alchemlyb Documentation

Release 0.5.1+0.gd0172d1.dirty

David Dotson, Ian Kenney, Oliver Beckstein, Shuai Liu, Travis Jen

Sep 17, 2021

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* and the API is still somewhat in flux. 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. See *Contributing* 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

CONTRIBUTING

Contributions are very welcome. If you have bug reports or feature requests or questions then please get in touch with us through the Issue Tracker. We also welcome code contributions: have a look at our Developer Guide and submit a pull request against the alchemistry/alchemlyb GitHub repository.

3.1 Installing alchemlyb

alchemlyb is pure-Python, so it can be installed easily via `pip`:

```
pip install alchemlyb
```

If you wish to install this in your user `site-packages`, use the `--user` flag:

```
pip install --user alchemlyb
```

3.1.1 Installing from source

from source. Clone the source from GitHub with:

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

then do:

```
cd alchemlyb
pip install .
```

If you wish to install this in your user `site-packages`, use the `--user` flag:

```
pip install --user .
```

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*.

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).

`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
 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.

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.

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*)

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.

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.

alchemlyb.parsing.gmx.**extract_u_nk**(*xvg*, *T*)

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.

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.

Amber parsing

Parsers for extracting alchemical data from Amber output files.

Most of the file parsing parts are inherited from alchemical-analysis.

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.

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_file*, *T*)

Return reduced potentials *u_nk* from NAMD fepout file.

Parameters

- **fep_file** (*str*) – Path to fepout 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.

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.

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.

API Reference

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

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.subsampleCorrelatedData()`.
- **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.statisticalInefficiency` detailed background

`pymbar.timeseries.subsampleCorrelatedData` used for subsampling

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

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

Subsample a DataFrame using automated equilibrium detection on a timeseries.

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.

Returns *df* subsampled according to subsampled *series*.

Return type DataFrame

See also:

`pymbar.timeseries.detectEquilibration` detailed background

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

>>> 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_BT 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
```

List of TI-based estimators

TI([verbose])	Thermodynamic integration (TI).

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

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.

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

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

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 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='hybr'*, *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="hybr"*) – The optimization routine to use. This can be any of the methods available via scipy.optimize.minimize() or scipy.optimize.root().
- **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

theta_

The theta matrix.

Type DataFrame

states_

Lambda states for which free energy differences were obtained.

Type list

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.

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

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

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 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

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_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_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 Tools for postprocessing

Tools are available for postprocessing the dataframes.

3.5.1 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
>>> 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
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.

3.5.2 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.

3.6 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.

3.6.1 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(forward, forward_error, ...)	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='kT'*, *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 label for the unit of the estimate. Default: “kT”
- **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 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='kT'*, *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. Default: "kT"
- **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 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 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**(*forward*, *forward_error*, *backward*, *backward_error*, *units='kT'*, *ax=None*)

Plot the forward and backward convergence.

Parameters

- **forward** (*List*) – A list of free energy estimate from the first X% of data.
- **forward_error** (*List*) – A list of error from the first X% of data.
- **backward** (*List*) – A list of free energy estimate from the last X% of data.

- **backward_error** (*List*) – A list of error from the last X% of data.
- **units** (*str*) – The label for the unit of the estimate. Default: "kT"
- **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 forward and backward convergence drawn.

Return type matplotlib.axes.Axes

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

The units variable is for labelling only. Changing it doesn't change the unit of the underlying variable.

New in version 0.4.0.

3.6.2 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

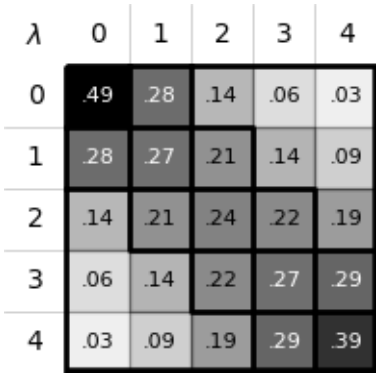

Fig. 1: Overlap between the distributions of potential energy differences is essential for accurate free energy calculations and can be quantified by computing the overlap matrix . Its elements are the probabilities of observing a sample from state i (th row) in state j (th column).

3.6.3 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

>>> 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

3.6.4 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)
>>> mbar_vdw = MBAR().fit(u_nk_vdw)

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

(continues on next page)

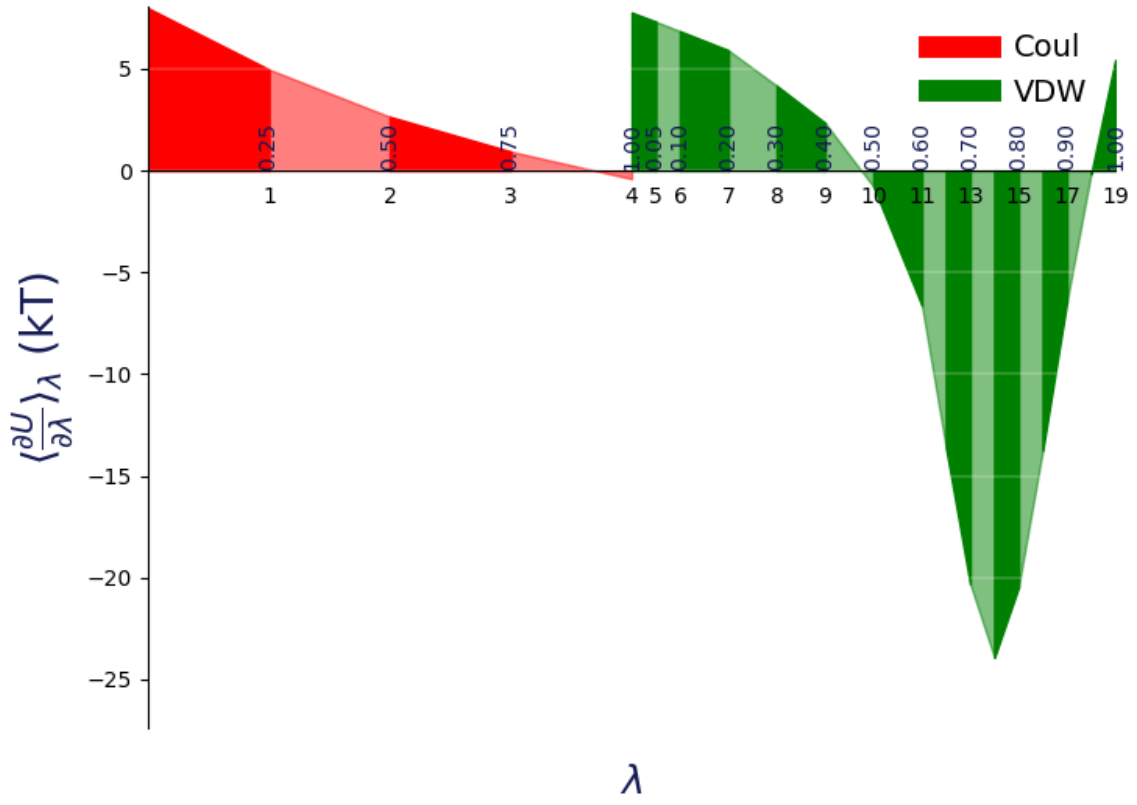

Fig. 2: A plot of / versus for thermodynamic integration, with filled areas indicating free energy estimates from the trapezoid rule. Different components are shown in distinct colors: in red is the electrostatic component (indices 0–4), while in green is the van der Waals component (indices 5–19). Color intensity alternates with increasing index.

(continued from previous page)

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

Will give a plot looks like this

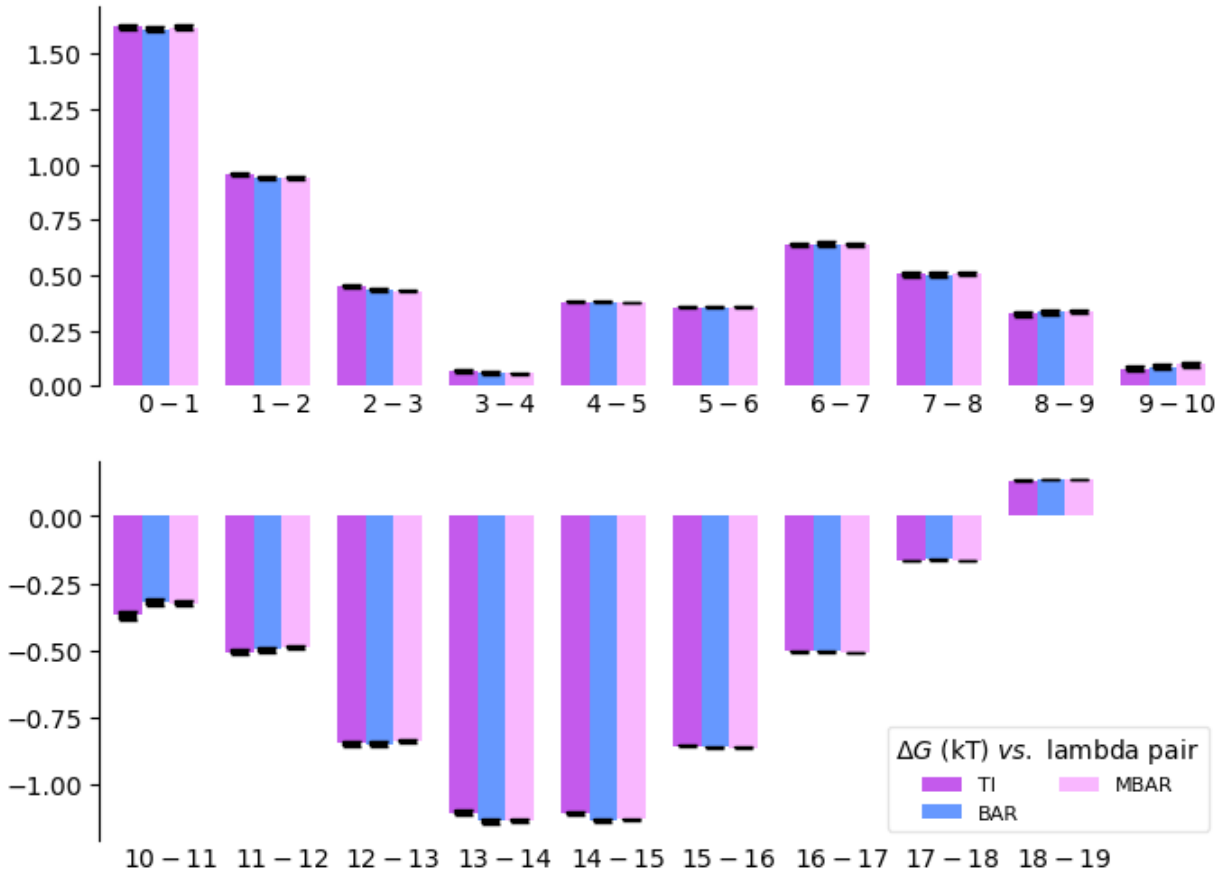

Fig. 3: A bar plot of the free energy differences evaluated between pairs of adjacent states via several methods, with corresponding error estimates for each method.

3.6.5 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
```

(continues on next page)

(continued from previous page)

```
>>> 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

3.7 API principles

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

3.7.1 *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.

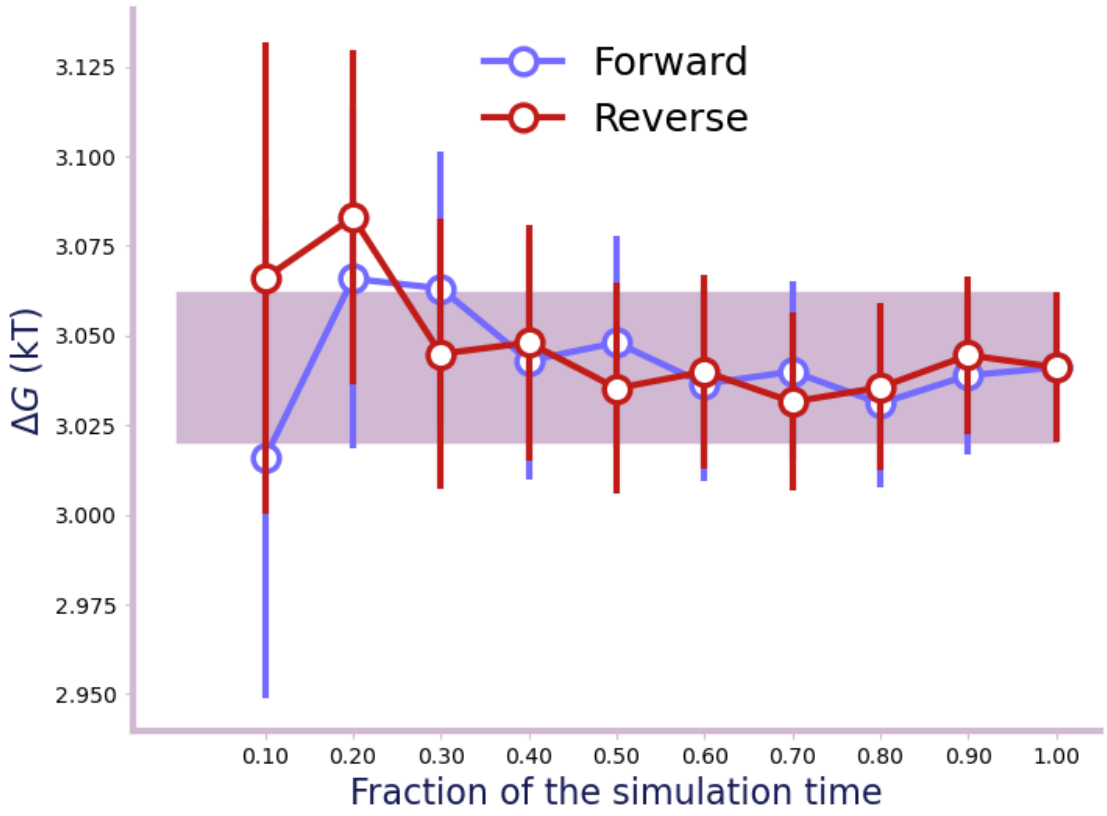

Fig. 4: A convergence plot of showing that the forward and backward has converged fully.

3.7.2 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.

3.7.3 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
│   └── ...
├── preprocessing
│   ├── subsampling.py
│   └── ...
├── estimators
│   ├── bar_.py
│   ├── mbar_.py
│   ├── ti_.py
│   └── ...
├── convergence          ### NOT IMPLEMENTED
│   ├── convergence.py
│   └── ...
├── postprocessors
│   ├── ...
│   └── units.py
├── visualisation
│   ├── convergence.py
│   ├── dF_state.py
│   ├── mbar_matrix.py
│   ├── ti_dhdl.py
│   └── ...
└── workflows
    └── ...
```

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. 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 will feature convenience functions/classes for doing convergence analysis using a given dataset and a chosen estimator, though the form of this is not yet thought-out. However, the gist a41e5756a58e1775e3e3a915f07bfd37 shows an example for how this can be done already in practice.

The `postprocessing` submodule contains functions to calculate new quantities or express data in different units.

The `visualization` submodule contains convenience plotting functions as known from, for example, alchemical-analysis.py.

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.

3.7.4 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.

3.7.5 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

[Klimovich2015] Klimovich, P.V., Shirts, M.R. & Mobley, D.L. Guidelines for the analysis of free energy calculations. J Comput Aided Mol Des 29, 397–411 (2015). https://doi.org/10.1007/s10822-015-9840-9

PYTHON MODULE INDEX

a

INDEX

R

S

T