alchemlyb Documentation

Release 0.4.1+0.g3ae6668.dirty

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

Jun 02, 2021

USER DOCUMENTATION

1 Core philosophy 3

2 Development model 5

3 Contributing 7

Bibliography 29

Python Module Index 31

Index 33

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

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 Travis CI, up-to-date documentation, and regular releases.

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 layout of these data structures allow for easy stacking of samples from different simulations while retaining information on where each sample came from using e.g. `pandas.concat()`.

`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

Throughout alchemlyb, energy quantities such as dHdl or u_nk are given in units of k_BT. 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.

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.

alchemlyb.parsing.gmx

Parsers for extracting alchemical data from Gromacs output files.

Functions

`extract_dHdl`(xvg, T)	Return gradients *dH/dl* from a Hamiltonian differences XVG file.
`extract_u_nk`(xvg, T)	Return reduced potentials *u_nk* from a Hamiltonian differences XVG file.

alchemlyb.parsing.amber

Parsers for extracting alchemical data from Amber output files.

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

Functions

`any_none`(sequence)	Check if any element of a sequence is None.
`convert_to_pandas`(file_datum)	Convert the data structure from numpy to pandas format
`extract_dHdl`(outfile, T)	Return gradients `dH/dl` from Amber TI outputfile.
`extract_u_nk`(outfile, T)	Return reduced potentials *u_nk* from Amber outputfile.
`file_validation`(outfile)	validate the energy output file

Classes

FEData()	A simple struct container to collect data from individual files.
SectionParser(filename)	A simple parser to extract data values from sections.

alchemlyb.parsing.namd

Parsers for extracting alchemical data from NAMD output files.

Functions

extract_u_nk(fep_file, T)	Return reduced potentials *u_nk* from NAMD fepout file.

alchemlyb.parsing.gomc

Parsers for extracting alchemical data from GOMC output files.

Functions

extract_dHdl(filename, T)	Return gradients *dH/dl* from a Hamiltonian differences free energy file.
extract_u_nk(filename, T)	Return reduced potentials *u_nk* from a Hamiltonian differences dat file.

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 alchemlyb.preprocessing.subsampling

Functions for subsampling datasets.

Functions

`equilibrium_detection`(df[, series, lower, ...])	Subsample a DataFrame using automated equilibrium detection on a timeseries.
`slicing`(df[, lower, upper, step, force])	Subsample a DataFrame using simple slicing.
`statistical_inefficiency`(df[, series, ...])	Subsample a DataFrame based on the calculated statistical inefficiency of a timeseries.

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 = pd.concat([extract_dHdl(xvg, T=300) for xvg in bz['Coulomb']])
>>> dHdl_vdw = pd.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).

alchemlyb.estimators.TI

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

__init__(*verbose=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([verbose])	Initialize self.
fit(dHdl)	Compute free energy differences between each state by integrating dHdl across lambda values.
get_params([deep])	Get parameters for this estimator.
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.
set_params(**params)	Set the parameters of this estimator.

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

(continues on next page)

(continued from previous page)

```
>>> 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).

alchemlyb.estimators.MBAR

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

__init__(*maximum_iterations=10000*, *relative_tolerance=1e-07*, *initial_f_k=None*, *method='hybr'*, *verbose=False*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([maximum_iterations, ...])	Initialize self.
`fit`(u_nk)	Compute overlap matrix of reduced potentials using multi-state Bennett acceptance ratio.
`get_params`([deep])	Get parameters for this estimator.
`predict`(u_ln)	
`set_params`(**params)	Set the parameters of this estimator.

Attributes

`overlap_matrix`	MBAR overlap matrix.

alchemlyb.estimators.BAR

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

__init__(*maximum_iterations=10000*, *relative_tolerance=1e-07*, *method='false-position'*, *verbose=False*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([maximum_iterations, ...])	Initialize self.
fit(u_nk)	Compute overlap matrix of reduced potentials using Bennett acceptance ratio.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.

3.5 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.

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.

3.5.1 alchemlyb.visualisation.plot_mbar_overlap_matrix

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.

3.5.2 alchemlyb.visualisation.plot_ti_dhdl

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.

The units variable is for labelling only. Changing it doesn’t change the unit of the underlying variable, which is in the unit of kT.

New in version 0.4.0.

3.5.3 alchemlyb.visualisation.plot_dF_state

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.

The units variable is for labelling only. Changing it doesn’t change the unit of the underlying variable, which is in the unit of kT.

New in version 0.4.0.

3.5.4 alchemlyb.visualisation.plot_convergence

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, which is in the unit of kT.

New in version 0.4.0.

3.5.5 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 = pd.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

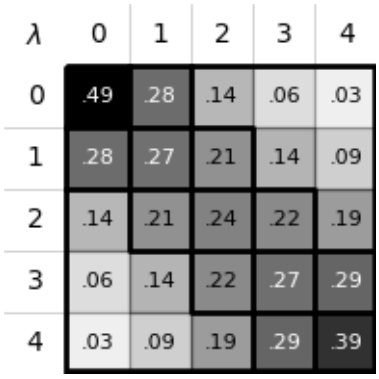

Fig. 1: Overlap between the distributions of potential energy differences is essential for accurate free energy calculations and can be quantified by computing the overlap matrix . Its elements are the probabilities of observing a sample from state i (th row) in state j (th column).

3.5.6 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 = pd.concat([extract_dHdl(xvg, T=300) for xvg in bz['Coulomb']])
>>> ti_coul = TI().fit(dHdl_coul)
>>> dHdl_vdw = pd.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

3.5.7 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
```

(continues on next page)

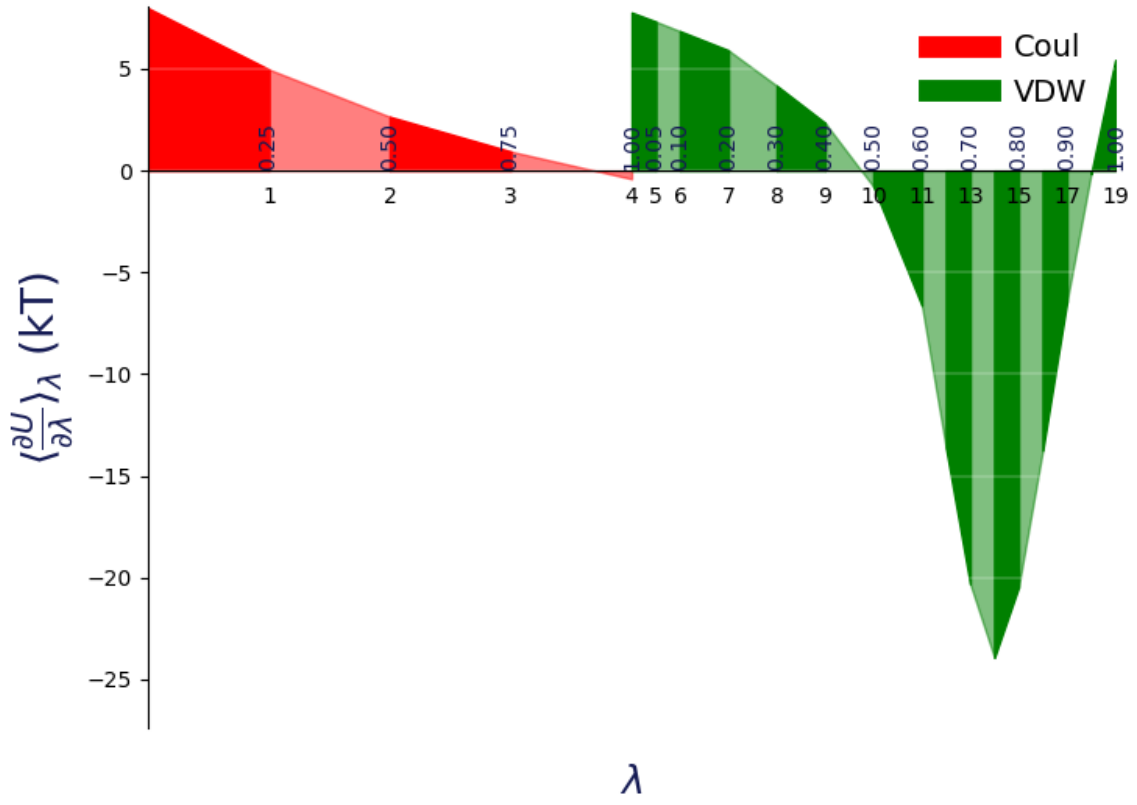

Fig. 2: A plot of / versus for thermodynamic integration, with filled areas indicating free energy estimates from the trapezoid rule. Different components are shown in distinct colors: in red is the electrostatic component (indices 0–4), while in green is the van der Waals component (indices 5–19). Color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

(continued from previous page)

```
>>> from alchemlyb.visualisation.dF_state import plot_dF_state
>>> bz = load_benzene().data
>>> u_nk_coul = pd.concat([extract_u_nk(xvg, T=300) for xvg in bz['Coulomb']])
>>> dHdl_coul = pd.concat([extract_dHdl(xvg, T=300) for xvg in bz['Coulomb']])
>>> u_nk_vdw = pd.concat([extract_u_nk(xvg, T=300) for xvg in bz['VDW']])
>>> dHdl_vdw = pd.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),]

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

Will give a plot looks like this

3.5.8 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 = pd.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 = pd.concat([data[-slice:] for data in data_list])
>>>     estimate = MBAR().fit(u_nk_coul)
```

(continues on next page)

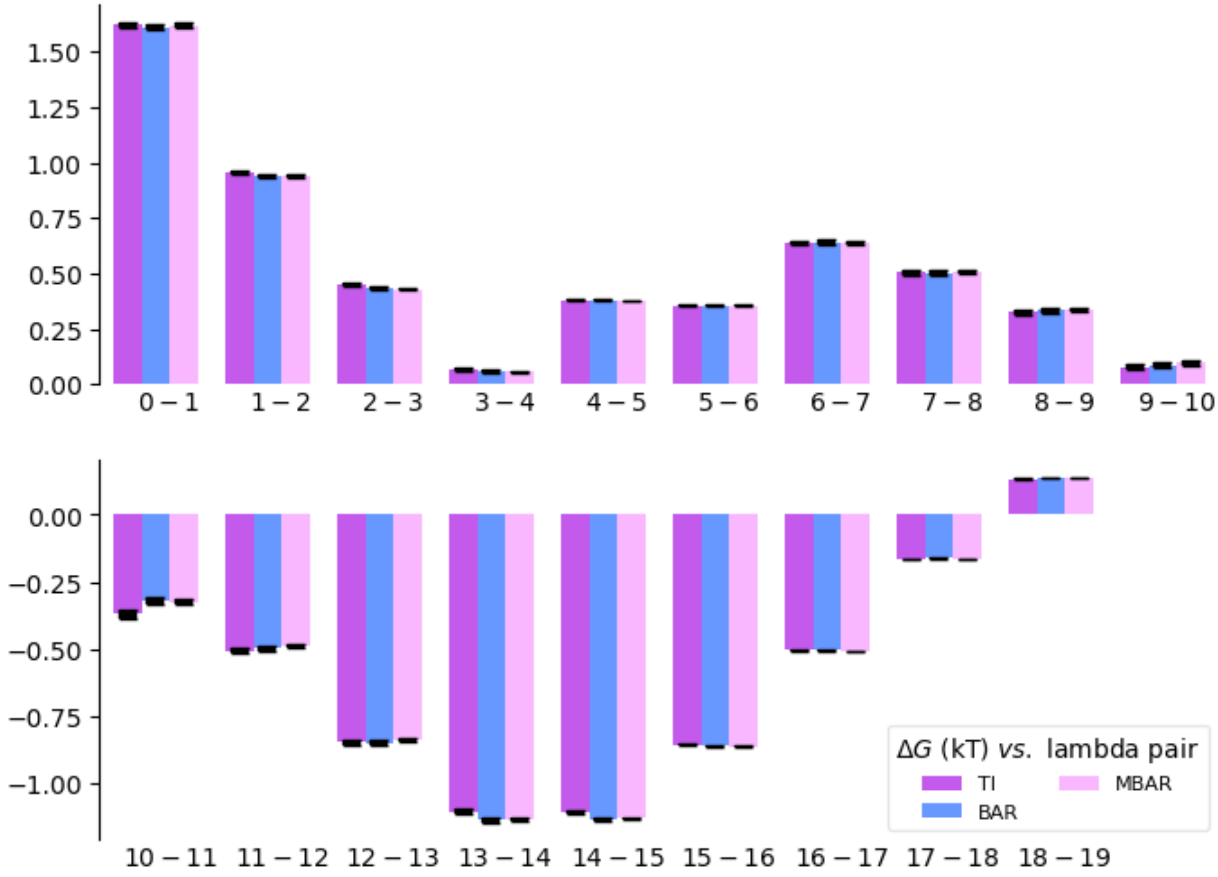

Fig. 3: A bar plot of the free energy differences evaluated between pairs of adjacent states via several methods, with corresponding error estimates for each method.

(continued from previous page)

```
>>>     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

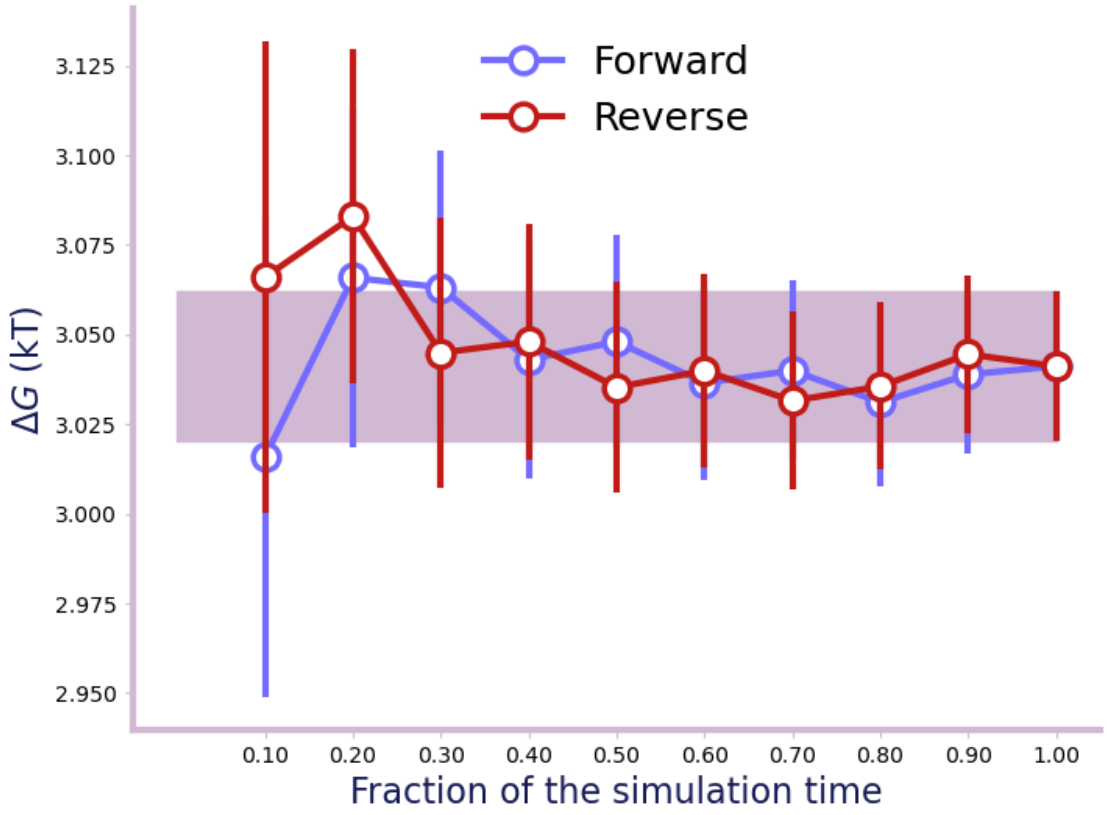

Fig. 4: A convergence plot of showing that the forward and backward has converged fully.

3.6 API principles

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

3.6.1 *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.

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.

3.6.2 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.

3.6.3 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
│   └── ...
└── visualisation
    ├── convergence.py
```

(continues on next page)

(continued from previous page)

```
    ├── dF_state.py
    ├── mbar_matrix.py
    ├── ti_dhdl.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. 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 `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.

3.6.4 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.

3.6.5 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

alchemlyb.parsing.amber, 10
alchemlyb.parsing.gmx, 10
alchemlyb.parsing.gomc, 11
alchemlyb.parsing.namd, 11
alchemlyb.preprocessing.subsampling, 11

INDEX

Symbols

__init__() (*alchemlyb.estimators.BAR method*), 17
__init__() (*alchemlyb.estimators.MBAR method*), 16
__init__() (*alchemlyb.estimators.TI method*), 14

A

alchemlyb.parsing.amber
    module, 10
alchemlyb.parsing.gmx
    module, 10
alchemlyb.parsing.gomc
    module, 11
alchemlyb.parsing.namd
    module, 11
alchemlyb.preprocessing.subsampling
    module, 11

B

BAR (*class in alchemlyb.estimators*), 17

D

d_delta_f_ (*alchemlyb.estimators.BAR attribute*), 17
d_delta_f_ (*alchemlyb.estimators.MBAR attribute*), 16
d_delta_f_ (*alchemlyb.estimators.TI attribute*), 13
delta_f_ (*alchemlyb.estimators.BAR attribute*), 17
delta_f_ (*alchemlyb.estimators.MBAR attribute*), 16
delta_f_ (*alchemlyb.estimators.TI attribute*), 13
dhdl (*alchemlyb.estimators.TI attribute*), 14

M

MBAR (*class in alchemlyb.estimators*), 16
module
    alchemlyb.parsing.amber, 10
    alchemlyb.parsing.gmx, 10
    alchemlyb.parsing.gomc, 11
    alchemlyb.parsing.namd, 11
    alchemlyb.preprocessing.subsampling, 11

P

plot_convergence() (*in module alchemlyb.visualisation*), 20
plot_dF_state() (*in module alchemlyb.visualisation*), 19
plot_mbar_overlap_matrix() (*in module alchemlyb.visualisation*), 18
plot_ti_dhdl() (*in module alchemlyb.visualisation*), 19

S

states_ (*alchemlyb.estimators.BAR attribute*), 17
states_ (*alchemlyb.estimators.MBAR attribute*), 16
states_ (*alchemlyb.estimators.TI attribute*), 14

T

theta_ (*alchemlyb.estimators.MBAR attribute*), 16
TI (*class in alchemlyb.estimators*), 13