alchemlyb Documentation

Release 2.2.0+0.geae82d8.dirty

Irfan Alibay, Bryce Allen, Mohammad S. Barhaghi, Oliver Beckstei

Apr 06, 2024

USER DOCUMENTATION

alchemlyb is a library for doing alchemical free energy calculations more easily.

It seeks to provide flexible building blocks covering functions for parsing data from formats common to existing MD engines, subsampling these data, fitting these data with an estimator to obtain free energies, and plotting the results.

These functions are simple in usage and pure in scope, and can be chained together to build customized analyses of data. General and robust workflows following best practices are also provided, which can be used as reference implementations and examples.

alchemlyb seeks to be as boring and simple as possible to enable more complex work. Its components allow work at all scales, from use on small systems using a single workstation to larger datasets that require distributed computing using libraries such as dask.

The library is *under active development*. However, it is used by multiple groups in a production environment. We use semantic versioning to indicate clearly what kind of changes you may expect between releases. Within any major release (1.x, 2.x, ...), the API is stable and is guaranteed to remain backwards-compatible.

Note: The **current 2.x release** of alchemlyb *only* supports pymbar releases **>= 4.0**. (Previous 1.x releases only support pymbar >= 3.0.5, <4.) See discussion #205 and issue #207.

See *Getting involved* for how to get in touch if you have questions or find problems.

CHAPTER

ONE

CORE PHILOSOPHY

With its goal to remain simple to use, **alchemlyb**'s design philosophy follows the following points:

1. Use functions when possible, classes only when necessary (or for estimators, see (2)).
2. For estimators, mimic the **scikit-learn** API as much as possible.
3. Aim for a consistent interface throughout, e.g. all parsers take similar inputs and yield a common set of outputs.

For more details, see the Roadmap.

CHAPTER

TWO

DEVELOPMENT MODEL

This is an open-source project, the hope of which is to produce a library with which the community is happy. To enable this, the library is a community effort. Development is done in the open on GitHub.

Software engineering best-practices are used throughout, including continuous integration testing via Github Actions, up-to-date documentation, and regular releases.

Note: With release 0.5.0, the alchemlyb project adopted NEP 29 to determine which versions of Python and NumPy will be supported. When we release a new major or minor version, alchemlyb will support *at least all minor versions of Python introduced and released in the prior 42 months from the release date with a minimum of 2 minor versions of Python*, and *all minor versions of NumPy released in the prior 24 months from the anticipated release date with a minimum of 3 minor versions of NumPy*.

The pandas package (one of our other primary dependencies) also follows NEP 29 so this support policy will make it easier to maintain **alchemlyb** in the future.

CHAPTER

THREE

GETTING INVOLVED

Contributions of all kinds are very welcome.

If you have questions or want to discuss alchemlyb please post in the alchemlyb Discussions.

If you have bug reports or feature requests then please get in touch with us through the Issue Tracker.

We also welcome code contributions: have a look at our Developer Guide. Open an issue with the proposed fix or change in the Issue Tracker and submit a pull request against the alchemistry/alchemlyb GitHub repository.

3.1 Installing alchemlyb

alchemlyb is available via the `pip` and `conda` package managers and can easily be installed with all its dependencies. Alternatively, it can also be directly installed from source

3.1.1 `conda` installation

The easiest way to keep track of all dependencies is to **install** *alchemlyb* as a conda package from the conda-forge (alchemlyb) channel

```
conda install -c conda-forge alchemlyb
```

You can later **update** your installation with

```
conda update -c conda-forge alchemlyb
```

3.1.2 `pip` installation

Install via `pip` from PyPi (alchemlyb)

```
pip install alchemlyb
```

Update with

```
pip install --update alchemlyb
```

3.1.3 Installing from source

To install from source, first clone the source code repository https://github.com/alchemistry/alchemlyb from GitHub with

```
git clone https://github.com/alchemistry/alchemlyb.git
```

and then install with `pip`

```
cd alchemlyb
pip install .
```

3.2 Parsing data files

alchemlyb features parsing submodules for getting raw data from different software packages into common data structures that can be used directly by its *subsamplers* and *estimators*. Each submodule features at least two functions, namely:

`extract_dHdl()`
: Extract the gradient of the Hamiltonian, $\frac{dH}{d\lambda}$, for each timestep of the sampled state. Required input for *TI-based estimators*.

`extract_u_nk()`
: Extract reduced potentials, u_{nk}, for each timestep of the sampled state and all neighboring states. Required input for *FEP-based estimators*.

`extract()`
: Extract both reduced potentials and the gradient of the Hamiltonian, u_{nk} and $\frac{dH}{d\lambda}$, in the form of a dictionary `'dHdl': Series, 'u_nk': DataFrame`. Required input for *FEP-based estimators* and *TI-based estimators*.

These functions have a consistent interface across all submodules, often taking a single file as input and any additional parameters required for giving either `dHdl` or `u_nk` in standard form.

3.2.1 Standard forms of raw data

All components of **alchemlyb** are designed to work together well with minimal work on the part of the user. To make this possible, the library deals in a common data structure for each `dHdl` and `u_nk`, and all parsers yield these quantities in these standard forms. The common data structure is a `pandas.DataFrame`. Normally, it should be sufficient to just pass the `dHdl` and `u_nk` dataframes from one `alchemlyb` function to the next. However, being a `DataFrame` provides enormous flexibility if the data need to be reorganized or transformed because of the powerful tools that `pandas` makes available to manipulate these data structures.

> **Warning:** When alchemlyb dataframes are transformed with standard pandas functions (such as `pandas.concat()`), care needs to be taken to ensure that `alchemlyb` metadata, which are stored in the dataframe, are maintained and propagated during processing of `alchemlyb` dataframes. See *metadata propagation* for how do work with dataframes safely in `alchemlyb`.

The metadata (such as the unit of the energy and temperature) are stored in `pandas.DataFrame.attrs`, a `dict`. Functions in `alchemlyb` are aware of these metadata but working with the data using `pandas` requires some care due to shortcomings in how pandas currently handles metadata (see issue pandas-dev/pandas#28283).

Serialisation

Alchemlyb data structures (`dHdl` and `u_nk`) can be serialized as dataframes and made persistent. We use the parquet format for serializing (writing) to a file and de-serializing (reading) from a parquet file.

For serialization we simply use the `pandas.DataFrame.to_parquet()` method of a `pandas.DataFrame`. For loading alchemlyb data we provide the *`alchemlyb.parsing.parquet.extract_dHdl()`* and *`alchemlyb.parsing.parquet.extract_u_nk()`* functions as shown in the example:

```
from alchemlyb.parsing.parquet import extract_dHdl, extract_u_nk
import pandas as pd

u_nk.to_parquet(path='u_nk.parquet', index=True)
dHdl.to_parquet(path='dHdl.parquet', index=True)

new_u_nk = extract_u_nk('u_nk.parquet', T=300)
new_dHdl = extract_dHdl('dHdl.parquet', T=300)
```

Note: Serialization of `pandas.DataFrame` to *parquet* file is only allowed for *pandas>=2*, whereas the deserialization is permitted for any pandas version.

`dHdl` standard form

All parsers yielding `dHdl` gradients return this as a `pandas.DataFrame` with the following structure:

```
                                    coul        vdw
time  coul-lambda vdw-lambda
  0.0 0.0         0.0          10.264125  -0.522539
  1.0 0.0         0.0           9.214077  -2.492852
  2.0 0.0         0.0          -8.527066  -0.405814
  3.0 0.0         0.0          11.544028  -0.358754
..... ...         ...          .........  .........
 97.0 1.0         1.0         -10.681702 -18.603644
 98.0 1.0         1.0          29.518990  -4.955664
 99 0 1.0         1.0          -3.833667  -0.836967
100.0 1.0         1.0         -12.835707   0.786278
```

This is a multi-index DataFrame, giving `time` for each sample as the outermost index, and the value of each λ from which the sample came as subsequent indexes. The columns of the DataFrame give the value of $\frac{dH}{d\lambda}$ with respect to each of these separate λ parameters.

For datasets that sample with only a single λ parameter, then the DataFrame will feature only a single column perhaps like:

```
                        fep
time  fep-lambda
  0.0 0.0         10.264125
  1.0 0.0          9.214077
  2.0 0.0         -8.527066
  3.0 0.0         11.544028
..... ...         .........
 97.0 1.0        -10.681702
```

(continues on next page)

(continued from previous page)

```
 98.0 1.0         29.518990
 99 0 1.0         -3.833667
100.0 1.0        -12.835707
```

u_nk standard form

All parsers yielding u_nk reduced potentials return this as a pandas.DataFrame with the following structure:

```
                                (0.0, 0.0) (0.25, 0.0) (0.5, 0.0)  ...  (1.0, 1.0)
time  coul-lambda vdw-lambda
  0.0 0.0         0.0          -22144.50   -22144.24  -22143.98        -21984.81
  1.0 0.0         0.0          -21985.24   -21985.10  -21984.96        -22124.26
  2.0 0.0         0.0          -22124.58   -22124.47  -22124.37        -22230.61
  3.0 1.0         0.1          -22230.65   -22230.63  -22230.62        -22083.04
..... ...         ...          .........   .........  .........  ...   .........
 97.0 1.0         1.0          -22082.29   -22082.54  -22082.79        -22017.42
 98.0 1.0         1.0          -22087.57   -22087.76  -22087.94        -22135.15
 99.0 1.0         1.0          -22016.69   -22016.93  -22017.17        -22057.68
100.0 1.0         1.0          -22137.19   -22136.51  -22135.83        -22101.26
```

This is a multi-index DataFrame, giving `time` for each sample as the outermost index, and the value of each λ from which the sample came as subsequent indexes. The columns of the DataFrame give the value of u_{nk} for each set of λ parameters values were recorded for. Column labels are the values of the λ parameters as a tuple in the same order as they appear in the multi-index.

For datasets that sample only a single λ parameter, then the DataFrame will feature only a single index in addition to `time`, with the values of λ for which reduced potentials were recorded given as column labels:

```
                       0.0        0.25        0.5  ...         1.0
time  fep-lambda
  0.0 0.0         -22144.50   -22144.24  -22143.98        -21984.81
  1.0 0.0         -21985.24   -21985.10  -21984.96        -22124.26
  2.0 0.0         -22124.58   -22124.47  -22124.37        -22230.61
  3.0 1.0         -22230.65   -22230.63  -22230.62        -22083.04
..... ...         .........   .........  .........  ...   .........
 97.0 1.0         -22082.29   -22082.54  -22082.79        -22017.42
 98.0 1.0         -22087.57   -22087.76  -22087.94        -22135.15
 99.0 1.0         -22016.69   -22016.93  -22017.17        -22057.68
100.0 1.0         -22137.19   -22136.51  -22135.83        -22101.26
```

A note on units

alchemlyb reads input files in native energy units and converts them to a common unit, the energy measured in k_BT, where k_B is Boltzmann's constant and T is the thermodynamic absolute temperature in Kelvin. Therefore, all parsers require specification of T.

Throughout alchemlyb, the metadata, such as the energy unit and temperature of the dataset, are stored as a dictionary in pandas.DataFrame.attrs metadata attribute. The keys of the attrs dictionary are

"temperature"
: the temperature at which the simulation was performed, in Kelvin

"energy_unit"
: the unit of energy, such as "kT", "kcal/mol", "kJ/mol" (as defined in *units*)

Conversion functions in `alchemlyb.postprocessing` and elsewhere may use the metadata for unit conversion and other transformations.

As the following example shows, after parsing of data files, the energy unit is "kT", i.e., the $\partial H/\partial\lambda$ timeseries is measured in multiples of k_BT per lambda step:

```
>>> from alchemtest.gmx import load_benzene
>>> from alchemlyb.parsing.gmx import extract_dHdl
>>> dataset = load_benzene()
>>> dhdl = extract_dHdl(dataset['data']['Coulomb'][0], 310)
>>> dhdl.attrs['temperature']
310
>>> dhdl.attrs['energy_unit']
'kT'
```

Also, although parsers will extract timestamps from input data, these are taken as-is and the library does not have any awareness of units for these. Keep this in mind when doing, e.g. *subsampling*.

Metadata Propagation

The metadata is stored in `pandas.DataFrame.attrs`. Though common pandas functions can safely propagate the metadata, the metadata might get lost during some operations such as concatenation (pandas-dev/pandas#28283). *`alchemlyb.concat()`* is provided to replace `pandas.concat()` allowing the safe propagation of metadata.

```
>>> import alchemlyb
>>> from alchemtest.gmx import load_benzene
>>> from alchemlyb.parsing.gmx import extract_dHdl
>>> dataset = load_benzene().data
>>> dhdl_coul = alchemlyb.concat([extract_dHdl(xvg, T=300) for xvg in dataset['Coulomb
↪']])
>>> dhdl_coul.attrs
{'temperature': 300, 'energy_unit': 'kT'}
```

`alchemlyb.`**`concat`**(*objs*, **args*, ***kwargs*)

Concatenate pandas objects while persevering the attrs.

Concatenate pandas objects along a particular axis with optional set logic along the other axes. If all pandas objects have the same attrs attribute, the new pandas objects would have this attrs attribute. A ValueError would be raised if any pandas object has a different attrs.

Parameters
: **`objs`** – A sequence or mapping of Series or DataFrame objects.

Returns
: Concatenated pandas object.

Return type
: DataFrame

Raises
: **`ValueError`** – If not all pandas objects have the same attrs.

See also:

`pandas.concat`

New in version 0.5.0.

Changed in version 1.0.1: When input is single dataframe, it will be sent out directly.

Although all functions in **alchemlyb** will safely propagate the metadata, if the user is interested in writing custom data manipulation functions, a decorator *alchemlyb.pass_attrs()* could be used to pass the metadata from the input data frame (first positional argument) to the output dataframe to ensure safe propagation of metadata.

```
>>> from alchemlyb import pass_attrs
>>> @pass_attrs
>>> def manipulation(dataframes, *args, **kwargs):
>>>     return func(dataframes, *args, **kwargs)
```

alchemlyb.**pass_attrs**(*func*)

Pass the attrs from the first positional argument to the output dataframe.

New in version 0.5.0.

3.2.2 Parsers by software package

alchemlyb tries to provide parser functions for as many simulation packages as possible. See the documentation for the package you are using for more details on parser usage, including the assumptions parsers make and suggestions for how output data should be structured for ease of use:

gmx	Parsers for extracting alchemical data from Gromacs output files.
amber	Parsers for extracting alchemical data from AMBER output files.
namd	Parsers for extracting alchemical data from NAMD output files.
gomc	Parsers for extracting alchemical data from GOMC output files.
parquet	

Gromacs parsing

Parsers for extracting alchemical data from Gromacs output files.

The parsers featured in this module are constructed to properly parse XVG files containing Hamiltonian differences (for obtaining reduced potentials, u_{nk}) and/or Hamiltonian derivatives (for obtaining gradients, $\frac{dH}{d\lambda}$). To produce such a file from an existing EDR energy file, use `gmx energy -f <.edr> -odh dhdl.xvg` with your installation of Gromacs.

If you wish to use FEP-based estimators such as *MBAR* that require reduced potentials for all lambda states in the alchemical leg, you will need to use these MDP options:

```
calc-lambda-neighbors = -1     ; calculate Delta H values for all other lambda windows
dhdl-print-energy = potential  ; total potential energy of system included
```

In addition, the full set of lambda states for the alchemical leg should be explicitly specified in the `fep-lambdas` option (or `coul-lambdas`, `vdw-lambdas`, etc.), since this is what Gromacs uses to determine what lambda values to calculate ΔH values for.

To use TI-based estimators that require gradients, you will need to include these options:

```
dhdl-derivatives = yes          ; write derivatives of Hamiltonian with respect to lambda
```

Additionally, the parsers can properly parse XVG files (containing Hamiltonian differences and/or Hamiltonian derivatives) produced during expanded ensemble simulations. To produce such a file during the simulation, use `gmx mdrun -deffnm <name> -dhdl dhdl.xvg` with your installation of Gromacs. To run an expanded ensemble simulation you will need to use the following MDP option:

```
free_energy = expanded        ; turns on expanded ensemble simulation, lambda state␣
↪becomes a dynamic variable
```

API Reference

This submodule includes these parsing functions:

alchemlyb.parsing.gmx.**extract_dHdl**(*xvg*, *T*, *filter=True*)

Return gradients *dH/dl* from a Hamiltonian differences XVG file.

Parameters

- **xvg** (*str*) – Path to XVG file to extract data from.
- **T** (*float*) – Temperature in Kelvin the simulations sampled.
- **filter** (*bool*) – Filter out the lines that cannot be parsed. Such as rows with incorrect number of Columns and incorrectly formatted numbers (e.g. 123.45.67, nan or -).

Returns

dH/dl – dH/dl as a function of time for this lambda window.

Return type

Series

Note: Previous versions of alchemlyb (<0.5.0) used the GROMACS value of the molar gas constant of $R = 8.3144621 \times 10^3\,\mathrm{kJ \cdot mol^{-1} \cdot K^{-1}}$ instead of the scipy value `scipy.constants.R` in `scipy.constants` (see *alchemlyb.postprocessors.units*). The relative difference between the two values is 6×10^{-8}.

Therefore, results in kT for GROMACS data will differ between alchemlyb 0.5.0 and previous versions; the relative difference is on the order of 10^{-7} for typical cases.

Changed in version 0.5.0: The `scipy.constants` is used for parsers instead of the constants used by the corresponding MD engine. This leads to slightly different results for GROMACS input compared to previous versions of alchemlyb.

Changed in version 0.7.0: The keyword filter is implemented to ignore the line that cannot be parsed and is turned on by default.

alchemlyb.parsing.gmx.**extract_u_nk**(*xvg*, *T*, *filter=True*)

Return reduced potentials *u_nk* from a Hamiltonian differences XVG file.

Parameters

- **xvg** (*str*) – Path to XVG file to extract data from.
- **T** (*float*) – Temperature in Kelvin the simulations sampled.
- **filter** (*bool*) – Filter out the lines that cannot be parsed. Such as rows with incorrect number of Columns and incorrectly formatted numbers (e.g. 123.45.67, nan or -).

Returns
u_nk – Potential energy for each alchemical state (k) for each frame (n).

Return type
DataFrame

Note: Previous versions of alchemlyb (<0.5.0) used the GROMACS value of the molar gas constant of $R = 8.3144621 \times 10^3\,\mathrm{kJ \cdot mol^{-1} \cdot K^{-1}}$ instead of the scipy value `scipy.constants.R` in `scipy.constants` (see *alchemlyb.postprocessors.units*). The relative difference between the two values is 6×10^{-8}.

Therefore, results in kT for GROMACS data will differ between alchemlyb 0.5.0 and previous versions; the relative difference is on the order of 10^{-7} for typical cases.

Changed in version 0.5.0: The `scipy.constants` is used for parsers instead of the constants used by the corresponding MD engine. This leads to slightly different results for GROMACS input compared to previous versions of alchemlyb.

Changed in version 0.7.0: The keyword filter is implemented to ignore the line that cannot be parsed and is turned on by default.

`alchemlyb.parsing.gmx.`**`extract`**(*xvg*, *T*, *filter=True*)

Return reduced potentials *u_nk* and gradients *dH/dl* from a Hamiltonian differences XVG file.

Parameters

- **xvg** (*str*) – Path to XVG file to extract data from.
- **T** (*float*) – Temperature in Kelvin the simulations sampled.
- **filter** (*bool*) – Filter out the lines that cannot be parsed. Such as rows with incorrect number of Columns and incorrectly formatted numbers (e.g. 123.45.67, nan or -).

Returns
A dictionary with keys of 'u_nk', which is a pandas DataFrame of potential energy for each alchemical state (k) for each frame (n), and 'dHdl', which is a Series of dH/dl as a function of time for this lambda window.

Return type
Dict

New in version 1.0.0.

Amber parsing

Parsers for extracting alchemical data from AMBER output files.

Some of the file parsing parts are adapted from alchemical-analysis.

Changed in version 1.0.0: Now raises `ValueError` when an invalid file is given to the parser. Now raises `ValueError` when inconsistency in MBAR states/data is found.

The parsers featured in this module are constructed to properly parse Amber MD output files containing derivatives of the Hamiltonian and FEP (BAR/MBAR) data.

API Reference

This submodule includes these parsing functions:

alchemlyb.parsing.amber.**extract_dHdl**(*outfile*, *T*)

Return gradients *dH/dl* from AMBER TI outputfile.

Parameters

- **outfile** (*str*) – Path to AMBER .out file to extract data from.
- **T** (*float*) – Temperature in Kelvin at which the simulations were performed

Returns
dH/dl – dH/dl as a function of time for this lambda window.

Return type
Series

Changed in version 0.5.0: The scipy.constants is used for parsers instead of the constants used by the corresponding MD engine.

alchemlyb.parsing.amber.**extract_u_nk**(*outfile*, *T*)

Return reduced potentials *u_nk* from AMBER outputfile.

Parameters

- **outfile** (*str*) – Path to AMBER .out file to extract data from.
- **T** (*float*) – Temperature in Kelvin at which the simulations were performed; needed to generated the reduced potential (in units of kT)

Returns
u_nk – Reduced potential for each alchemical state (k) for each frame (n).

Return type
DataFrame

Changed in version 0.5.0: The scipy.constants is used for parsers instead of the constants used by the corresponding MD engine.

alchemlyb.parsing.amber.**extract**(*outfile*, *T*)

Return reduced potentials *u_nk* and gradients *dH/dl* from AMBER outputfile.

Parameters

- **outfile** (*str*) – Path to AMBER .out file to extract data from.
- **T** (*float*) – Temperature in Kelvin at which the simulations were performed; needed to generated the reduced potential (in units of kT)

Returns
A dictionary with keys of 'u_nk', which is a pandas DataFrame of reduced potentials for each alchemical state (k) for each frame (n), and 'dHdl', which is a Series of dH/dl as a function of time for this lambda window.

Return type
Dict

New in version 1.0.0.

NAMD parsing

Parsers for extracting alchemical data from NAMD output files.

The parsers featured in this module are constructed to properly parse NAMD `.fepout` output files containing derivatives of the Hamiltonian and FEP (BAR) data. See the NAMD documentation for the theoretical backdrop and implementation details.

If you wish to use BAR on FEP data, be sure to provide the `.fepout` file from both the forward and reverse transformations.

After calling *extract_u_nk()* on the forward and reverse work values, these dataframes can be combined into one:

```
# replace zeroes in initial dataframe with nan
u_nk_fwd.replace(0, np.nan, inplace=True)
# replace the nan values with the reverse dataframe --
# this should not overwrite any of the fwd work values
u_nk_fwd[u_nk_fwd.isnull()] = u_nk_rev
# replace remaining nan values back to zero
u_nk_fwd.replace(np.nan, 0, inplace=True)
# sort final dataframe by `fep-lambda` (as opposed to `timestep`)
u_nk = u_nk_fwd.sort_index(level=u_nk_fwd.index.names[1:])
```

The `fep-lambda` index states at which lambda this particular frame was sampled, whereas the columns are the evaluations of the Hamiltonian (or the potential energy U) at other lambdas (sometimes called "foreign lambdas").

API Reference

This submodule includes these parsing functions:

`alchemlyb.parsing.namd.`**`extract_u_nk`**(*fep_files*, *T*)

Return reduced potentials *u_nk* from NAMD fepout file(s).

Parameters

- **fep_file** (*str or list of str*) – Path to fepout file(s) to extract data from. These are sorted by filename, not including the path, prior to processing, using natural-sort. This way, filenames including numbers without leading zeros are handled intuitively.

  Windows may be split across files, or more than one window may be present in a given file. Windows without footer lines (which may be in a different file than the respective header lines) will raise an error. This means that while windows may have been interrupted and restarted, they must be complete. Lambda values are expected to increase or decrease monotonically, and match between header and footer of each window.

- **T** (*float*) – Temperature in Kelvin at which the simulation was sampled.

Returns

u_nk – Potential energy for each alchemical state (k) for each frame (n).

Return type

DataFrame

Note: If the number of forward and backward samples in a given window are different, the extra sample(s) will be discarded. This is typically zero or one sample.

Changed in version 0.5.0: The `scipy.constants` is used for parsers instead of the constants used by the corresponding MD engine.

Changed in version 0.6.0: Support for Interleaved Double-Wide Sampling files added, with various robustness checks.

fep_files can now be a list of filenames.

`alchemlyb.parsing.namd.extract`(*fep_files*, *T*)

Return reduced potentials *u_nk* from NAMD fepout file(s).

Parameters

- **fep_file** (*str or list of str*) – Path to fepout file(s) to extract data from. These are sorted by filename, not including the path, prior to processing, using natural-sort. This way, filenames including numbers without leading zeros are handled intuitively.

  Windows may be split across files, or more than one window may be present in a given file. Windows without footer lines (which may be in a different file than the respective header lines) will raise an error. This means that while windows may have been interrupted and restarted, they must be complete. Lambda values are expected to increase or decrease monotonically, and match between header and footer of each window.

- **T** (*float*) – Temperature in Kelvin at which the simulation was sampled.

Returns
A dictionary with keys of 'u_nk', which is a pandas DataFrame of potential energy for each alchemical state (k) for each frame (n).

Return type
Dict

Note: If the number of forward and backward samples in a given window are different, the extra sample(s) will be discarded. This is typically zero or one sample.

New in version 1.0.0.

GOMC parsing

Parsers for extracting alchemical data from GOMC output files.

The parsers featured in this module are constructed to properly parse GOMC free energy output files, containing the Hamiltonian derivatives ($\frac{dH}{d\lambda}$) for TI-based estimators and Hamiltonian differences (ΔH for all lambda states in the alchemical leg) for FEP-based estimators (BAR/MBAR).

API Reference

This submodule includes these parsing functions:

`alchemlyb.parsing.gomc.extract_dHdl`(*filename*, *T*)

Return gradients *dH/dl* from a Hamiltonian differences free energy file.

Parameters

- **filename** (*str*) – Path to free energy file to extract data from.
- **T** (*float*) – Temperature in Kelvin at which the simulation was sampled.

Returns
dH/dl – dH/dl as a function of step for this lambda window.

Return type
Series

Changed in version 0.5.0: The `scipy.constants` is used for parsers instead of the constants used by the corresponding MD engine.

`alchemlyb.parsing.gomc.extract_u_nk(filename, T)`

Return reduced potentials *u_nk* from a Hamiltonian differences dat file.

Parameters

- **filename** (*str*) – Path to free energy file to extract data from.
- **T** (*float*) – Temperature in Kelvin at which the simulation was sampled.

Returns
u_nk – Potential energy for each alchemical state (k) for each frame (n).

Return type
DataFrame

Changed in version 0.5.0: The `scipy.constants` is used for parsers instead of the constants used by the corresponding MD engine.

`alchemlyb.parsing.gomc.extract(filename, T)`

Return reduced potentials *u_nk* and gradients *dH/dl* from a Hamiltonian differences free energy file.

Parameters

- **xvg** (*str*) – Path to free energy file to extract data from.
- **T** (*float*) – Temperature in Kelvin the simulations sampled.
- **filter** (*bool*) – Filter out the lines that cannot be parsed. Such as rows with incorrect number of Columns and incorrectly formatted numbers (e.g. 123.45.67, nan or -).

Returns
A dictionary with keys of 'u_nk', which is a pandas DataFrame of potential energy for each alchemical state (k) for each frame (n), and 'dHdl', which is a Series of dH/dl as a function of time for this lambda window.

Return type
Dict

New in version 1.0.0.

API Reference

This submodule includes these parsing functions:

`alchemlyb.parsing.parquet.extract_u_nk(path, T)`

Return reduced potentials *u_nk* (unit: kT) from a pandas parquet file.

The parquet file should be serialised from the dataframe output from any parser with command (`u_nk_df.to_parquet(path=path, index=True)`).

Parameters

- **path** (*str*) – Path to parquet file to extract dataframe from.
- **T** (*float*) – Temperature in Kelvin of the simulations.

Returns
u_nk – Potential energy for each alchemical state (k) for each frame (n).

Return type
DataFrame

Note: pyarraw serializers would handle the float or string column name fine but will convert multi-lambda column name from *(0.0, 0.0)* to *"('0.0', '0.0')"*. This parser will restore the correct column name. Also parquet serialisation doesn't preserve the `pandas.DataFrame.attrs`. So the temperature is assigned in this function.

New in version 2.1.0.

`alchemlyb.parsing.parquet.`**`extract_dHdl`**`(`*path*, *T*`)`

Return gradients *dH/dl* (unit: kT) from a pandas parquet file.

The parquet file should be serialised from the dataframe output from any parser with command (*dHdl_df.to_parquet(path=path, index=True)*).

Parameters

- **path** (*str*) – Path to parquet file to extract dataframe from.
- **T** (*float*) – Temperature in Kelvin the simulations sampled.

Returns
dH/dl – dH/dl as a function of time for this lambda window.

Return type
DataFrame

Note: Parquet serialisation doesn't preserve the `pandas.DataFrame.attrs`. So the temperature is assigned in this function.

New in version 2.1.0.

3.3 Preprocessing datasets

It is often the case that some initial pre-processing of raw datasets are desirable before feeding these to an estimator. **alchemlyb** features some commonly-used pre-processing tools as a convenience. These are featured in the following submodules:

subsampling	Functions for subsampling datasets.

3.3.1 Subsampling

Functions for subsampling datasets.

The functions featured in this module can be used to easily subsample either *dHdl* or *u_nk* datasets to give less correlated timeseries.

High-level functions

Two high-level functions *decorrelate_u_nk()* and *decorrelate_dhdl()* can be used to preprocess the *dHdl* or *u_nk* in an automatic fashion. The following code removes an initial "burnin" period and decorrelates the data.

```
>>> from alchemlyb.parsing.gmx import extract_u_nk, extract_dHdl
>>> from alchemlyb.preprocessing.subsampling import (decorrelate_u_nk,
>>>     decorrelate_dhdl)
>>> bz = load_benzene().data
>>> u_nk = extract_u_nk(bz['Coulomb'], T=300)
>>> decorrelated_u_nk = decorrelate_u_nk(u_nk, method='dhdl',
>>>     remove_burnin=True)
>>> dhdl = extract_dHdl(bz['Coulomb'], T=300)
>>> decorrelated_dhdl = decorrelate_dhdl(dhdl, remove_burnin=True)
```

Low-level functions

To decorrelate the data, in addition to the dataframe that contains the *dHdl* or *u_nk*, a `pandas.Series` is needed for the autocorrection analysis. The series could be generated with *u_nk2series()* or *dhdl2series()* and feed into *statistical_inefficiency()* or *equilibrium_detection()*.

```
>>> from alchemlyb.parsing.gmx import extract_u_nk, extract_dHdl
>>> from alchemlyb.preprocessing.subsampling import (u_nk2series,
>>>     dhdl2series, statistical_inefficiency, equilibrium_detection)
>>> bz = load_benzene().data
>>> u_nk = extract_u_nk(bz['Coulomb'], T=300)
>>> u_nk_series = u_nk2series(u_nk, method='dE')
>>> decorrelate_u_nk = statistical_inefficiency(u_nk, series=u_nk_series)
>>> decorrelate_u_nk = equilibrium_detection(u_nk, series=u_nk_series)
>>> dhdl = extract_dHdl(bz['Coulomb'], T=300)
>>> dhdl_series = dhdl2series(dhdl)
>>> decorrelate_dhdl = statistical_inefficiency(dhdl, series=dhdl_series)
>>> decorrelate_dhdl = equilibrium_detection(dhdl, series=dhdl_series)
```

API Reference

alchemlyb.preprocessing.subsampling.**decorrelate_u_nk**(*df*, *method='dE'*, *drop_duplicates=True*, *sort=True*, *remove_burnin=False*, ***kwargs*)

Subsample an u_nk DataFrame based on the selected method.

The method can be either 'all' (obtained as a sum over all energy components) or 'dE'. In the latter case the energy differences $dE_{i,i+1}$ ($dE_{i,i-1}$ for the last lambda) are used. This is a wrapper function around the function *statistical_inefficiency()* or *equilibrium_detection()*.

Parameters

- **df** (*DataFrame*) – DataFrame to be subsampled according to the selected method.
- **method** (*{'all', 'dE'}*) – Method for decorrelating the data.
- **drop_duplicates** (*bool*) – Drop the duplicated lines based on time.
- **sort** (*bool*) – Sort the Dataframe based on the time column.
- **remove_burnin** (*bool*) – Whether to perform equilibrium detection (`True`) or just do statistical inefficiency (`False`).

  New in version 1.0.0.
- ****kwargs** – Additional keyword arguments for *statistical_inefficiency()* or *equilibrium_detection()*.

Returns
df subsampled according to selected *method*.

Return type
DataFrame

Note: The default of `True` for *drop_duplicates* and *sort* should result in robust decorrelation but can lose data.

New in version 0.6.0.

Changed in version 1.0.0: Add the *remove_burnin* keyword to allow unequilibrated frames to be removed. Rename *method* value 'dhdl_all' to 'all' and deprecate the 'dhdl'.

alchemlyb.preprocessing.subsampling.**decorrelate_dhdl**(*df*, *drop_duplicates=True*, *sort=True*, *remove_burnin=False*, ***kwargs*)

Subsample a dhdl DataFrame. This is a wrapper function around the function *statistical_inefficiency()* and *equilibrium_detection()*.

Parameters

- **df** (*DataFrame*) – DataFrame to subsample according to the selected method.
- **drop_duplicates** (*bool*) – Drop the duplicated lines based on time.
- **sort** (*bool*) – Sort the Dataframe based on the time column.
- **remove_burnin** (*bool*) – Whether to perform equilibrium detection (`True`) or just do statistical inefficiency (`False`).

  New in version 1.0.0.
- ****kwargs** – Additional keyword arguments for *statistical_inefficiency()* or *equilibrium_detection()*.

Returns
df subsampled.

Return type
DataFrame

Note: The default of `True` for *drop_duplicates* and *sort* should result in robust decorrelation but can loose data.

New in version 0.6.0.

Changed in version 1.0.0: Add the *remove_burnin* keyword to allow unequilibrated frames to be removed.

alchemlyb.preprocessing.subsampling.**u_nk2series**(*df*, *method='dE'*)

Convert an u_nk DataFrame into a series based on the selected method for subsampling.

The method can be either 'all' (obtained as a sum over all energy components) or 'dE'. In the latter case the energy differences $dE_{i,i+1}$ ($dE_{i,i-1}$ for the last lambda) are used.

Parameters

- **df** (*DataFrame*) – DataFrame to be converted according to the selected method.
- **method** (*{'all', 'dE'}*) – Method for converting the data.

Returns
series to be used as input for *statistical_inefficiency()* or *equilibrium_detection()*.

Return type
Series

New in version 1.0.0.

Changed in version 2.0.1: The *dE* method computes the difference between the current lambda and the next lambda (previous lambda for the last window), instead of using the next lambda or the previous lambda for the last window.

alchemlyb.preprocessing.subsampling.**dhdl2series**(*df*, *method='all'*)

Convert a dhdl DataFrame to a series for subsampling.

The series is generated by summing over all energy components (axis 1 of *df*), as for `method='all'` in *u_nk2series()*. Commonly, *df* only contains a single energy component but in some cases (such as using a split protocol in GROMACS), it can contain multiple columns for different energy terms.

Parameters

- **df** (*DataFrame*) – DataFrame to subsample according to the selected method.
- **method** (*'all'*) – Only 'all' is available; the keyword is provided for compatibility with *u_nk2series()*.

Returns
series to be used as input for *statistical_inefficiency()* or *equilibrium_detection()*.

Return type
Series

New in version 1.0.0.

alchemlyb.preprocessing.subsampling.**slicing**(*df*, *lower=None*, *upper=None*, *step=None*, *force=False*)

Subsample a DataFrame using simple slicing.

Parameters

- **df** (*DataFrame*) – DataFrame to subsample.
- **lower** (*float*) – Lower time to slice from.
- **upper** (*float*) – Upper time to slice to (inclusive).
- **step** (*int*) – Step between rows to slice by.
- **force** (*bool*) – Ignore checks that DataFrame is in proper form for expected behavior.

Returns
df subsampled.

Return type
DataFrame

Changed in version 1.0.1: The rows with NaN values are not dropped by default.

alchemlyb.preprocessing.subsampling.**statistical_inefficiency**(*df*, *series=None*, *lower=None*, *upper=None*, *step=None*, *conservative=True*, *drop_duplicates=False*, *sort=False*)

Subsample a DataFrame based on the calculated statistical inefficiency of a timeseries.

If *series* is `None`, then this function will behave the same as *slicing()*.

Parameters

- **df** (*DataFrame*) – DataFrame to subsample according statistical inefficiency of *series*.
- **series** (*Series*) – Series to use for calculating statistical inefficiency. If `None`, no statistical inefficiency-based subsampling will be performed.
- **lower** (*float*) – Lower bound to pre-slice *series* data from.
- **upper** (*float*) – Upper bound to pre-slice *series* to (inclusive).
- **step** (*int*) – Step between *series* items to pre-slice by.
- **conservative** (*bool*) – `True` use `ceil(statistical_inefficiency)` to slice the data in uniform intervals (the default). `False` will sample at non-uniform intervals to closely match the (fractional) statistical_inefficieny, as implemented in `pymbar.timeseries.subsample_correlated_data()`.
- **drop_duplicates** (*bool*) – Drop the duplicated lines based on time.
- **sort** (*bool*) – Sort the Dataframe based on the time column.

Returns

df subsampled according to subsampled *series*.

Return type

DataFrame

Warning: The *series* and the data to be sliced, *df*, need to have the same number of elements because the statistical inefficiency is calculated based on the index of the series (and not an associated time). At the moment there is no automatic conversion from a time to an index.

Note: For a non-integer statistical ineffciency g, the default value `conservative=True` will provide _fewer_ data points than allowed by g and thus error estimates will be _higher_. For large numbers of data points and converged free energies, the choice should not make a difference. For small numbers of data points, `conservative=True` decreases a false sense of accuracy and is deemed the more careful and conservative approach.

See also:

`pymbar.timeseries.statistical_inefficiency`
detailed background

`pymbar.timeseries.subsample_correlated_data`
used for subsampling

Changed in version 0.2.0: The `conservative` keyword was added and the method is now using `pymbar.timeseries.statistical_inefficiency()`; previously, the statistical inefficiency was _rounded_ (instead of `ceil()`) and thus one could end up with correlated data.

Changed in version 1.0.0: Fixed a bug that would effectively ignore the `lower` and `step` keywords when returning the subsampled DataFrame object. See issue #198 for more details.

alchemlyb.preprocessing.subsampling.**equilibrium_detection**(*df*, *series=None*, *lower=None*, *upper=None*, *step=None*, *drop_duplicates=False*, *sort=False*)

Subsample a DataFrame using automated equilibrium detection on a timeseries.

This function uses the `pymbar` implementation of the *simple automated equilibrium detection* algorithm in [Chodera2016].

If *series* is `None`, then this function will behave the same as *slicing()*.

Parameters

- **df** (*DataFrame*) – DataFrame to subsample according to equilibrium detection on *series*.
- **series** (*Series*) – Series to detect equilibration on. If `None`, no equilibrium detection-based subsampling will be performed.
- **lower** (*float*) – Lower bound to pre-slice *series* data from.
- **upper** (*float*) – Upper bound to pre-slice *series* to (inclusive).
- **step** (*int*) – Step between *series* items to pre-slice by.
- **drop_duplicates** (*bool*) – Drop the duplicated lines based on time.
- **sort** (*bool*) – Sort the Dataframe based on the time column.

Returns
df subsampled according to subsampled *series*.

Return type
DataFrame

Notes

Please cite [Chodera2016] when you use this function in published work.

See also:

`pymbar.timeseries.detect_equilibration`
detailed background

`pymbar.timeseries.subsample_correlated_data`
used for subsampling

Changed in version 1.0.0: Add the drop_duplicates and sort keyword to unify the behaviour between *statistical_inefficiency()* or *equilibrium_detection()*.

3.4 Using estimators to obtain free energies

Calculating free energy differences from raw alchemical data requires the use of some *estimator*. All estimators in **alchemlyb** conform to a common design pattern, with a form similar to that of estimators found in scikit-learn. If you have familiarity with the usage of estimators in **scikit-learn**, then working with estimators in **alchemlyb** should be somewhat straightforward.

alchemlyb provides implementations of many commonly-used estimators, which come in two varieties: TI-based and FEP-based.

3.4.1 TI-based estimators

TI-based estimators such as *TI* take as input *dHdl* gradients for the calculation of free energy differences. All TI-based estimators integrate these gradients with respect to λ, differing only in *how* they numerically perform this integration.

As a usage example, we'll use *TI* to calculate the free energy of solvation of benzene in water. We'll use the benzene-in-water dataset from `alchemtest.gmx`:

```
>>> from alchemtest.gmx import load_benzene
>>> bz = load_benzene().data
```

and parse the datafiles separately for each alchemical leg using *alchemlyb.parsing.gmx.extract_dHdl()* to obtain *dHdl* gradients:

```
>>> from alchemlyb.parsing.gmx import extract_dHdl
>>> import pandas as pd
>>> import alchemlyb

>>> dHdl_coul = alchemlyb.concat([extract_dHdl(xvg, T=300) for xvg in bz['Coulomb']])
>>> dHdl_vdw = alchemlyb.concat([extract_dHdl(xvg, T=300) for xvg in bz['VDW']])
```

We can now use the *TI* estimator to obtain the free energy differences between each λ window sampled. The *fit()* method is used to perform the free energy estimate, given the gradient data:

```
>>> from alchemlyb.estimators import TI

>>> ti_coul = TI()
>>> ti_coul.fit(dHdl_coul)
TI(verbose=False)

# we could also just call the `fit` method
# directly, since it returns the `TI` object
>>> ti_vdw = TI().fit(dHdl_vdw)
```

The sum of the endpoint free energy differences will be the free energy of solvation for benzene in water. The free energy differences (in units of k_BT) between each λ window can be accessed via the `delta_f_` attribute:

```
>>> ti_coul.delta_f_
          0.00      0.25      0.50      0.75      1.00
0.00  0.000000  1.620328  2.573337  3.022170  3.089027
0.25 -1.620328  0.000000  0.953009  1.401842  1.468699
0.50 -2.573337 -0.953009  0.000000  0.448832  0.515690
0.75 -3.022170 -1.401842 -0.448832  0.000000  0.066857
1.00 -3.089027 -1.468699 -0.515690 -0.066857  0.000000
```

So we can get the endpoint differences (free energy difference between $\lambda = 0$ and $\lambda = 1$) of each with:

```
>>> ti_coul.delta_f_.loc[0.00, 1.00]
3.0890270218676896

>>> ti_vdw.delta_f_.loc[0.00, 1.00]
-3.0558175199846058
```

giving us a solvation free energy in units of $k_B T$ for benzene of:

```
>>> ti_coul.delta_f_.loc[0.00, 1.00] + ti_vdw.delta_f_.loc[0.00, 1.00]
0.033209501883083803
```

In addition to the free energy differences, we also have access to the errors on these differences via the `d_delta_f_` attribute:

```
>>> ti_coul.d_delta_f_
          0.00      0.25      0.50      0.75      1.00
0.00  0.000000  0.009706  0.013058  0.015038  0.016362
0.25  0.009706  0.000000  0.008736  0.011486  0.013172
0.50  0.013058  0.008736  0.000000  0.007458  0.009858
0.75  0.015038  0.011486  0.007458  0.000000  0.006447
1.00  0.016362  0.013172  0.009858  0.006447  0.000000
```

Estimate free energy with gaussian quadrature

If the simulations are performed at certain gaussian quadrature points, *TI_GQ* can be used to estimate the free energy. The usage of *TI_GQ* is similar to *TI*, but instead of the free energy differences between λ windows, the values in the `delta_f_` and `d_delta_f_` tables are cumulative addition of estimation from one λ window to another. To be consistent with *TI* and other estimators, the diagonal values are set to zeros and two end states at λ 0 and 1 are added, although the simulation may not be performed at λ 0 and 1. The value at λ 0 is set to zero and the value at λ 1 is the same as the previous gaussian quadrature point.

List of TI-based estimators

TI([verbose])	Thermodynamic integration (TI).
TI_GQ([verbose])	Thermodynamic integration (TI) with gaussian quadrature estimation.

TI

The *TI* estimator is a simple implementation of thermodynamic integration that uses the trapezoid rule for integrating the space between $\left\langle \frac{dH}{d\lambda} \right\rangle$ values for each λ sampled.

API Reference

class alchemlyb.estimators.**TI**(*verbose=False*)

Thermodynamic integration (TI).

Parameters
verbose (*bool, optional*) – Set to True if verbose debug output is desired.

delta_f_

The estimated dimensionless free energy difference between each state.

Type
DataFrame

d_delta_f_

The estimated statistical uncertainty (one standard deviation) in dimensionless free energy differences.

Type
DataFrame

states_

Lambda states for which free energy differences were obtained.

Type
list

dhdl

The estimated dhdl of each state.

Type
DataFrame

Changed in version 1.0.0: *delta_f_*, *d_delta_f_*, *states_* are view of the original object.

fit(*dHdl*)

Compute free energy differences between each state by integrating dHdl across lambda values.

Parameters
dHdl (*DataFrame*) – dHdl[n,k] is the potential energy gradient with respect to lambda for each configuration n and lambda k.

separate_dhdl()

For transitions with multiple lambda, the attr:*dhdl* would return a `DataFrame` which gives the dHdl for all the lambda states, regardless of whether it is perturbed or not. This function creates a list of `pandas.Series` for each lambda, where each `pandas.Series` describes the potential energy gradient for the lambdas state that is perturbed.

Returns
dHdl_list – A list of `pandas.Series` such that `dHdl_list[k]` is the potential energy gradient with respect to lambda for each configuration that lambda k is perturbed.

Return type
list

get_metadata_routing()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns
routing – A `MetadataRequest` encapsulating routing information.

Return type
MetadataRequest

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters
deep (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns
params – Parameter names mapped to their values.

Return type
dict

`set_fit_request`(*, *dHdl: bool | None | str = '$UNCHANGED$'*) → *TI*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
dHdl (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `dHdl` parameter in `fit`.

Returns
self – The updated object.

Return type
object

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters
****params** (*dict*) – Estimator parameters.

Returns
self – Estimator instance.

Return type
estimator instance

TI_GQ

The *TI_GQ* estimator is an implementation of thermodynamic integration that uses the gaussian quadrature for integrating the space between $\left\langle \frac{dH}{d\lambda} \right\rangle$ values for each λ sampled. To use this method, please make sure that the simulations are performed at certain λ values using fixed gaussian quadrature points (e.g., [He2020]). Currently, up to 16 guassian quadrature points are supported (see the table below).

Table 1: gaussian quadrature points

number of λ	λ values
1	0.5
2	0.21132, 0.78867
3	0.1127, 0.5, 0.88729
4	0.06943, 0.33001, 0.66999, 0.93057
5	0.04691, 0.23076, 0.5, 0.76923, 0.95308
6	0.03377, 0.1694 , 0.38069, 0.61931, 0.8306 , 0.96623
7	0.02544, 0.12923, 0.29707, 0.5, 0.70292, 0.87076, 0.97455
8	0.01986, 0.10167, 0.23723, 0.40828, 0.59172, 0.76277, 0.89833, 0.98014
9	0.01592, 0.08198, 0.19331, 0.33787, 0.5, 0.66213, 0.80669, 0.91802, 0.98408
10	0.01305, 0.06747, 0.1603, 0.2833, 0.42556, 0.57444, 0.7167, 0.8397, 0.93253, 0.98695
11	0.01089, 0.05647, 0.13492, 0.24045, 0.36523, 0.5, 0.63477, 0.75955, 0.86508, 0.94353, 0.98911
12	0.00922, 0.04794, 0.11505, 0.20634, 0.31608, 0.43738, 0.56262, 0.68392, 0.79366, 0.88495, 0.95206, 0.99078
13	0.00791, 0.0412, 0.09921, 0.17883, 0.27575, 0.38477, 0.5, 0.61523, 0.72425, 0.82117, 0.90079, 0.9588, 0.99209
14	0.00686, 0.03578, 0.0864, 0.15635, 0.24238, 0.34044, 0.44597, 0.55403, 0.65956, 0.75762, 0.84365, 0.9136, 0.96422, 0.99314
15	0.006, 0.03136, 0.0759, 0.13779, 0.21451, 0.30292, 0.3994 , 0.5, 0.6006, 0.69708, 0.78549, 0.86221, 0.9241, 0.96864, 0.994
16	0.0053, 0.02771, 0.06718, 0.1223, 0.19106, 0.27099, 0.3592, 0.45249, 0.54751, 0.6408, 0.72901, 0.80894, 0.8777, 0.93282, 0.97229, 0.9947

API Reference

class alchemlyb.estimators.**TI_GQ**(*verbose=False*)

Thermodynamic integration (TI) with gaussian quadrature estimation.

Parameters
verbose (*bool, optional*) – Set to True if verbose debug output is desired.

delta_f_

The estimated cumulative free energy from one state to another.

Type
DataFrame

d_delta_f_

The estimated statistical uncertainty (one standard deviation) in dimensionless cumulative free energies.

Type
DataFrame

states_

Lambda states for which free energy estimation were obtained.

Type
list

dhdl

The estimated dhdl of each state.

Type
DataFrame

New in version 2.1.0.

fit(*dHdl*)

Compute cumulative free energy from one state to another by integrating dHdl across lambda values.

Parameters
dHdl (*DataFrame*) – dHdl[n,k] is the potential energy gradient with respect to lambda for each configuration n and lambda k.

static separate_mean_variance(*means*, *variances*)

For transitions with multiple lambda, the attr:*dhdl* would return a `DataFrame` which gives the dHdl for all the lambda states, regardless of whether it is perturbed or not. This function creates 3 lists of `numpy.array`, `pandas.Series` and `pandas.Series` for each lambda, where the lists describe the lambda values, potential energy gradient and variance values for the lambdas state that is perturbed.

Parameters

- **means** (*DataFrame*) – means is the average potential energy gradient at each lambda.
- **variances** (*DataFrame*) – variances is variance of the potential energy gradient at each lambda.

Returns

- **lambda_list** (*list*) – A list of `numpy.array` such that `lambda_list[k]` is the lambda values with respect to each type of lambda.
- **dhdl_list** (*list*) – A list of `pandas.Series` such that `dHdl_list[k]` is the potential energy gradient with respect to lambda for each configuration that lambda k is perturbed.
- **variance_list** (*list*) – A list of `pandas.Series` such that `variance_list[k]` is the variance of the potential energy gradient with respect to lambda for each configuration that lambda k is perturbed.
- **index_list** (*list*) – A list of `float` or `tuple` such that each `float` or `tuple` is the index of the final *delta_f_* and *d_delta_f_*

get_metadata_routing()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns
routing – A `MetadataRequest` encapsulating routing information.

Return type
MetadataRequest

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters
deep (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns
params – Parameter names mapped to their values.

Return type
dict

`set_fit_request`(*, *dHdl: bool | None | str = '$UNCHANGED$'*) → *TI_GQ*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
dHdl (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `dHdl` parameter in `fit`.

Returns
self – The updated object.

Return type
object

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters
****params** (*dict*) – Estimator parameters.

Returns
self – Estimator instance.

Return type
estimator instance

3.4.2 FEP-based estimators

FEP-based estimators such as *MBAR* take as input *u_nk* reduced potentials for the calculation of free energy differences. All FEP-based estimators make use of the overlap between distributions of these values for each sampled λ, differing in *how* they use this overlap information to give their free energy difference estimate.

As a usage example, we'll use *MBAR* to calculate the free energy of solvation of benzene in water. We'll use the benzene-in-water dataset from `alchemtest.gmx`:

```
>>> from alchemtest.gmx import load_benzene
>>> bz = load_benzene().data
```

and parse the datafiles separately for each alchemical leg using *alchemlyb.parsing.gmx.extract_u_nk()* to obtain *u_nk* reduced potentials:

```
>>> from alchemlyb.parsing.gmx import extract_u_nk
>>> import pandas as pd

>>> u_nk_coul = alchemlyb.concat([extract_u_nk(xvg, T=300) for xvg in bz['Coulomb']])
>>> u_nk_vdw = alchemlyb.concat([extract_u_nk(xvg, T=300) for xvg in bz['VDW']])
```

We can now use the *MBAR* estimator to obtain the free energy differences between each λ window sampled. The *fit()* method is used to perform the free energy estimate, given the gradient data:

```
>>> from alchemlyb.estimators import MBAR

>>> mbar_coul = MBAR()
>>> mbar_coul.fit(u_nk_coul)
MBAR(initial_f_k=None, maximum_iterations=10000, method=({'method': 'hybr'},),
   relative_tolerance=1e-07, verbose=False)

# we could also just call the `fit` method
# directly, since it returns the `MBAR` object
>>> mbar_vdw = MBAR().fit(u_nk_vdw)
```

The sum of the endpoint free energy differences will be the free energy of solvation for benzene in water. The free energy differences (in units of k_BT) between each λ window can be accessed via the `delta_f_` attribute:

```
>>> mbar_coul.delta_f_
          0.00      0.25      0.50      0.75      1.00
0.00  0.000000  1.619069  2.557990  2.986302  3.041156
0.25 -1.619069  0.000000  0.938921  1.367232  1.422086
0.50 -2.557990 -0.938921  0.000000  0.428311  0.483165
0.75 -2.986302 -1.367232 -0.428311  0.000000  0.054854
1.00 -3.041156 -1.422086 -0.483165 -0.054854  0.000000
```

So we can get the endpoint differences (free energy difference between $\lambda = 0$ and $\lambda = 1$) of each with:

```
>>> mbar_coul.delta_f_.loc[0.00, 1.00]
3.0411558818767954

>>> mbar_vdw.delta_f_.loc[0.00, 1.00]
-3.0067874666136074
```

giving us a solvation free energy in units of k_BT for benzene of:

```
>>> mbar_coul.delta_f_.loc[0.00, 1.00] + mbar_vdw.delta_f_.loc[0.00, 1.00]
0.034368415263188012
```

In addition to the free energy differences, we also have access to the errors on these differences via the `d_delta_f_` attribute:

```
>>> mbar_coul.d_delta_f_
          0.00      0.25      0.50      0.75      1.00
0.00  0.000000  0.008802  0.014432  0.018097  0.020879
0.25  0.008802  0.000000  0.006642  0.011404  0.015143
0.50  0.014432  0.006642  0.000000  0.005362  0.009983
0.75  0.018097  0.011404  0.005362  0.000000  0.005133
1.00  0.020879  0.015143  0.009983  0.005133  0.000000
```

List of FEP-based estimators

MBAR([maximum_iterations, ...])	Multi-state Bennett acceptance ratio (MBAR).
BAR([maximum_iterations, ...])	Bennett acceptance ratio (BAR).

MBAR

The *MBAR* estimator is a light wrapper around the reference implementation of MBAR [Shirts2008] from `pymbar` (`pymbar.mbar.MBAR`). As a generalization of BAR, it uses information from all sampled states to generate an estimate for the free energy difference between each state.

API Reference

class alchemlyb.estimators.**MBAR**(*maximum_iterations=10000*, *relative_tolerance=1e-07*, *initial_f_k=None*, *method='robust'*, *n_bootstraps=0*, *verbose=False*)

Multi-state Bennett acceptance ratio (MBAR).

Parameters

- **maximum_iterations** (*int, optional*) – Set to limit the maximum number of iterations performed.
- **relative_tolerance** (*float, optional*) – Set to determine the relative tolerance convergence criteria.
- **initial_f_k** (*np.ndarray, float, shape=(K), optional*) – Set to the initial dimensionless free energies to use as a guess (default None, which sets all $f_k = 0$).

- **method** (*str, optional, default="robust"*) – The optimization routine to use. This can be any of the methods available via `scipy.optimize.minimize()` or `scipy.optimize.root()`.
- **n_bootstraps** (*int, optional*) – Whether to use bootstrap to estimate uncertainty. *0* means use analytic error estimation. 50~200 is a reasonable range to do bootstrap.
- **verbose** (*bool, optional*) – Set to `True` if verbose debug output from `pymbar` is desired.

delta_f_

The estimated dimensionless free energy difference between each state.

Type
DataFrame

d_delta_f_

The estimated statistical uncertainty (one standard deviation) in dimensionless free energy differences.

Type
DataFrame

theta_

The theta matrix.

Type
DataFrame

states_

Lambda states for which free energy differences were obtained.

Type
list

Notes

See [Shirts2008] for details of the derivation and cite the paper when using MBAR in published work.

See also:

`pymbar.mbar.MBAR`

Changed in version 1.0.0: *delta_f_*, *d_delta_f_*, *states_* are view of the original object.

Changed in version 2.0.0: default value for *method* was changed from "hybr" to "robust"

Changed in version 2.1.0: *n_bootstraps* option added.

fit(*u_nk*)

Compute overlap matrix of reduced potentials using multi-state Bennett acceptance ratio.

Parameters
u_nk (*DataFrame*) – `u_nk[n, k]` is the reduced potential energy of uncorrelated configuration `n` evaluated at state `k`.

property overlap_matrix

MBAR overlap matrix.

The estimated state overlap matrix O_{ij} is an estimate of the probability of observing a sample from state i in state j.

The *overlap_matrix* is computed on-the-fly. Assign it to a variable if you plan to re-use it.

See also:

`pymbar.mbar.MBAR.computeOverlap`

`get_metadata_routing()`

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns
routing – A `MetadataRequest` encapsulating routing information.

Return type
MetadataRequest

`get_params(deep=True)`

Get parameters for this estimator.

Parameters
deep (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns
params – Parameter names mapped to their values.

Return type
dict

`set_fit_request(*, u_nk: bool | None | str = '$UNCHANGED$') → MBAR`

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
u_nk (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `u_nk` parameter in `fit`.

Returns
self – The updated object.

Return type
object

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters
****params** (*dict*) – Estimator parameters.

Returns
self – Estimator instance.

Return type
estimator instance

BAR

The *BAR* estimator is a light wrapper around the implementation of the Bennett Acceptance Ratio (BAR) method [Bennett1976] from `pymbar` (`pymbar.mbar.BAR`). It uses information from neighboring sampled states to generate an estimate for the free energy difference between these state.

See also:

alchemlyb.estimators.MBAR

API Reference

`class alchemlyb.estimators.BAR`(*maximum_iterations=10000*, *relative_tolerance=1e-07*, *method='false-position'*, *verbose=False*)

Bennett acceptance ratio (BAR).

Parameters

- **maximum_iterations** (*int, optional*) – Set to limit the maximum number of iterations performed.
- **relative_tolerance** (*float, optional*) – Set to determine the relative tolerance convergence criteria.
- **method** (*str, optional, default='false-position'*) – choice of method to solve BAR nonlinear equations, one of 'self-consistent-iteration' or 'false-position' (default: 'false-position')
- **verbose** (*bool, optional*) – Set to True if verbose debug output is desired.

`delta_f_`

The estimated dimensionless free energy difference between each state.

Type
DataFrame

`d_delta_f_`

The estimated statistical uncertainty (one standard deviation) in dimensionless free energy differences.

Type
DataFrame

`states_`

Lambda states for which free energy differences were obtained.

Type
list

Notes

See [Bennett1976] for details of the derivation and cite the paper (together with [Shirts2008] for the Python implementation in `pymbar`) when using BAR in published work.

When possible, use MBAR instead of BAR as it makes better use of the available data.

See also:

MBAR

Changed in version 1.0.0: *delta_f_*, *d_delta_f_*, *states_* are view of the original object.

`fit`(*u_nk*)

Compute overlap matrix of reduced potentials using Bennett acceptance ratio.

Parameters
u_nk (*DataFrame*) – u_nk[n,k] is the reduced potential energy of uncorrelated configuration n evaluated at state k.

`get_metadata_routing`()

Get metadata routing of this object.

Please check User Guide on how the routing mechanism works.

Returns
routing – A `MetadataRequest` encapsulating routing information.

Return type
MetadataRequest

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters
deep (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns
params – Parameter names mapped to their values.

Return type
dict

`set_fit_request`(*, *u_nk: bool | None | str = '$UNCHANGED$'*) → *BAR*

Request metadata passed to the `fit` method.

Note that this method is only relevant if `enable_metadata_routing=True` (see `sklearn.set_config()`). Please see User Guide on how the routing mechanism works.

The options for each parameter are:

- `True`: metadata is requested, and passed to `fit` if provided. The request is ignored if metadata is not provided.
- `False`: metadata is not requested and the meta-estimator will not pass it to `fit`.
- `None`: metadata is not requested, and the meta-estimator will raise an error if the user provides it.
- `str`: metadata should be passed to the meta-estimator with this given alias instead of the original name.

The default (`sklearn.utils.metadata_routing.UNCHANGED`) retains the existing request. This allows you to change the request for some parameters and not others.

New in version 1.3.

Note: This method is only relevant if this estimator is used as a sub-estimator of a meta-estimator, e.g. used inside a `Pipeline`. Otherwise it has no effect.

Parameters
u_nk (*str, True, False, or None, default=sklearn.utils.metadata_routing.UNCHANGED*) – Metadata routing for `u_nk` parameter in `fit`.

Returns
self – The updated object.

Return type
object

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters
****params** (*dict*) – Estimator parameters.

Returns
self – Estimator instance.

Return type
estimator instance

3.5 Assessing convergence

For a result to be valid, we need to ensure that longer simulation time would not result in different results, i.e., that our results are *converged*. The *alchemlyb.convergence* module provides functions to assess the convergence of free energy estimates or other quantities.

3.5.1 Time Convergence

One way of determining the simulation end point is to compute and plot the forward and backward convergence of the estimate using *forward_backward_convergence()* and *plot_convergence()* [Klimovich2015].

```
>>> from alchemtest.gmx import load_benzene
>>> from alchemlyb.parsing.gmx import extract_u_nk
>>> from alchemlyb.visualisation import plot_convergence
>>> from alchemlyb.convergence import forward_backward_convergence

>>> bz = load_benzene().data
>>> data_list = [extract_u_nk(xvg, T=300) for xvg in bz['Coulomb']]
>>> df = forward_backward_convergence(data_list, 'mbar')
>>> ax = plot_convergence(df)
>>> ax.figure.savefig('dF_t.pdf')
```

Will give a plot looks like this

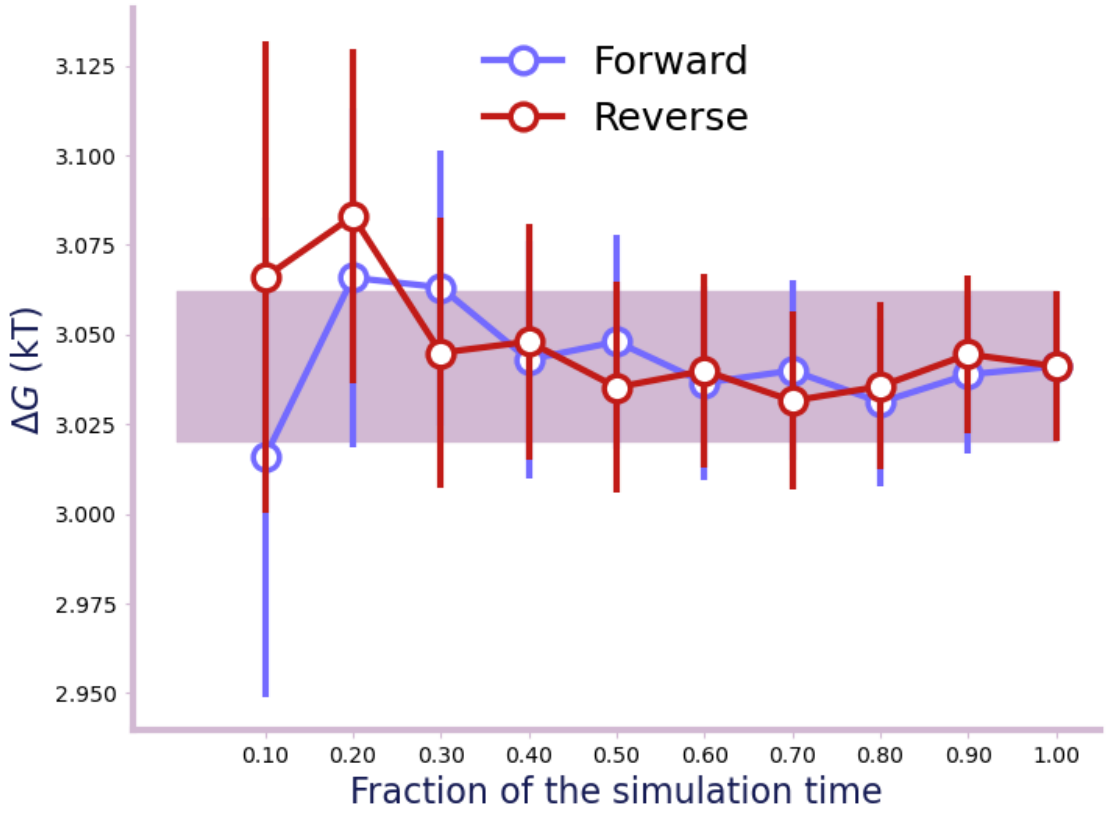


Fig. 1: A convergence plot of showing that the forward and backward has converged fully.

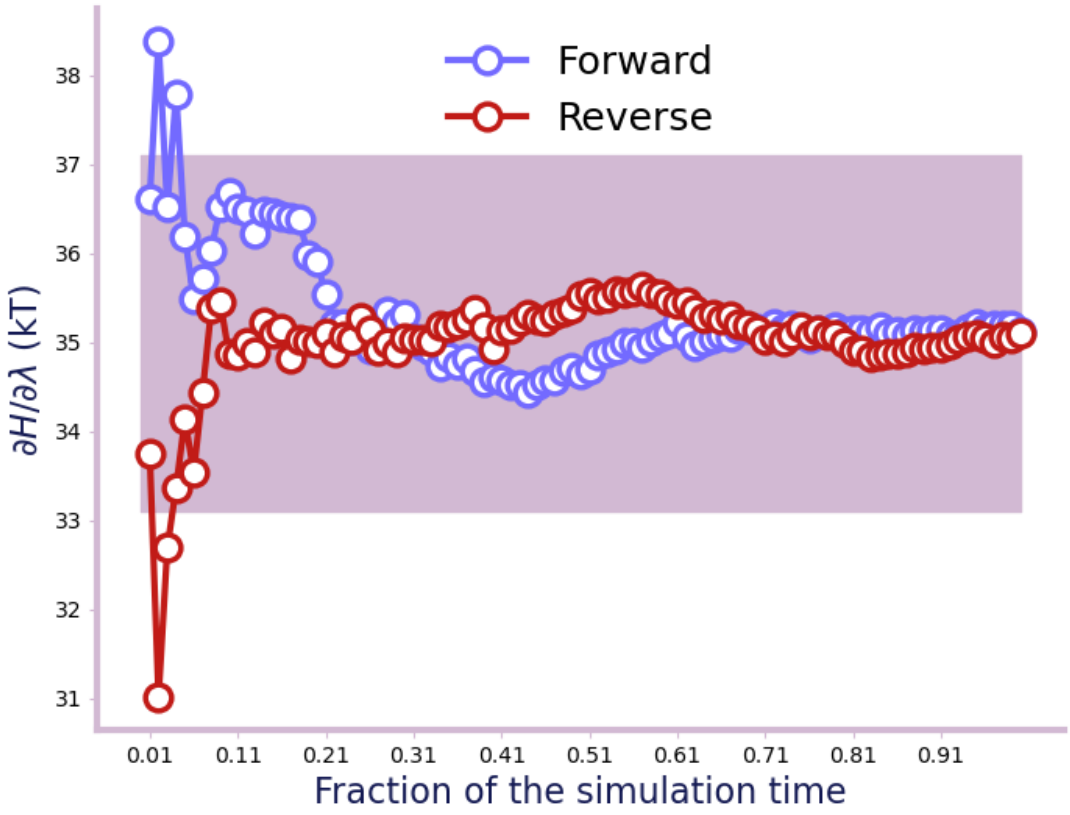
3.5.2 Fractional equilibration time

Another way of assessing whether the simulation has converged is to check the energy files. In [Fan2021] (and [Fan2020]), R_c and A_c are two criteria of checking the convergence. *fwdrev_cumavg_Rc()* takes a decorrelated pandas.Series as input and gives the metric R_c, which is 0 for fully-equilibrated simulation and 1 for fully-unequilibrated simulation.

```
>>> from alchemtest.gmx import load_ABFE
>>> from alchemlyb.parsing.gmx import extract_dHdl
>>> from alchemlyb.preprocessing import decorrelate_dhdl, dhdl2series
>>> from alchemlyb.convergence import fwdrev_cumavg_Rc
>>> from alchemlyb.visualisation import plot_convergence

>>> file = load_ABFE().data['ligand'][0]
>>> dhdl = extract_dHdl(file, T=300)
>>> decorrelated = decorrelate_dhdl(dhdl, remove_burnin=True)
>>> R_c, running_average = fwdrev_cumavg_Rc(dhdl2series(decorrelated), tol=2)
>>> print(R_c)
0.04
>>> ax = plot_convergence(running_average, final_error=2)
>>> ax.set_ylabel("$\partial H/\partial\lambda$ (in kT)")
```

Will give a plot like this.



The *A_c()* on the other hand, takes in a list of decorrelated pandas.Series and gives a metric of how converged the set is, where 0 fully-unequilibrated and 1.0 is fully-equilibrated.

```
>>> from alchemlyb.convergence import A_c
>>> dhdl_list = []
>>> for file in load_ABFE().data['ligand']:
>>>     dhdl = extract_dHdl(file, T=300)
>>>     decorrelated = decorrelate_dhdl(dhdl, remove_burnin=True)
>>>     decorrelated = dhdl2series(decorrelated)
>>>     dhdl_list.append(decorrelated)
>>> value = A_c(dhdl_list, tol=2)
0.7085
```

3.5.3 Convergence functions

Convergence functions are available from *alchemlyb.convergence*. Internally, they are imported from submodules, as documented below.

convergence	Functions for assessing convergence of free energy estimates and raw data.

Convergence API Reference

Functions for assessing convergence of free energy estimates and raw data.

The *alchemlyb.convergence.convergence* module contains building blocks that perform a specific convergence analysis. They typically operate on lists of raw data and either run estimators on these data sets to obtain free energies as a function of the amount of data or they directly assess the convergence of the raw data.

Note: Read the original literature to learn the exact meaning of parameters and how to interpret the output of the convergence analysis.

All convergence functions are located in this submodule but for convenience they are also made available from *alchemlyb.convergence*, as shown here:

alchemlyb.convergence.**forward_backward_convergence**(*df_list*, *estimator='MBAR'*, *num=10*, ***kwargs*)

Forward and backward convergence of the free energy estimate.

Generate the free energy estimate as a function of time in both directions, with the specified number of equally spaced points in the time [Klimovich2015]. For example, setting *num* to 10 would give the forward convergence which is the free energy estimate from the first 10%, 20%, 30%, … of the data. The Backward would give the estimate from the last 10%, 20%, 30%, … of the data.

Parameters

- **df_list** (*list*) – List of DataFrame of either dHdl or u_nk.
- **estimator** (*{'MBAR', 'BAR', 'TI'}*) – Name of the estimators. See the important note below on the use of "MBAR".

  Deprecated since version 1.0.0: Lower case input is also accepted until release 2.0.0.
- **num** (*int*) – The number of time points.

- **kwargs** (*dict*) – Keyword arguments to be passed to the estimator.

Returns

The DataFrame with convergence data.

	Forward	Forward_Error	Backward	Backward_Error	data_fraction
0	3.016442	0.052748	3.065176	0.051036	0.1
1	3.078106	0.037170	3.078567	0.036640	0.2
2	3.072561	0.030186	3.047357	0.029775	0.3
3	3.048325	0.026070	3.057527	0.025743	0.4
4	3.049769	0.023359	3.037454	0.023001	0.5
5	3.034078	0.021260	3.040484	0.021075	0.6
6	3.043274	0.019642	3.032495	0.019517	0.7
7	3.035460	0.018340	3.036670	0.018261	0.8
8	3.042032	0.017319	3.046597	0.017233	0.9
9	3.044149	0.016405	3.044385	0.016402	1.0

Return type
pandas.DataFrame

New in version 0.6.0.

Changed in version 1.0.0: The `estimator` accepts uppercase input. The default for using `estimator='MBAR'` was changed from *MBAR* to `AutoMBAR`.

Changed in version 2.0.0: Use pymbar.MBAR instead of the AutoMBAR option.

alchemlyb.convergence.**fwdrev_cumavg_Rc**(*series*, *precision=0.01*, *tol=2*)

Generate the convergence criteria R_c for a single simulation.

The input will be pandas.Series generated by *decorrelate_u_nk()* or *decorrelate_dhdl()*.

The output will be the float R_c [Fan2020] [Fan2021] and a pandas.DataFrame with the forward and backward cumulative average at *precision* fractional increments, as described below.

$R_c = 0$ indicates that the system is well equilibrated right from the beginning while $R_c = 1$ signifies that the whole trajectory is not equilibrated.

Parameters

- **series** (*pandas.Series*) – The input energy array.
- **precision** (*float*) – The precision of the output R_c. To speed the calculation up, the data has been block-averaged before doing the calculation, the size of the block is controlled by the desired precision.
- **tol** (*float*) – Tolerance (or convergence threshold ϵ in [Fan2021]) in kT.

Returns

- *float* – Convergence time fraction R_c [Fan2021]
- pandas.DataFrame –

  The DataFrame with moving average.

	Forward	Backward	data_fraction
0	3.016442	3.065176	0.1
1	3.078106	3.078567	0.2
2	3.072561	3.047357	0.3
3	3.048325	3.057527	0.4

(continues on next page)

(continued from previous page)

```
4  3.049769  3.037454           0.5
5  3.034078  3.040484           0.6
6  3.043274  3.032495           0.7
7  3.035460  3.036670           0.8
8  3.042032  3.046597           0.9
9  3.044149  3.044385           1.0
```

Notes

This function computes R_c from equation 16 from [Fan2021]. The code is modified based on Shujie Fan's (@VOD555) work. Zhiyi Wu (@xiki-tempula) improved the performance of the original algorithm.

Please cite [Fan2021] when using this function.

See also:

A_c

New in version 1.0.0.

alchemlyb.convergence.**A_c**(*series_list*, *precision=0.01*, *tol=2*)

Generate the ensemble convergence criteria A_c for a set of simulations.

The input is a `list` of `pandas.Series` generated by *decorrelate_u_nk()* or *decorrelate_dhdl()*.

The output will the float A_c [Fan2020] [Fan2021]. A_c is a number between 0 and 1 that can be interpreted as the ratio of the total equilibrated simulation time to the whole simulation time for a full set of simulations. $A_c = 1$ means that all simulation time frames in all windows can be considered equilibrated, while $A_c = 0$ indicates that nothing is equilibrated.

Parameters

- **series_list** (*list*) – A list of `pandas.Series` energy array.
- **precision** (*float*) – The precision of the output A_c. To speed the calculation up, the data has been block-averaged before doing the calculation, the size of the block is controlled by the desired precision.
- **tol** (*float*) – Tolerance (or convergence threshold ϵ in [Fan2021]) in kT.

Returns

The area A_c under curve for convergence time fraction.

Return type

float

Notes

This function computes A_c from equation 18 from [Fan2021].

Please cite [Fan2021] when using this function.

See also:

fwdrev_cumavg_Rc

New in version 1.0.0.

3.6 Tools for postprocessing

Tools are available for postprocessing the dataframes.

3.6.1 Unit Conversion

For all of the input and output dataframes (such as `u_nk`, `dHdl`, `Estimator.delta_f_`, `Estimator.d_delta_f_`), the *metadata* is stored as `pandas.DataFrame.attrs`. The unit of the data can be converted to kT, kJ/mol or kcal/mol via the functions *to_kT()*, *to_kJmol()*, *to_kcalmol()*.

Unit Conversion Functions

units	Unit conversion and constants

alchemlyb.postprocessors.units

Unit conversion and constants

Some examples are given here to illustrate how to use the unit converter functions to convert units.

```
>>> import pandas as pd
>>> import alchemlyb
>>> from alchemtest.gmx import load_benzene
>>> from alchemlyb.parsing.gmx import extract_u_nk
>>> from alchemlyb.estimators import MBAR
>>> from alchemlyb.postprocessors.units import to_kcalmol, to_kJmol, to_kT
>>> bz = load_benzene().data
>>> u_nk_coul = alchemlyb.concat([extract_u_nk(xvg, T=300) for xvg in bz['Coulomb']])
>>> mbar_coul = MBAR().fit(u_nk_coul)
>>> mbar_coul.delta_f_
          0.00      0.25      0.50      0.75      1.00
0.00  0.000000  1.619069  2.557990  2.986302  3.041156
0.25 -1.619069  0.000000  0.938921  1.367232  1.422086
0.50 -2.557990 -0.938921  0.000000  0.428311  0.483165
0.75 -2.986302 -1.367232 -0.428311  0.000000  0.054854
1.00 -3.041156 -1.422086 -0.483165 -0.054854  0.000000
>>> mbar_coul.delta_f_.attrs
{'temperature': 300, 'energy_unit': 'kT'}
```

The default unit is in kT, which could be changed to $kcal/mol$.

```
>>> delta_f_ = to_kcalmol(mbar_coul.delta_f_)
>>> delta_f_
          0.00      0.25      0.50      0.75      1.00
0.00  0.000000  0.965228  1.524977  1.780319  1.813021
0.25 -0.965228  0.000000  0.559749  0.815092  0.847794
0.50 -1.524977 -0.559749  0.000000  0.255343  0.288045
0.75 -1.780319 -0.815092 -0.255343  0.000000  0.032702
1.00 -1.813021 -0.847794 -0.288045 -0.032702  0.000000
```

(continues on next page)

(continued from previous page)

```
>>> delta_f_.attrs
{'temperature': 300, 'energy_unit': 'kcal/mol'}
```

alchemlyb.postprocessors.units.**to_kcalmol**(*df*, *T=None*)

Convert the unit of a DataFrame to kcal/mol.

If temperature *T* is not provided, the DataFrame need to have attribute *temperature* and *energy_unit*. Otherwise, the temperature of the output dateframe will be set accordingly.

Parameters

- **df** (*DataFrame*) – DataFrame to convert unit.
- **T** (*float*) – Temperature (default: None).

Returns
df converted.

Return type
DataFrame

The unit could also be changed to kJ/mol.

```
>>> delta_f_ = to_kJmol(delta_f_)
>>> delta_f_
          0.00      0.25      0.50      0.75      1.00
0.00  0.000000  4.038508  6.380495  7.448848  7.585673
0.25 -4.038508  0.000000  2.341987  3.410341  3.547165
0.50 -6.380495 -2.341987  0.000000  1.068354  1.205178
0.75 -7.448848 -3.410341 -1.068354  0.000000  0.136825
1.00 -7.585673 -3.547165 -1.205178 -0.136825  0.000000
>>> delta_f_.attrs
{'temperature': 300, 'energy_unit': 'kJ/mol'}
```

alchemlyb.postprocessors.units.**to_kJmol**(*df*, *T=None*)

Convert the unit of a DataFrame to kJ/mol.

If temperature *T* is not provided, the DataFrame need to have attribute *temperature* and *energy_unit*. Otherwise, the temperature of the output dateframe will be set accordingly.

Parameters

- **df** (*DataFrame*) – DataFrame to convert unit.
- **T** (*float*) – Temperature (default: None).

Returns
df converted.

Return type
DataFrame

And change back to kT again.

```
>>> delta_f_ = to_kT(delta_f_)
          0.00      0.25      0.50      0.75      1.00
0.00  0.000000  1.619069  2.557990  2.986302  3.041156
0.25 -1.619069  0.000000  0.938921  1.367232  1.422086
0.50 -2.557990 -0.938921  0.000000  0.428311  0.483165
```

(continues on next page)

(continued from previous page)

```
0.75 -2.986302 -1.367232 -0.428311  0.000000  0.054854
1.00 -3.041156 -1.422086 -0.483165 -0.054854  0.000000
>>> delta_f_.attrs
{'temperature': 300, 'energy_unit': 'kT'}
```

alchemlyb.postprocessors.units.**to_kT**(*df*, *T=None*)

Convert the unit of a DataFrame to *kT*.

If temperature *T* is not provided, the DataFrame need to have attribute *temperature* and *energy_unit*. Otherwise, the temperature of the output dateframe will be set accordingly.

Parameters

- **df** (*DataFrame*) – DataFrame to convert unit.
- **T** (*float*) – Temperature (default: None).

Returns
df converted.

Return type
DataFrame

A dispatch table approach is also provided to return the relevant converter for every units.

alchemlyb.postprocessors.units.**get_unit_converter**(*units*)

Obtain the converter according to the unit string.

If *units* is 'kT', the *to_kT* converter is returned. If *units* is 'kJ/mol', the *to_kJmol* converter is returned. If *units* is 'kcal/mol', the *to_kcalmol* converter is returned.

Parameters
units (*str*) – The unit that the function converts to.

Returns
converter

Return type
func

New in version 0.5.0.

3.6.2 Constants and auxiliary functions

The postprocessing functions can make use of the following auxiliary functions, which in turn may use constants defined *alchemlyb.postprocessors.units*.

Scientific constants

Common scientific constants based on `scipy.constants` and are provided for use across **alchemlyb**.

alchemlyb.postprocessors.units.**kJ2kcal = 0.2390057361376673**

conversion factor from kJ to kcal, based on `scipy.constants.calorie` in `scipy.constants`

alchemlyb.postprocessors.units.**R_kJmol = 0.008314462618**

gas constant R in kJ/(mol K), based on `scipy.constants.R` in `scipy.constants`

Unit conversion developer information

The function *alchemlyb.postprocessors.units.get_unit_converter()* provides the relevant converter for unit conversion via a built-in dispatch table:

```
>>> from alchemlyb.postprocessors.units import get_unit_converter
>>> get_unit_converter('kT')
<function to_kT>
>>> get_unit_converter('kJ/mol')
<function to_kJmol>
>>> get_unit_converter('kcal/mol')
<function to_kcalmol>
```

For unit conversion to work, the dataframes *must* maintain the **energy_unit** and **temperature** metadata in `pandas.DataFrame.attrs` as described under *A note on units*.

When *implementing* code then ensure that the *metadata are maintained* by using *alchemlyb.concat()* in place of `pandas.concat()` and use the *alchemlyb.pass_attrs()* decorator to copy metadata from an input dataframe to an output dataframe.

3.7 Visualisation of the results

It is quite often that the user want to visualise the results to gain confidence on the computed free energy. **alchemlyb** provides various visualisation tools to help user to judge the estimate.

3.7.1 Plotting Functions

plot_mbar_overlap_matrix(matrix[, ...])	Plot the MBAR overlap matrix.
plot_ti_dhdl(dhdl_data[, labels, colors, ...])	Plot the dhdl of TI.
plot_dF_state(estimators[, labels, colors, ...])	Plot the dhdl of TI.
plot_convergence(dataframe[, units, ...])	Plot the forward and backward convergence.

Plot Overlap Matrix from MBAR

The function *plot_mbar_overlap_matrix()* allows the user to plot the overlap matrix from *overlap_matrix*. The user can pass `matplotlib.axes.Axes` into the function to have the overlap maxtrix drawn on a specific axes. The user could also specify a list of lambda states to be skipped when labelling the states.

Please check *How to plot MBAR overlap matrix* for usage.

API Reference

alchemlyb.visualisation.**plot_mbar_overlap_matrix**(*matrix*, *skip_lambda_index=[]*, *ax=None*)

Plot the MBAR overlap matrix.

Parameters

- **matrix** (*numpy.matrix*) – DataFrame of the overlap matrix obtained from *overlap_matrix*
- **skip_lambda_index** (*List*) – list of lambda indices to be omitted from plotting process. Default: `[]`.
- **ax** (*matplotlib.axes.Axes*) – Matplotlib axes object where the plot will be drawn on. If `ax=None`, a new axes will be generated.

Returns

An axes with the overlap matrix drawn.

Return type

matplotlib.axes.Axes

Note: The code is taken and modified from Alchemical Analysis.

New in version 0.4.0.

Plot dhdl from TI

The function *plot_ti_dhdl()* allows the user to plot the dhdl from *TI* estimator. Several *TI* estimators could be passed to the function to give a concerted picture of the whole alchemical transformation. When custom labels are desirable, the user could pass a list of strings to the *labels* for labelling each alchemical transformation differently. The color of each alchemical transformation could also be set by passing a list of color string to the *colors*. The unit in the y axis could be labelled to other units by setting *units*, which by default is kT. The user can pass `matplotlib.axes.Axes` into the function to have the dhdl drawn on a specific axes.

Please check *How to plot TI dhdl* for usage.

API Reference

alchemlyb.visualisation.**plot_ti_dhdl**(*dhdl_data*, *labels=None*, *colors=None*, *units=None*, *ax=None*)

Plot the dhdl of TI.

Parameters

- **dhdl_data** (*TI* or list) – One or more *TI* estimator, where the dhdl value will be taken from.
- **labels** (*List*) – list of labels for labelling all the alchemical transformations.
- **colors** (*List*) – list of colors for plotting all the alchemical transformations. Default: [‘r’, ‘g’, ‘#7F38EC’, ‘#9F000F’, ‘b’, ‘y’]
- **units** (*str*) – The unit of the estimate. The default is *None*, which is to use the unit in the input. Setting this will change the output unit.
- **ax** (*matplotlib.axes.Axes*) – Matplotlib axes object where the plot will be drawn on. If `ax=None`, a new axes will be generated.

Returns
An axes with the TI dhdl drawn.

Return type
matplotlib.axes.Axes

Note: The code is taken and modified from Alchemical Analysis.

Changed in version 1.0.0: If no units is given, the *units* in the input will be used.

Changed in version 0.5.0: The *units* will be used to change the underlying data instead of only changing the figure legend.

New in version 0.4.0.

Plot dF states from multiple estimators

The function *plot_dF_state()* allows the user to plot and compare the free energy difference between states ("dF") from various kinds of *estimators*.

To compare the dF states of a single alchemical transformation among various *estimators*, the user can pass a list of *estimators*. (e.g. *estimators* = [*TI*, *BAR*, *MBAR*])

To compare the dF states of a multiple alchemical transformations, results from the same *estimators* can be concatenated into a list, which is then bundled to to another list of different *estimators*. (e.g. *estimators* = [(*TI*, *TI*), (*BAR*, *BAR*), (*MBAR*, *MBAR*)])

The figure could be plotted in *portrait* or *landscape* mode by setting the *orientation*. *nb* is used to control the number of dF states in one row. The user could pass a list of strings to *labels* to name the *estimators* or a list of strings to *colors* to color the estimators differently. The unit in the y axis could be labelled to other units by setting *units*, which by default is kT.

Please check *How to plot dF states* for a complete example.

API Reference

alchemlyb.visualisation.**plot_dF_state**(*estimators*, *labels=None*, *colors=None*, *units=None*, *orientation='portrait'*, *nb=10*)

Plot the dhdl of TI.

Parameters

- **estimators** (*estimators* or list) – One or more *estimators*, where the dhdl value will be taken from. For more than one estimators with more than one alchemical transformation, a list of list format is used.
- **labels** (*List*) – list of labels for labelling different estimators.
- **colors** (*List*) – list of colors for plotting different estimators.
- **units** (*str*) – The unit of the estimate. The default is *None*, which is to use the unit in the input. Setting this will change the output unit.
- **orientation** (*string*) – The orientation of the figure. Can be *portrait* or *landscape*
- **nb** (*int*) – Maximum number of dF states in one row in the *portrait* mode

Returns
An Figure with the dF states drawn.

Return type
matplotlib.figure.Figure

Note: The code is taken and modified from Alchemical Analysis.

Changed in version 1.0.0: If no units is given, the *units* in the input will be used.

Changed in version 0.5.0: The *units* will be used to change the underlying data instead of only changing the figure legend.

New in version 0.4.0.

Plot the Forward and Backward Convergence

The function `plot_convergence()` allows the user to visualise the convergence by plotting the free energy change computed using the equilibrated snapshots between the proper target time frames. The data could be provided as a Dataframe from `alchemlyb.convergence.forward_backward_convergence()` or provided explicitly in both forward (data points are stored in *forward* and *forward_error*) and reverse (data points are stored in *backward* and *backward_error*) directions.

The unit in the y axis could be labelled to other units by setting *units*, which by default is kT. The user can pass `matplotlib.axes.Axes` into the function to have the convergence drawn on a specific axes.

Please check *How to plot convergence* for usage.

API Reference

`alchemlyb.visualisation.`**`plot_convergence`**(*dataframe*, *units=None*, *final_error=None*, *ax=None*)

Plot the forward and backward convergence.

The input could be the result from `forward_backward_convergence()` or `fwdrev_cumavg_Rc()`. The input should be a `pandas.DataFrame` which has column *Forward*, *Backward* and `pandas.DataFrame.attrs` should compile with *A note on units*. The errorbar will be plotted if column *Forward_Error* and *Backward_Error* is present.

Forward: A column of free energy estimate from the first X% of data, where optional *Forward_Error* column is the corresponding error.

Backward: A column of free energy estimate from the last X% of data., where optional *Backward_Error* column is the corresponding error.

final_error is the error of the final value and is shown as the error band around the final value. It can be provided in case an estimate is available that is more appropriate than the default, which is the error of the last value in *Backward*.

dataframe
[Dataframe] Output Dataframe has column *Forward*, *Backward* or optionally *Forward_Error*, *Backward_Error* see *plot_convergence*.

units
[str] The unit of the estimate. The default is *None*, which is to use the unit in the input. Setting this will change the output unit.

final_error
[float] The error of the final value in `units`. If not given, takes the last error in *backward_error*.

ax
[matplotlib.axes.Axes] Matplotlib axes object where the plot will be drawn on. If `ax=None`, a new axes will be generated.

matplotlib.axes.Axes
An axes with the forward and backward convergence drawn.

The code is taken and modified from Alchemical Analysis.

Changed in version 1.0.0: Keyword arg final_error for plotting a horizontal error bar. The array input has been deprecated. The units default to *None* which uses the units in the input.

Changed in version 0.6.0: data now takes in dataframe

New in version 0.4.0.

3.7.2 Overlap Matrix of the MBAR

The accuracy of the *MBAR* estimator depends on the overlap between different lambda states. The overlap matrix from the *MBAR* estimator could be plotted using *plot_mbar_overlap_matrix()* to check the degree of overlap. It is recommended that there should be at least **0.03** [Klimovich2015] overlap between neighboring states.

```
>>> import pandas as pd
>>> from alchemtest.gmx import load_benzene
>>> from alchemlyb.parsing.gmx import extract_u_nk
>>> from alchemlyb.estimators import MBAR

>>> bz = load_benzene().data
>>> u_nk_coul = alchemlyb.concat([extract_u_nk(xvg, T=300) for xvg in bz['Coulomb']])
>>> mbar_coul = MBAR()
>>> mbar_coul.fit(u_nk_coul)

>>> from alchemlyb.visualisation import plot_mbar_overlap_matrix
>>> ax = plot_mbar_overlap_matrix(mbar_coul.overlap_matrix)
>>> ax.figure.savefig('O_MBAR.pdf', bbox_inches='tight', pad_inches=0.0)
```

Will give a plot looks like this

3.7.3 dhdl Plot of the TI

In order for the *TI* estimator to work reliably, the change in the dhdl between lambda state 0 and lambda state 1 should be adequately sampled. The function *plot_ti_dhdl()* can be used to assess the change of the dhdl across the lambda states.

More than one *TI* estimators can be plotted together as well.

```
>>> import pandas as pd
>>> from alchemtest.gmx import load_benzene
>>> from alchemlyb.parsing.gmx import extract_dHdl
>>> from alchemlyb.estimators import TI
```

(continues on next page)

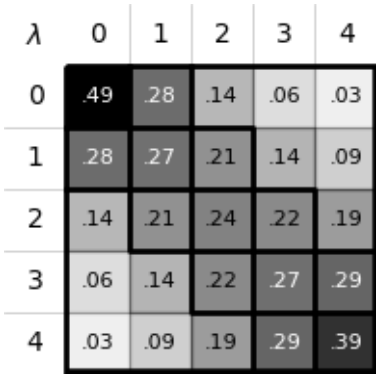


Fig. 2: Overlap between the distributions of potential energy differences is essential for accurate free energy calculations and can be quantified by computing the overlap matrix . Its elements are the probabilities of observing a sample from state i (th row) in state j (th column).

(continued from previous page)

```
>>> bz = load_benzene().data
>>> dHdl_coul = alchemlyb.concat([extract_dHdl(xvg, T=300) for xvg in bz['Coulomb']])
>>> ti_coul = TI().fit(dHdl_coul)
>>> dHdl_vdw = alchemlyb.concat([extract_dHdl(xvg, T=300) for xvg in bz['VDW']])
>>> ti_vdw = TI().fit(dHdl_vdw)

>>> from alchemlyb.visualisation import plot_ti_dhdl
>>> ax = plot_ti_dhdl([ti_coul, ti_vdw], labels=['Coul', 'VDW'], colors=['r', 'g'])
>>> ax.figure.savefig('dhdl_TI.pdf')
```

Will give a plot looks like this

3.7.4 dF States Plots between Different estimators

Another way of assessing the quality of free energy estimate would be comparing the free energy difference between adjacent lambda states (dF) using different estimators [Klimovich2015]. The function *plot_dF_state()* can be used, for example, to compare the dF of both Coulombic and VDW transformations using *TI*, *BAR* and *MBAR* estimators.

```
>>> from alchemtest.gmx import load_benzene
>>> from alchemlyb.parsing.gmx import extract_u_nk, extract_dHdl
>>> from alchemlyb.estimators import MBAR, TI, BAR
>>> import matplotlib.pyplot as plt
>>> import pandas as pd
>>> from alchemlyb.visualisation.dF_state import plot_dF_state
>>> bz = load_benzene().data
>>> u_nk_coul = alchemlyb.concat([extract_u_nk(xvg, T=300) for xvg in bz['Coulomb']])
>>> dHdl_coul = alchemlyb.concat([extract_dHdl(xvg, T=300) for xvg in bz['Coulomb']])
>>> u_nk_vdw = alchemlyb.concat([extract_u_nk(xvg, T=300) for xvg in bz['VDW']])
>>> dHdl_vdw = alchemlyb.concat([extract_dHdl(xvg, T=300) for xvg in bz['VDW']])
>>> ti_coul = TI().fit(dHdl_coul)
>>> ti_vdw = TI().fit(dHdl_vdw)
>>> bar_coul = BAR().fit(u_nk_coul)
>>> bar_vdw = BAR().fit(u_nk_vdw)
>>> mbar_coul = MBAR().fit(u_nk_coul)
```

(continues on next page)

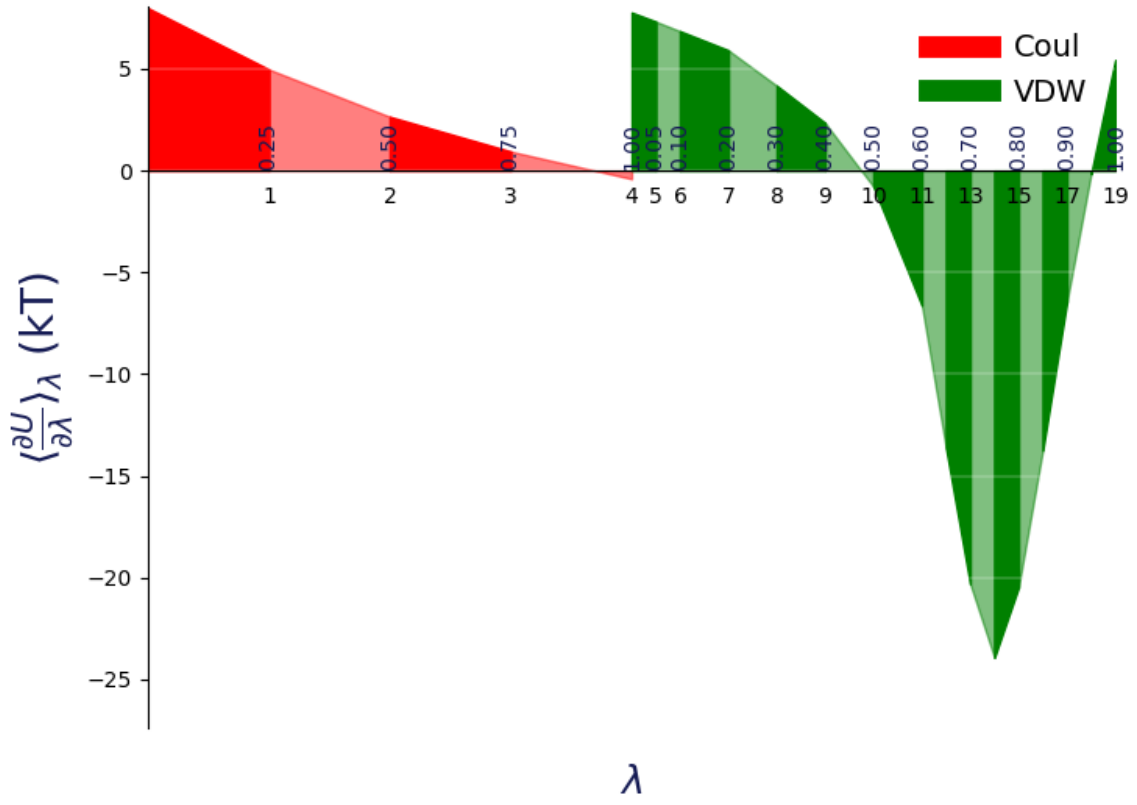


Fig. 3: A plot of ⟨/⟩ versus for thermodynamic integration, with filled areas indicating free energy estimates from the trapezoid rule. Different components are shown in distinct colors: in red is the electrostatic component (indices 0–4), while in green is the van der Waals component (indices 5–19). Color intensity alternates with increasing index.

(continued from previous page)

```
>>> mbar_vdw = MBAR().fit(u_nk_vdw)

>>> estimators = [(ti_coul, ti_vdw),
                  (bar_coul, bar_vdw),
                  (mbar_coul, mbar_vdw),]

>>> fig = plot_dF_state(estimators, orientation='portrait')
>>> fig.savefig('dF_state.pdf', bbox_inches='tight')
```

Will give a plot looks like this

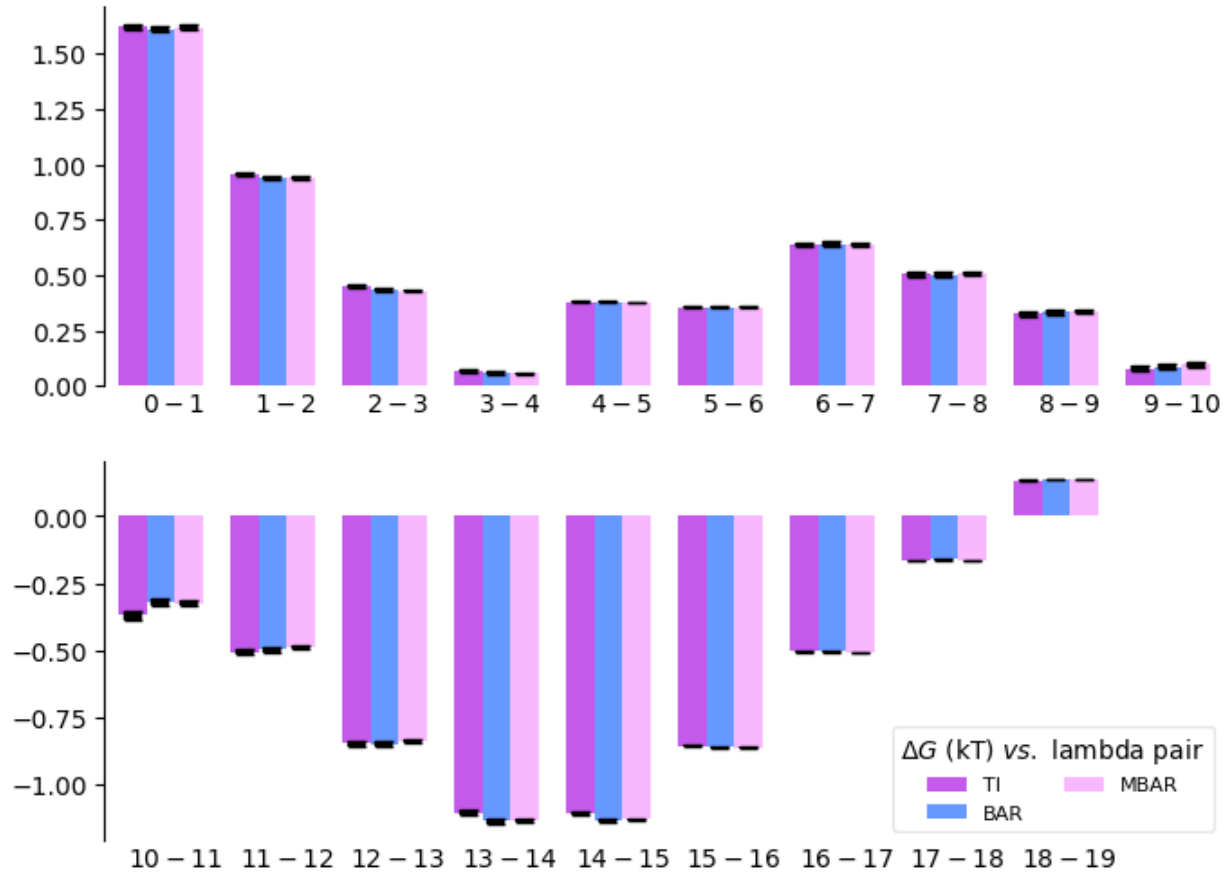


Fig. 4: A bar plot of the free energy differences evaluated between pairs of adjacent states via several methods, with corresponding error estimates for each method.

3.7.5 Forward and Backward Convergence

One way of determining the simulation end point is to plot the forward and backward convergence of the estimate using *plot_convergence()*.

Note that this is just a plotting function to plot [Klimovich2015] style convergence plot. The user need to provide the forward and backward data list and the corresponding error.

```
>>> import pandas as pd
>>> from alchemtest.gmx import load_benzene
>>> from alchemlyb.parsing.gmx import extract_u_nk
>>> from alchemlyb.estimators import MBAR

>>> bz = load_benzene().data
>>> data_list = [extract_u_nk(xvg, T=300) for xvg in bz['Coulomb']]
>>> forward = []
>>> forward_error = []
>>> backward = []
>>> backward_error = []
>>> num_points = 10
>>> for i in range(1, num_points+1):
>>>     # Do the forward
>>>     slice = int(len(data_list[0])/num_points*i)
>>>     u_nk_coul = alchemlyb.concat([data[:slice] for data in data_list])
>>>     estimate = MBAR().fit(u_nk_coul)
>>>     forward.append(estimate.delta_f_.iloc[0,-1])
>>>     forward_error.append(estimate.d_delta_f_.iloc[0,-1])
>>>     # Do the backward
>>>     u_nk_coul = alchemlyb.concat([data[-slice:] for data in data_list])
>>>     estimate = MBAR().fit(u_nk_coul)
>>>     backward.append(estimate.delta_f_.iloc[0,-1])
>>>     backward_error.append(estimate.d_delta_f_.iloc[0,-1])

>>> from alchemlyb.visualisation import plot_convergence
>>> ax = plot_convergence(forward, forward_error, backward, backward_error)
>>> ax.figure.savefig('dF_t.pdf')
```

Will give a plot looks like this

3.8 Automatic workflow

Though **alchemlyb** is a library offering great flexibility in deriving free energy estimate, it also provides workflows that provides automatic analysis of the results and step-by-step version that allows more flexibility.

For developers, the skeleton of the workflow should follow the example in *alchemlyb.workflows.base.WorkflowBase*.

For users, **alchemlyb** offers a workflow *alchemlyb.workflows.ABFE* similar to Alchemical Analysis for doing automatic absolute binding free energy (ABFE) analysis.

base	Basic building blocks for free energy workflows.
ABFE(T[, units, software, dir, prefix, ...])	Workflow for absolute and relative binding free energy calculations.

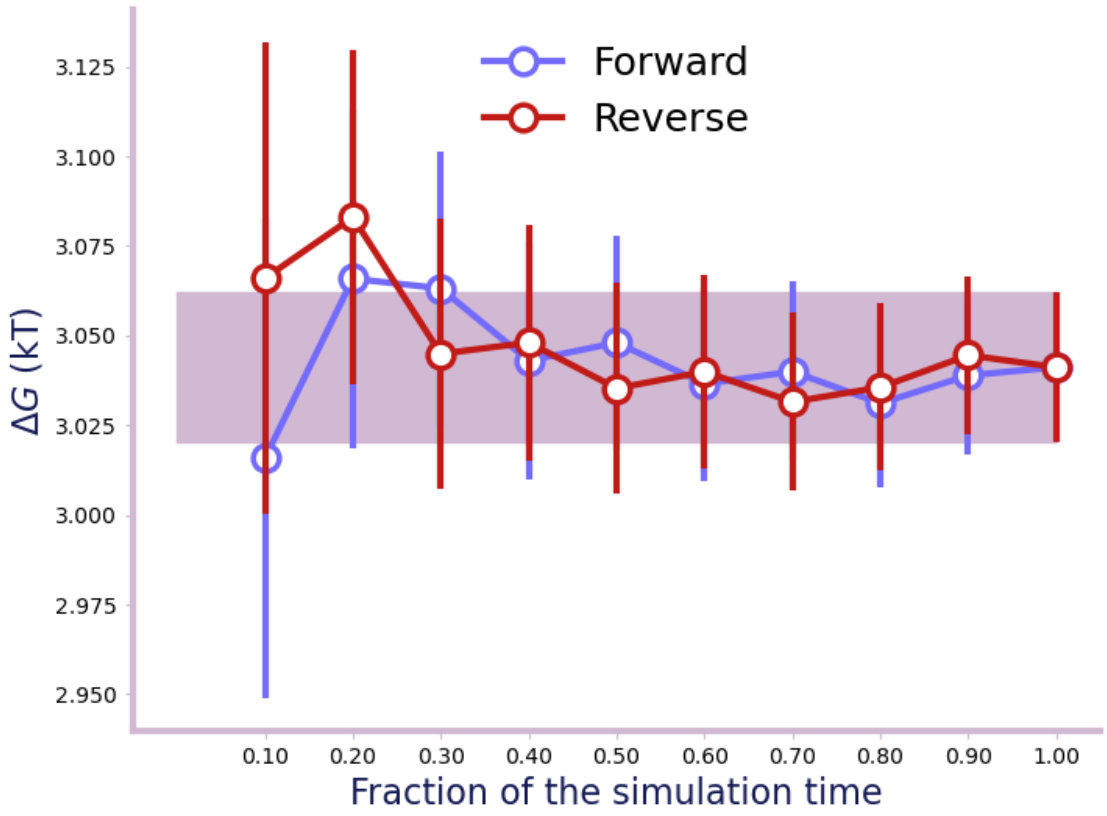


Fig. 5: A convergence plot of showing that the forward and backward has converged fully.

3.8.1 The base workflow

Basic building blocks for free energy workflows.

The *alchemlyb.workflows.base.WorkflowBase* class provides a basic API template for the workflow development. The workflow should be able to run in an automatic fashion.

```
>>> from alchemlyb.workflows.base import WorkflowBase
>>> workflow = WorkflowBase(units='kT', software='Gromacs', T=298,
    out='./', *args, **kwargs)
>>> workflow.run(*args, **kwargs)
```

Three main functions are provided such that the workflow could be run in a step-by-step fashion.

```
>>> from alchemlyb.workflows.base import WorkflowBase
>>> workflow = WorkflowBase(units='kT', software='Gromacs', T=298,
    out='./', *args, **kwargs)
>>> workflow.read(*args, **kwargs)
>>> workflow.preprocess(*args, **kwargs)
>>> workflow.estimate(*args, **kwargs)
>>> workflow.check_convergence(*args, **kwargs)
>>> workflow.plot(*args, **kwargs)
```

API Reference

class alchemlyb.workflows.base.**WorkflowBase**(*units='kT'*, *software='Gromacs'*, *T=298*, *out='./'*, **args*, ***kwargs*)

The base class for the Workflow.

This is the base class for the creation of new Workflow. The initialisation method takes in the MD engine, unit, temperature and output directory. The goal of the initialisation is to check the input files and store them in `file_list` such that they can be read by the `read()` method.

Parameters

- **units** (*string, optional*) – The unit used for printing and plotting results. {'kcal/mol', 'kJ/mol', 'kT'}. Default: 'kT'.
- **software** (*string, optional*) – The software used for generating input. {'Gromacs', 'Amber'}
- **T** (*float, optional,*) – Temperature in K. Default: 298.
- **out** (*string, optional*) – Directory in which the output files produced by this script will be stored. Default: './'.

file_list

A list of files to be read by the parser.

Type
list

New in version 0.7.0.

run(**args*, ***kwargs*)

Run the workflow in an automatic fashion.

This method would execute the `read()`, `preprocess()`, `estimate()`, `check_convergence()`, `plot()` sequentially such that the whole analysis could be done automatically.

This method takes in an arbitrary number of arguments and pass all of them to the underlying methods. The methods will be selecting the keywords that they would like to use.

Running this method would generate the resulting attributes for the user to retrieve the results.

u_nk_list

A list of `pandas.DataFrame` of u_nk.

Type
list

dHdl_list

A list of `pandas.DataFrame` of dHdl.

Type
list

u_nk_sample_list

A list of `pandas.DataFrame` of the subsampled u_nk.

Type
list

dHdl_sample_list

A list of `pandas.DataFrame` of the subsampled dHdl.

Type
list

result

The main result of the workflow.

Type
pandas.Dataframe

convergence

The result of the convergence analysis.

Type
pandas.Dataframe

read(**args*, ***kwargs*)

The function that reads the files in *file_list* and parse them into u_nk and dHdl files.

u_nk_list

A list of `pandas.DataFrame` of u_nk.

Type
list

dHdl_list

A list of `pandas.DataFrame` of dHdl.

Type
list

preprocess(**args*, ***kwargs*)

The function that subsample the u_nk and dHdl in *u_nk_list* and *dHdl_list*.

u_nk_sample_list

A list of `pandas.DataFrame` of the subsampled u_nk.

Type
list

dHdl_sample_list

A list of `pandas.DataFrame` of the subsampled dHdl.

Type

list

estimate(**args*, ***kwargs*)

The function that runs the estimator based on *u_nk_sample_list* and *dHdl_sample_list*.

result

The main result of the workflow.

Type

pandas.Dataframe

check_convergence(**args*, ***kwargs*)

The function for doing convergence analysis.

convergence

The result of the convergence analysis.

Type

pandas.Dataframe

plot(**args*, ***kwargs*)

The function for producing any plots.

3.8.2 The ABFE workflow

The *Absolute binding free energy* (ABFE) workflow provides a complete workflow that uses the energy files generated by MD engine as input and generates the binding free energy as well as the analysis plots.

Fully Automatic analysis

Absolute binding free energy (ABFE) calculations can be analyzed with two lines of code in a fully automated manner (similar to Alchemical Analysis). In this case, any parameters are set when invoking *ABFE* and reasonable defaults are chosen for any parameters not set explicitly. The two steps are to

1. initialize an instance of the *ABFE* class
2. invoke the *run()* method to execute complete workflow.

For a GROMACS ABFE simulation, executing the workflow would look similar to the following code (*See how to configure the logger*).

```
>>> from alchemtest.gmx import load_ABFE
>>> from alchemlyb.workflows import ABFE
>>> # Obtain the path of the data
>>> import os
>>> dir = os.path.dirname(load_ABFE()['data']['complex'][0])
>>> print(dir)
'alchemtest/gmx/ABFE/complex'
>>> workflow = ABFE(units='kcal/mol', software='GROMACS', dir=dir,
>>>                 prefix='dhdl', suffix='xvg', T=298, outdirectory='./')
>>> workflow.run(skiptime=10, uncorr='dhdl', threshold=50,
>>>              estimators=('MBAR', 'BAR', 'TI'), overlap='O_MBAR.pdf',
>>>              breakdown=True, forwrev=10)
```

The workflow uses the *parsing* to parse the data from the energy files, remove the initial unequilibrated frames and decorrelate the data with *subsampling*. The decorrelated dataset *dHdl* and *u_nk* are then passed to *estimators* for free energy estimation. The workflow will also perform a set of analysis that allows the user to examine the quality of the estimation.

File Input

This command expects the energy files to be structured in two common ways. It could either be

```
simulation
├── lambda_0
│   ├── prod.xvg
│   └── ...
├── lambda_1
│   ├── prod.xvg
│   └── ...
└── ...
```

Where `dir='simulation/lambda_*', prefix='prod', suffix='xvg'`. Or

```
dhdl_files
├── dhdl_0.xvg
├── dhdl_1.xvg
└── ...
```

Where `dir='dhdl_files', prefix='dhdl_', suffix='xvg'`.

Output

The workflow returns the free energy estimate using all of *TI*, *BAR*, *MBAR*. For ABFE calculations, the alchemical transformation is usually done is three stages, the *bonded*, *coul* and *vdw* which corresponds to the free energy contribution from applying the restraint to restrain the ligand to the protein, decouple/annihilate the coulombic interaction between the ligand and the protein and decouple/annihilate the protein-ligand lennard jones interactions. The result will be stored in *summary* as `pandas.Dataframe`.

		MBAR	MBAR_Error	BAR	BAR_Error	TI	TI_Error
States	0 -- 1	0.065967	0.001293	0.066544	0.001661	0.066663	0.001675
	1 -- 2	0.089774	0.001398	0.089303	0.002101	0.089566	0.002144
	2 -- 3	0.132036	0.001638	0.132687	0.002990	0.133292	0.003055
...							
	26 -- 27	1.243745	0.011239	1.245873	0.015711	1.248959	0.015762
	27 -- 28	1.128429	0.012859	1.124554	0.016999	1.121892	0.016962
	28 -- 29	1.010313	0.016442	1.005444	0.017692	1.019747	0.017257
Stages	coul	10.215658	0.033903	10.017838	0.041839	10.017854	0.048744
	vdw	22.547489	0.098699	22.501150	0.060092	22.542936	0.106723
	bonded	2.374144	0.014995	2.341631	0.005507	2.363828	0.021078
	TOTAL	35.137291	0.103580	34.860619	0.087022	34.924618	0.119206

Output Files

For quality assessment, a couple of plots were generated and written to the folder specified by *outdirectory*.

The *overlay matrix for the MBAR estimator* will be plotted and saved to `O_MBAR.pdf`, which examines the overlap between different lambda windows.

The *dHdl for TI* will be plotted to `dhdl_TI.pdf`, allows one to examine if the lambda scheduling has covered the change of the gradient in the lambda space.

The *dF states* will be plotted to `dF_state.pdf` in portrait model and `dF_state_long.pdf` in landscape model, which allows the user to example the contributions from each lambda window.

The forward and backward convergence will be plotted to `dF_t.pdf` using *MBAR* and saved in *convergence*, which allows the user to examine if the simulation time is enough to achieve a converged result.

Semi-automatic analysis

The same analysis could also performed in steps allowing access and modification to the data generated at each stage of the analysis.

```
>>> from alchemtest.gmx import load_ABFE
>>> from alchemlyb.workflows import ABFE
>>> # Obtain the path of the data
>>> import os
>>> dir = os.path.dirname(load_ABFE()['data']['complex'][0])
>>> print(dir)
'alchemtest/gmx/ABFE/complex'
>>> # Load the data
>>> workflow = ABFE(software='GROMACS', dir=dir,
>>>                 prefix='dhdl', suffix='xvg', T=298, outdirectory='./')
>>> # Set the unit.
>>> workflow.update_units('kcal/mol')
>>> # Read the data
>>> workflow.read()
>>> # Decorrelate the data.
>>> workflow.preprocess(skiptime=10, uncorr='dhdl', threshold=50)
>>> # Run the estimator
>>> workflow.estimate(estimators=('mbar', 'bar', 'ti'))
>>> # Retrieve the result
>>> summary = workflow.generate_result()
>>> # Plot the overlap matrix
>>> workflow.plot_overlap_matrix(overlap='O_MBAR.pdf')
>>> # Plot the dHdl for TI
>>> workflow.plot_ti_dhdl(dhdl_TI='dhdl_TI.pdf')
>>> # Plot the dF states
>>> workflow.plot_dF_state(dF_state='dF_state.pdf')
>>> # Convergence analysis
>>> workflow.check_convergence(10, dF_t='dF_t.pdf')
```

API Reference

class alchemlyb.workflows.**ABFE**(*T*, *units='kT'*, *software='GROMACS'*, *dir='.'*, *prefix='dhdl'*, *suffix='xvg'*, *outdirectory='.'*)

Workflow for absolute and relative binding free energy calculations.

This workflow provides functionality similar to the `alchemical-analysis.py` script. It loads multiple input files from alchemical free energy calculations and computes the free energies between different alchemical windows using different estimators. It produces plots to aid in the assessment of convergence.

Parameters

- **T** (*float*) – Temperature in K.
- **units** (*str*) – The unit used for printing and plotting results. {'kcal/mol', 'kJ/mol', 'kT'}. Default: 'kT'.
- **software** (*str*) – The software used for generating input (case-insensitive). {'GROMACS', 'AMBER', 'PARQUET'}. This option chooses the appropriate parser for the input file.
- **dir** (*str*) – Directory in which data files are stored. Default: os.path.curdir.
- **prefix** (*str*) – Prefix for datafile sets. This argument accepts regular expressions and the input files are searched using `Path(dir).glob("**/" + prefix + "*" + suffix)`. Default: 'dhdl'.
- **suffix** (*str*) – Suffix for datafile sets. Default: 'xvg'.
- **outdirectory** (*str*) – Directory in which the output files produced by this script will be stored. Default: os.path.curdir.

logger

The logging object.

Type
Logger

file_list

The list of filenames sorted by the lambda state.

Type
list

New in version 1.0.0.

Changed in version 2.0.1: The *dir* argument expects a real directory without wildcards and wildcards will no longer work as expected. Use *prefix* to specify wildcard-based patterns to search under *dir*.

Changed in version 2.1.0: The serialised dataframe could be read via software='PARQUET'.

read(*read_u_nk=True*, *read_dHdl=True*)

Read the u_nk and dHdL data from the *file_list*

Parameters

- **read_u_nk** (*bool*) – Whether to read the u_nk.
- **read_dHdl** (*bool*) – Whether to read the dHdl.

u_nk_list

A list of `pandas.DataFrame` of u_nk.

Type
list

dHdl_list

A list of pandas.DataFrame of dHdl.

Type
list

run(*skiptime=0*, *uncorr='dE'*, *threshold=50*, *estimators=('MBAR', 'BAR', 'TI')*, *overlap='O_MBAR.pdf'*, *breakdown=True*, *forwrev=None*, **args*, ***kwargs*)

The method for running the automatic analysis.

Parameters

- **skiptime** (*float*) – Discard data prior to this specified time as 'equilibration' data. Units are specified by the corresponding MD Engine. Default: 0.
- **uncorr** (*str*) – The observable to be used for the autocorrelation analysis; 'dE'.
- **threshold** (*int*) – Proceed with correlated samples if the number of uncorrelated samples is found to be less than this number. If 0 is given, the time series analysis will not be performed at all. Default: 50.
- **estimators** (*str or list of str*) – A list of the estimators to estimate the free energy with. Default: *('MBAR', 'BAR', 'TI')*.
- **overlap** (*str*) – The filename for the plot of overlap matrix. Default: 'O_MBAR.pdf'.
- **breakdown** (*bool*) – Plot the free energy differences evaluated for each pair of adjacent states for all methods, including the dH/dlambda curve for TI. Default: True.
- **forwrev** (*int*) – Plot the free energy change as a function of time in both directions, with the specified number of points in the time plot. The number of time points (an integer) must be provided. Specify as None will not do the convergence analysis. Default: None. By default, 'MBAR' estimator will be used for convergence analysis, as it is usually the fastest converging method. If the dataset does not contain u_nk, please run meth:*~alchemlyb.workflows.ABFE.check_convergence* manually with estimator='TI'.

summary

The summary of the free energy estimate.

Type
Dataframe

convergence

The summary of the convergence results. See *forward_backward_convergence()* for further explanation.

Type
DataFrame

update_units(*units=None*)

Update the unit.

Parameters
units (*{'kcal/mol', 'kJ/mol', 'kT'}*) – The unit used for printing and plotting results.

preprocess(*skiptime=0*, *uncorr='dE'*, *threshold=50*)

Preprocess the data by removing the equilibration time and decorrelate the date.

Parameters

- **skiptime** (*float*) – Discard data prior to this specified time as 'equilibration' data. Units are specified by the corresponding MD Engine. Default: 0.
- **uncorr** (*str*) – The observable to be used for the autocorrelation analysis; 'dE'.

- **threshold** (*int*) – Proceed with correlated samples if the number of uncorrelated samples is found to be less than this number. If 0 is given, the time series analysis will not be performed at all. Default: 50.

u_nk_sample_list

The list of u_nk after decorrelation.

Type
list

dHdl_sample_list

The list of dHdl after decorrelation.

Type
list

estimate(*estimators='MBAR', 'BAR', 'TI'*, ***kwargs*)

Estimate the free energy using the selected estimator.

Parameters

- **estimators** (*str or list of str*) – A list of the estimators to estimate the free energy with. Default: [‘TI’, ‘BAR’, ‘MBAR’].
- **kwargs** (*dict*) – Keyword arguments to be passed to the estimator.

estimator

The dictionary of estimators. The keys are in [‘TI’, ‘BAR’, ‘MBAR’]. Note that the estimators are in their original form where no unit conversion has been attempted.

Type
dict

Changed in version 2.1.0: DeprecationWarning for using analytic error for MBAR estimator.

generate_result()

Summarise the result into a dataframe.

Returns

The DataFrame with convergence data.

```
                     MBAR  MBAR_Error        BAR  BAR_Error          ␣
↪TI  TI_Error
States 0 -- 1     0.065967    0.001293   0.066544   0.001661   0.
↪066663  0.001675
       1 -- 2     0.089774    0.001398   0.089303   0.002101   0.
↪089566  0.002144
       2 -- 3     0.132036    0.001638   0.132687   0.002990   0.
↪133292  0.003055
       3 -- 4     0.116494    0.001213   0.116348   0.002691   0.
↪116845  0.002750
       4 -- 5     0.105251    0.000980   0.106344   0.002337   0.
↪106603  0.002362
       5 -- 6     0.349320    0.002781   0.343399   0.006839   0.
↪350568  0.007393
       6 -- 7     0.402346    0.002767   0.391368   0.006641   0.
↪395754  0.006961
       7 -- 8     0.322284    0.002058   0.319395   0.005333   0.
↪321542  0.005434
       8 -- 9     0.434999    0.002683   0.425680   0.006823   0.
```

(continues on next page)

(continued from previous page)

```
↪430251  0.007155
      9 -- 10    0.355672    0.002219   0.350564   0.005472   0.
↪352745  0.005591
      10 -- 11   3.574227    0.008744   3.513595   0.018711   3.
↪514790  0.018078
      11 -- 12   2.896685    0.009905   2.821760   0.017844   2.
↪823210  0.018088
      12 -- 13   2.223769    0.011229   2.188885   0.018438   2.
↪189784  0.018478
      13 -- 14   1.520978    0.012526   1.493598   0.019155   1.
↪490070  0.019288
      14 -- 15   0.911279    0.009527   0.894878   0.015023   0.
↪896010  0.015140
      15 -- 16   0.892365    0.010558   0.886706   0.015260   0.
↪884698  0.015392
      16 -- 17   1.737971    0.025315   1.720643   0.031416   1.
↪741028  0.030624
      17 -- 18   1.790706    0.025560   1.788112   0.029435   1.
↪801695  0.029244
      18 -- 19   1.998635    0.023340   2.007404   0.027447   2.
↪019213  0.027096
      19 -- 20   2.263475    0.020286   2.265322   0.025023   2.
↪282040  0.024566
      20 -- 21   2.565680    0.016695   2.561324   0.023611   2.
↪552977  0.023753
      21 -- 22   1.384094    0.007553   1.385837   0.011672   1.
↪381999  0.011991
      22 -- 23   1.428567    0.007504   1.422689   0.012524   1.
↪416010  0.013012
      23 -- 24   1.440581    0.008059   1.412517   0.013125   1.
↪408267  0.013539
      24 -- 25   1.411329    0.009022   1.419167   0.013356   1.
↪411446  0.013795
      25 -- 26   1.340320    0.010167   1.360679   0.015213   1.
↪356953  0.015260
      26 -- 27   1.243745    0.011239   1.245873   0.015711   1.
↪248959  0.015762
      27 -- 28   1.128429    0.012859   1.124554   0.016999   1.
↪121892  0.016962
      28 -- 29   1.010313    0.016442   1.005444   0.017692   1.
↪019747  0.017257
Stages coul      10.215658    0.033903  10.017838   0.041839  10.
↪017854  0.048744
      vdw       22.547489    0.098699  22.501150   0.060092  22.
↪542936  0.106723
      bonded     2.374144    0.014995   2.341631   0.005507   2.
↪363828  0.021078
      TOTAL     35.137291    0.103580  34.860619   0.087022  34.
↪924618  0.119206
```

Return type
DataFrame

summary

The summary of the free energy estimate.

Type
Dataframe

plot(**args*, ***kwargs*)

The function for producing any plots.

plot_overlap_matrix(*overlap='O_MBAR.pdf'*, *ax=None*)

Plot the overlap matrix for MBAR estimator using *plot_mbar_overlap_matrix()*.

Parameters

- **overlap** (*str*) – The filename for the plot of overlap matrix. Default: 'O_MBAR.pdf'
- **ax** (*matplotlib.axes.Axes*) – Matplotlib axes object where the plot will be drawn on. If ax=None, a new axes will be generated.

Returns
An axes with the overlap matrix drawn.

Return type
matplotlib.axes.Axes

plot_ti_dhdl(*dhdl_TI='dhdl_TI.pdf'*, *labels=None*, *colors=None*, *ax=None*)

Plot the dHdl for TI estimator using *plot_ti_dhdl()*.

Parameters

- **dhdl_TI** (*str*) – The filename for the plot of TI dHdl. Default: 'dhdl_TI.pdf'
- **labels** (*List*) – list of labels for labelling all the alchemical transformations.
- **colors** (*List*) – list of colors for plotting all the alchemical transformations. Default: ['r', 'g', '#7F38EC', '#9F000F', 'b', 'y']
- **ax** (*matplotlib.axes.Axes*) – Matplotlib axes object where the plot will be drawn on. If ax=None, a new axes will be generated.

Returns
An axes with the TI dhdl drawn.

Return type
matplotlib.axes.Axes

plot_dF_state(*dF_state='dF_state.pdf'*, *labels=None*, *colors=None*, *orientation='portrait'*, *nb=10*)

Plot the dF states using *plot_dF_state()*.

Parameters

- **dF_state** (*str*) – The filename for the plot of dF states. Default: 'dF_state.pdf'
- **labels** (*List*) – list of labels for labelling different estimators.
- **colors** (*List*) – list of colors for plotting different estimators.
- **orientation** (*string*) – The orientation of the figure. Can be *portrait* or *landscape*
- **nb** (*int*) – Maximum number of dF states in one row in the *portrait* mode

Returns
An Figure with the dF states drawn.

Return type
matplotlib.figure.Figure

check_convergence(*forwrev*, *estimator='MBAR'*, *dF_t='dF_t.pdf'*, *ax=None*, ***kwargs*)

Compute the forward and backward convergence using `plot_convergence()`.

Parameters

- **forwrev** (*int*) – Plot the free energy change as a function of time in both directions, with the specified number of points in the time plot. The number of time points (an integer) must be provided.
- **estimator** (*{'TI', 'BAR', 'MBAR'}*) – The estimator used for convergence analysis. Default: 'MBAR'
- **dF_t** (*str*) – The filename for the plot of convergence. Default: 'dF_t.pdf'
- **ax** (*matplotlib.axes.Axes*) – Matplotlib axes object where the plot will be drawn on. If `ax=None`, a new axes will be generated.
- **kwargs** (*dict*) – Keyword arguments to be passed to the estimator.

convergence

Type
DataFrame

Returns
An axes with the convergence drawn.

Return type
matplotlib.axes.Axes

3.9 Miscellaneous

This page includes aspects that would improve your usage of **alchemlyb**.

3.9.1 Logging

In **alchemlyb**, we use `loguru` for logging. By default, the `loguru` will print the logging information into the sys.stderr.

Print to the stderr

If you want to customise the printing to the *stderr*, you could remove the existing sink first

```
from loguru import logger
logger.remove()
```

Then add other custom sink

```
logger.add(sys.stderr, format="{time} {level} {message}", level="INFO")
```

The loguru logger is compatible with the `logging` module of the Python standard library and can easily be configured to log to a logging handler.

Save to a file

Alternatively, one could save to a file simply with

```
from loguru import logger
logger.add("file_{time}.log")
```

See configure to log to a file for more explanation.

3.10 References

3.11 API principles

The following is an overview over the guiding principles and ideas that underpin the API of alchemlyb.

3.11.1 *alchemlyb*

alchemlyb is a library that seeks to make doing alchemical free energy calculations easier and less error prone. It includes functions for parsing data from formats common to existing MD engines, subsampling these data, and fitting these data with an estimator to obtain free energies. These functions are simple in usage and pure in scope, and can be chained together to build customized analyses of data. General and robust workflows following best practices are also provided, which can be used as reference implementations and examples.

alchemlyb seeks to be as boring and simple as possible to enable more complex work. Its components allow work at all scales, from use on small systems using a single workstation to larger datasets that require distributed computing using libraries such as dask.

First and foremost, scientific code must be *correct* and we try to ensure this requirement by following best software engineering practices during development, close to full test coverage of all code in the library, and providing citations to published papers for included algorithms. We use a curated, public data set (alchemtest) for automated testing.

3.11.2 Core philosophy

1. Use functions when possible, classes only when necessary (or for estimators, see (2)).
2. For estimators, mimic the **scikit-learn** API as much as possible.
3. Aim for a consistent interface throughout, e.g. all parsers take similar inputs and yield a common set of outputs.
4. Have all functionality tested.

3.11.3 API components

The library is structured as follows, following a similar style to **scikit-learn**:

```
alchemlyb
├── parsing
│   ├── amber.py
│   ├── gmx.py
│   ├── gomc.py
│   ├── namd.py
```

(continues on next page)

(continued from previous page)

```
│   └── ...
├── preprocessing
│   ├── subsampling.py
│   └── ...
├── estimators
│   ├── bar_.py
│   ├── mbar_.py
│   ├── ti_.py
│   └── ...
├── convergence
│   ├── convergence.py
│   └── ...
├── postprocessors
│   ├── ...
│   └── units.py
├── visualisation
│   ├── convergence.py
│   ├── dF_state.py
│   ├── mbar_matrix.py
│   ├── ti_dhdl.py
│   └── ...
└── workflows           ### WORK IN PROGRESS
    ├── base.py
    ├── abfe.py
```

- The *parsing* submodule contains parsers for individual MD engines, since the output files needed to perform alchemical free energy calculations vary widely and are not standardized. Each module at the very least provides an *extract_u_nk* function for extracting reduced potentials (needed for MBAR), as well as an *extract_dHdl* function for extracting derivatives required for thermodynamic integration. Moreover an *extract* ` functon is provided, which returns a dict containing both derivatives and reduced potentials. Other helper functions may be exposed for additional processing, such as generating an XVG file from an EDR file in the case of GROMACS. All *extract_** functions take similar arguments (a file path, parameters such as temperature), and produce standard outputs (`pandas.DataFrame` for reduced potentials, `pandas.Series` for derivatives).
- The *preprocessing* submodule features functions for subsampling timeseries, as may be desired before feeding them to an estimator. So far, these are limited to *slicing*, *statistical_inefficiency*, and *equilibrium_detection* functions, many of which make use of subsampling schemes available from `pymbar`. These functions are written in such a way that they can be easily composed as parts of complex processing pipelines.
- The *estimators* module features classes *a la* **scikit-learn** that can be initialized with parameters that determine their behavior and then "trained" on a *fit* method. MBAR, BAR, and thermodynamic integration (TI) as the major methods are all implemented. Correct error estimates require the use of time series with independent samples.
- The *convergence* submodule features convenience functions/classes for doing convergence analysis using a given dataset and a chosen estimator.
- The *postprocessors* submodule contains functions to calculate new quantities or express data in different units.
- The *visualisation* submodule contains convenience plotting functions as known from, for example, alchemical-analysis.py.
- **The `workflows` submodule contains complete analysis workflows . . .**
  For example, `alchemlyb.workflows.abfe` implements a complete absolute binding free energy calculation.".

All of these components lend themselves well to writing clear and flexible pipelines for processing data needed for alchemical free energy calculations, and furthermore allow for scaling up via libraries like dask or joblib.

3.11.4 Development model

This is an open-source project, the hope of which is to produce a library with which the community is happy. To enable this, the library will be a community effort. Development is done in the open on GitHub. Software engineering best-practices will be used throughout, including continuous integration testing via Travis CI, up-to-date documentation, and regular releases.

Following discussion, refinement, and consensus on this proposal, issues for each need will be posted and work will begin on filling out the rest of the library. In particular, parsers will be crowdsourced from the existing community and refined into the consistent form described above.

3.11.5 Historical notes

Some of the components were originally demoed in gist a41e5756a58e1775e3e3a915f07bfd37.

David Dotson (@dotsdl) started the project while employed as a software engineer by Oliver Beckstein (@orbeckst), and this project was a primary point of focus for him in this position.

BIBLIOGRAPHY

[Bennett1976] C. H. Bennett. (1976). Efficient estimation of free energy differences from Monte Carlo data. J Comp Phys 22, 245–268. doi: 10.1016/0021-9991(76)90078-4.

[Shirts2008] M. R. Shirts and J. D. Chodera. (2008). Statistically optimal analysis of samples from multiple equilibrium states. J Chem Phys 129, 124105. doi: 10.1063/1.2978177.

[Klimovich2015] Klimovich, P.V., M. R. Shirts, and D. L. Mobley. (2015) Guidelines for the analysis of free energy calculations. Journal of Computer-Aided Molecular Design 29, 397-411. doi: 10.1007/s10822-015-9840-9.

[Chodera2016] J. D. Chodera. (2016). A simple method for automated equilibration detection in molecular simulations. Journal of Chemical Theory and Computation 12, 1799-1805. doi: 10.1021/acs.jctc.5b00784.

[He2020] X. He, S. Liu, T.-S. Lee, B. Ji, V. H. Man, D. M. York, J. Wang. (2020). Fast, Accurate, and Reliable Protocols for Routine Calculations of Protein–Ligand Binding Affinities in Drug Design Projects Using AMBER GPU-TI with Ff14SB/GAFF. ACS Omega 5, 4611-4619. doi: 10.1021/acsomega.9b04233.

[Fan2020] Fan, S., B. I. Iorga, and O. Beckstein. (2020). Prediction of octanol-water partition coefficients for the SAMPL6-log P molecules using molecular dynamics simulations with OPLS-AA, AMBER and CHARMM force fields. Journal of Computer-Aided Molecular Design 34, 543–560. doi:10.1007/s10822-019-00267-z.

[Fan2021] Fan, S., Nedev, H., Vijayan, R., Iorga, B.I., and Beckstein, O. (2021). Precise force-field-based calculations of octanol-water partition coefficients for the SAMPL7 molecules. Journal of Computer-Aided Molecular Design 35, 853–887. doi: 10.1007/s10822-021-00407-4.

PYTHON MODULE INDEX

a

INDEX

F

G

K

L

M

O

P

R

S

T

U

W