alchemlyb Documentation

Release 0.1.0

David Dotson

May 27, 2017

User Documentation

Warning: This library is in an **alpha** state. The library and the documentation is incomplete. Use in production at your own risk.

alchemlyb is a library for doing alchemical free energy calculations more easily and with less prone for error. 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.

CHAPTER 1

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.

CHAPTER 2

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, with a Gitter channel for discussion among developers for fast turnaround on ideas.

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

2.1 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
```

2.1.1 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 .
```

2.2 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.

2.2.1 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
 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 $k_B T$. 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*.

2.2.2 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.

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
```

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.

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

Return type Series

`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

2.3 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.

2.3.1 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*)

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.

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

Return type DataFrame

See also:

`pymbar.timeseries.statisticalInefficiency()` detailed background

`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

2.4 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.

2.4.1 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 $k_B T$) between each λ window can be accessed via the `delta_f_` attribute:

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

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

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

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

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

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

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

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

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_`

DataFrame – The estimated dimensionless free energy difference between each state.

`d_delta_f_`

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

`states_`

list – Lambda states for which free energy differences were obtained.

`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`** (`boolean, optional`) – 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 mapping of string to any

`set_params`(***params*)

Set the parameters of this estimator.

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

Returns

Return type self

2.4.2 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 *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_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()
>>> 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 $k_B T$) 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 $k_B T$ 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).

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_`

DataFrame – The estimated dimensionless free energy difference between each state.

`d_delta_f_`

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

`theta_`

DataFrame – The theta matrix.

`states_`

list – Lambda states for which free energy differences were obtained.

`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** (*boolean, optional*) – 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 mapping of string to any

set_params(***params*)

Set the parameters of this estimator.

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

Returns

Return type self

2.5 API Proposal

The following is an API proposal for the library. This proposal has been prototyped, with some of the components described already implemented at a basic level. This functionality is demoed in [this gist](https://gist.github.com/dotsdl/a41e5756a58e1775e3e3a915f07bfd37).

2.5.1 *alchemlyb*

alchemlyb is a library that seeks to make doing alchemical free energy calculations easier and less error prone. It will include 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 will be 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.

2.5.2 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.

2.5.3 API components

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

```
alchemlyb
|
 -- parsing
  |   |
  |    -- gmx
  |   |
  |    -- amber
  |   |
  |    -- openmm
  |   |
  |    -- namd
  |
   -- preprocessing
  |   |
  |    -- subsampling
  |
   -- estimators
      |
       -- mbar_
      |
       -- ti_
```

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's 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. So far, *MBAR* has been partially implemented, and because the numerical heavy-lifting is already well-implemented in *pymbar.MBAR*, this class serves to give an interface that will be familiar and consistent with the others. Thermodynamic integration is not yet implemented.

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 shows an example for how this can be done already in practice.

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*.

2.5.4 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, with a Gitter channel for discussion among developers for fast turnaround on ideas. Software engineering best-practices will be used throughout, including continuous integration testing via Travis CI, up-to-date documentation, and regular releases.

David Dotson (@dotsdl) is employed as a software engineer by Oliver Beckstein (@orbeckst), and this project is a primary point of focus for him in this position. Ian Kenney (@ianmkenney) and Hannes Loeffler (@halx) have also expressed interest in direct development.

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. Expertise in ensuring theoretical correctness of each component, in particular estimators, will be needed from David Mobley (@davidmobley), John Chodera (@jchodera), and Michael Shirts (@mrshirts).

Python Module Index

a

Index