single string or a list of strings. A string should be one of

1. a relative or absolute path,
2. a filename in the package template directory (defined in the template dictionary `gromacs.config.templates`) or
3. a key into `templates`.

The first match (in this order) is returned. If the argument is a single string then a single string is returned, otherwise a list of strings.

Arguments *t* : template file or key (string or list of strings)

Returns os.path.realpath(*t*) (or a list thereof)

Raises `ValueError` if no file can be located.

mdpow.config.**get_templates**(*t*)
Find template file(s) *t* and return their real paths.

t can be a single string or a list of strings. A string should be one of

1. a relative or absolute path,
2. a filename in the package template directory (defined in the template dictionary `gromacs.config.templates`) or

3. a key into `templates`.

The first match (in this order) is returned for each input argument.

Arguments *t* : template file or key (string or list of strings)

Returns list of os.path.realpath(*t*)

Raises `ValueError` if no file can be located.

`mdpow.config.`**`get_configuration`**(*filename=None*)
Reads and parses a run input config file.

Uses the package-bundled defaults as a basis.

Developer note

Templates have to be extracted from the egg because they are used by external code. All template filenames are stored in `config.templates` or `config.topfiles`.

Sub-directories are extracted (see Resource extraction) but the file names themselves will not appear in the template dict. Thus, only store files in subdirectories that don't have to be explicitly found by the package (e.g. the Gromacs force field files are ok).

`mdpow.config.`**`_generate_template_dict`**(*dirname*)
Generate a list of included top-level files *and* extract them to a temp space.

Only lists files and directories at the *top level* of the *dirname*; however, all directories are extracted recursively and will be available.

5.8.1.4 Exceptions

exception `mdpow.config.`**`NoSectionError`**
Section entry is missing.

New in version 0.8.0.

class `mdpow.config.`**`NoOptionWarning`**
Warning that an option is missing.

5.8.2 `mdpow.log` — Configure logging for POW analysis

Import this module if logging is desired in application code and create the logger in `__init__.py`:

```
import log
logger = log.create(logname, logfile)
```

In modules simply use:

```
import logging
logger = logging.getLogger(logname)
```

5.8.3 `mdpow.restart` — Restarting and checkpointing

The module provides classes and functions to keep track of which stages of a simulation protocol have been completed. It uses a *`Journal`* class for the book-keeping. Together with saving the current state of a protocol to a

checkpoint file (using *Journalled.save()*) it is possible to implement restartable simulation protocols (for example **mdpow-equilibrium**).

exception mdpow.restart.**JournalSequenceError**
Raised when a stage is started without another one having been completed.

class mdpow.restart.**Journal**(*stages*)
Class that keeps track of the stage in a protocol.

Transaction blocks have to be bracketed by calls to *start()* and *completed()*. If a block is started before completion, a *JournalSequenceError* will be raised.

Other methods such as *has_completed()* and *has_not_completed()* can be used to query the status. The attribute *incomplete* flags the state of the current stage (*current*).

All completed stages are recorded in the attribute *history*.

The current (incomplete) stage can be reset to its initial state with *Journal.clear()*.

Example:

```
J = Journal(['pre', 'main', 'post'])
J.start('pre')
...
J.completed('pre')
J.start('main')
...
# main does not finish properly
print(J.incomplete)
# --> 'main'
J.start('post')
# raises JournalSequenceError
```

Initialise the journal that keeps track of stages.

Arguments

stages list of the stage identifiers, in the order that they should per performed. Stage identifiers are checked against this list before they are accepted as arguments to most methods.

clear()
Reset incomplete status and current stage

completed(*stage*)
Record completed stage and reset *Journal.current*

current
Current stage identifier

has_completed(*stage*)
Returns `True` if the *stage* has been started and completed at any time.

has_not_completed(*stage*)
Returns `True` if the *stage* had been started but not completed yet.

history
List of stages completed

incomplete
This last stage was not completed.

start(*stage*)
Record that *stage* is starting.

class mdpow.restart.**Journalled**(**args*, ***kwargs*)
A base class providing methods for journalling and restarts.

It installs an instance of *Journal* in the attribute `Journalled.journal` if it does not exist already.

get_protocol(*protocol*)
Return method for *protocol*.

- If *protocol* is a real method of the class then the method is returned.
- If *protocol* is a registered protocol name but no method of the name exists (i.e. *protocol* is a "dummy protocol") then a wrapper function is returned. The wrapper has the signature

  dummy_protocol(*func*, **args*, ***kwargs*)
  Runs *func* with the arguments and keywords between calls to *Journal.start()* and *Journal.completed()*, with the stage set to *protocol*.
- Raises a `ValueError` if the *protocol* is not registered (i.e. not found in *Journalled.protocols*).

load(*filename=None*)
Re-instantiate class from pickled file.

If no *filename* was supplied then the filename is taken from the attribute `filename`.

Changed in version 0.7.1: Can read pickle files with either Python 2.7 or 3.x, regardless of the Python version that created the pickle.

protocols = []
Class-attribute that contains the names of computation protocols supported by the class. These are either method names or dummy names, whose logic is provided by an external callback function. The method *get_protocol()* raises a `ValueError` if a protocol is not listed in *protocols*.

save(*filename=None*)
Save instance to a pickle file.

The default filename is the name of the file that was last loaded from or saved to. Also sets the attribute `filename` to the absolute path of the saved file.

mdpow.restart.**checkpoint**(*name*, *sim*, *filename*)
Execute the *Journalled.save()* method and log the event.

5.8.4 mdpow.run — Performing complete simulation protocols

The module provides building blocks for complete simulation protocols (or pipelines). Each protocol is written as a function that takes a run input file and the solvent type as input.

The mdpow- scripts* make use of the building blocks.

Typically, *journalling* is enabled, i.e. the tasks remember which stages have already been completed and can be restarted directly from the last completed stage. (Restarts are only implemeneted at the level of individual steps in a MDPOW protocol, not at the level of continuing interrupted simulations using the Gromacs restart files.)

Input is read from the run input *cfg* file.

See also:

mdpow.restart for the journalling required for restarts.

5.8.4.1 Protocols

mdpow.run.**equilibrium_simulation**(*cfg*, *solvent*, ***kwargs*)
Set up and run equilibrium simulation.

See tutorial for the individual steps.

Note: Depending on the settings in the run input file (`runlocal`), **mdrun** is executed at various stages, and hence this process can take a while.

---

mdpow.run.**fep_simulation**(*cfg*, *solvent*, ***kwargs*)
Set up and run FEP simulation.

See tutorial for the individual steps.

Note: Depending on the settings in the run input file (`runlocal`), **mdrun** is executed sequentially for all windows and hence this can take a long time. It is recommended to use `runlocal = False` in the run input file and submit all window simulations to a cluster.

5.8.4.2 Support

mdpow.run.**setupMD**(*S*, *protocol*, *cfg*)
setup MD simulation *protocol* using the runinput file *cfg*

mdpow.run.**get_mdp_files**(*cfg*, *protocols*)
Get file names of MDP files from *cfg* for all *protocols*

mdpow.run.**runMD_or_exit**(*S*, *protocol*, *params*, *cfg*, *exit_on_error=True*, ***kwargs*)
run simulation

Can launch **mdrun** itself (`gromacs.run.MDrunner`) or exit so that the user can run the simulation independently. Checks if the simulation has completed and sets the return value to `True` if this is the case.

- Configuration parameters are taken from the section *protocol* of the runinput configuration *cfg*.
- The directory in which the simulation input files reside can be provided as keyword argument *dirname* or taken from *S.dirs[protocol]*.
- The *exit_on_error* kwarg determines if `sys.exit()` is called if mdrun fails to complete (`True`, default) or if instead we raise an `gromacs.exceptions.GromacsError` (`False`).
- Other *kwargs* are interpreted as options for `Mdrun`.

Changed in version 0.9.0: New kwarg *exit_on_error*.

5.9 Force field selection

The *mdpow.forcefields* module contains settings for selecting different force fields and the corresponding solvent topologies.

The OPLS-AA, CHARMM/CGENFF and the AMBER/GAFF force field are directly supported. It is possible to use a different forcefield by implementing a *Forcefield* with the correct files and supplying suitable `.mdp` files. For an example of how to do this, look at the `martini-example.ipynb` under `doc/examples/martini-example`.

mdpow.forcefields.**DEFAULT_FORCEFIELD = 'OPLS-AA'**
Default force field. At the moment, OPLS-AA, CHARMM/CGENFF, and AMBER/GAFF are directly supported. However, it is not recommended to change the default here as this behavior is not tested.

5.9.1 Solvent models

Different **water models** are already supported

mdpow.forcefields.**GROMACS_WATER_MODELS = {'m24': <M24 water: identifier=m24, ff=OPLS-AA>,**
Dictionary of *GromacsSolventModel* instances, one for each Gromacs water model available under the force field directory. The keys are the water model identifiers. For OPLS-AA the following ones are available.

as well as different general **solvent models**

mdpow.forcefields.**GROMACS_SOLVENT_MODELS = {'AMBER': {'cyclohexane': <CYCLOHEXANE water:**
Solvents available in GROMACS; the keys of the dictionary are the forcefields.

5.9.2 Internal data

mdpow.forcefields.**SPECIAL_WATER_COORDINATE_FILES = {'m24': 'spc216.gro', 'spc': 'spc216.g**
For some water models we cannot derive the filename for the equilibrated box so we supply them explicitly.

mdpow.forcefields.**GROMACS_WATER_MODELS = {'m24': <M24 water: identifier=m24, ff=OPLS-AA>,**
Dictionary of *GromacsSolventModel* instances, one for each Gromacs water model available under the force field directory. The keys are the water model identifiers. For OPLS-AA the following ones are available.

mdpow.forcefields.**GROMACS_SOLVENT_MODELS = {'AMBER': {'cyclohexane': <CYCLOHEXANE water:**
Solvents available in GROMACS; the keys of the dictionary are the forcefields.

mdpow.forcefields.**ALL_FORCEFIELDS = {'AMBER': AMBER, 'CHARMM': CHARMM, 'OPLS-AA': OPLS-AA}**
The builtin forcefields' names and the corresponding *Forcefield* instance

5.9.3 Internal classes and functions

class mdpow.forcefields.**GromacsSolventModel**(*identifier: str, name: Optional[str] = None, itp: Union[str, os.PathLike, None] = None, coordinates: Union[str, os.PathLike, None] = None, description: Optional[str] = None, forcefield: str = 'OPLS-AA'*)
Data for a solvent model in Gromacs.

guess_filename(*extension*)
Guess the filename for the model and add *extension*

class mdpow.forcefields.**Forcefield**(*name: str, solvent_models: Dict[str, mdpow.forcefields.GromacsSolventModel], forcefield_dir: pathlib.Path, ions_itp: pathlib.Path, default_water_itp: pathlib.Path, default_water_model: str = 'tip4p', water_models: Optional[Dict[str, mdpow.forcefields.GromacsSolventModel]] = None*)
Contains information about files corresponding to a forcefield.

New in version 0.9.0.

ff_paths
Get the path to the forcefield directory and the ITP files for the ions and default water model.

mdpow.forcefields.**get_water_model**(*watermodel='tip4p'*)
Return a *GromacsSolventModel* corresponding to identifier *watermodel*

mdpow.forcefields.**get_solvent_identifier**(*solvent_type*, *model=None*, *forcefield: Union[mdpow.forcefields.Forcefield, str] = OPLS-AA*)
Get the identifier for a solvent model.

The identifier is needed to access a water model (i.e., a *GromacsSolventModel*) through *get_solvent_model()*. Because we have multiple water models but only limited other solvents, the organization of these models is a bit convoluted and it is best to obtain the desired water model in these two steps:

```
identifier = get_solvent_identifier("water", model="tip3p")
model = get_solvent_model(identifier)
```

For `solvent_type` *water*: either provide `None` or "water" for the specific `model` (and the default water model for the *Forcefield* will be selected, or a specific water model such as "tip3p" or "spce" (see *GROMACS_WATER_MODELS*). For other "octanol" or "wetoctanol" of OPLS-AA forcefield, the `model` is used to select a specific model. For other solvents and forcefields, "model" is not required.

Raises ValueError If there is no identifier found for the combination.

Returns An identifier

Changed in version 0.9.0: Raises `ValueError` instead of returning `None`.

mdpow.forcefields.**get_solvent_model**(*identifier*, *forcefield: Union[mdpow.forcefields.Forcefield, str] = OPLS-AA*)
Return a *GromacsSolventModel* corresponding to identifier *identifier*.

If identifier is "water" then the default water model for the *Forcefield* is assumed.

Changed in version 0.9.0: Function can now also accept a *Forcefield* for the `forcefield` argument.

Indices and tables

- genindex
- modindex
- search

Bibliography

[Jorge2010] M. Jorge, N.M. Garrido, A.J. Queimada, I.G. Economou, and E.A. Macedo. Effect of the integration method on the accuracy and computational efficiency of free energy calculations using thermodynamic integration. Journal of Chemical Theory and Computation, 6 (4):1018–1027, 2010. 10.1021/ct900661c.

[FrenkelSmit2002] D. Frenkel and B. Smit, Understanding Molecular Simulation. Academic Press, San Diego 2002

[Jorge2010] M. Jorge, N.M. Garrido, A.J. Queimada, I.G. Economou, and E.A. Macedo. Effect of the integration method on the accuracy and computational efficiency of free energy calculations using thermodynamic integration. Journal of Chemical Theory and Computation, 6 (4):1018–1027, 2010. 10.1021/ct900661c.

[FrenkelSmit2002] D. Frenkel and B. Smit, Understanding Molecular Simulation. Academic Press, San Diego 2002

[Jorge2010] M. Jorge, N.M. Garrido, A.J. Queimada, I.G. Economou, and E.A. Macedo. Effect of the integration method on the accuracy and computational efficiency of free energy calculations using thermodynamic integration. Journal of Chemical Theory and Computation, 6 (4):1018–1027, 2010. 10.1021/ct900661c.

[FrenkelSmit2002] D. Frenkel and B. Smit, Understanding Molecular Simulation. Academic Press, San Diego 2002

[Jorge2010] M. Jorge, N.M. Garrido, A.J. Queimada, I.G. Economou, and E.A. Macedo. Effect of the integration method on the accuracy and computational efficiency of free energy calculations using thermodynamic integration. Journal of Chemical Theory and Computation, 6 (4):1018–1027, 2010. 10.1021/ct900661c.

[FrenkelSmit2002] D. Frenkel and B. Smit, Understanding Molecular Simulation. Academic Press, San Diego 2002

[Jorge2010] M. Jorge, N.M. Garrido, A.J. Queimada, I.G. Economou, and E.A. Macedo. Effect of the integration method on the accuracy and computational efficiency of free energy calculations using thermodynamic integration. Journal of Chemical Theory and Computation, 6 (4):1018–1027, 2010. 10.1021/ct900661c.

[FrenkelSmit2002] D. Frenkel and B. Smit, Understanding Molecular Simulation. Academic Press, San Diego 2002

[Lescoulie2021] A. Lescoulie, "SPIDAL Summer REU 2021: Upgrading MDPOW and adding analysis functionality," Technical Report, Arizona State University, Tempe, AZ, 2021. doi: 10.6084/m9.figshare.17156018

[Jensen2006] K.P. Jensen and W.L. Jorgensen, *J Comp Theor Comput* **2** (2006), 1499. doi:10.1021/ct600252r

Python Module Index

m

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

W