pymbar Documentation

Release 4.0.1

Levi N. Naden, Jaime Rodríguez-Guerra, Michael R. Shirts and Jo

Jul 27, 2022

CONTENTS

Python implementation of the multistate Bennett acceptance ratio (MBAR) method for estimating expectations and free energy differences

CHAPTER ONE

DOCUMENTATION

1.1 Getting started

1.1.1 Installing pymbar

This documentation covers pymbar 4. For the previous versions of pymbar, see: pymbar 3.0.7.

conda (recommended)

The easiest way to install the pymbar release is via conda:

```
$ conda install -c conda-forge pymbar
```

You can also install pymbar from the Python package index using pip:

```
$ pip install pymbar
```

Development version

The development version can be installed directly from GitHub via pip:

```
$ pip install git+https://github.com/choderalab/pymbar.git
```

In beta testing, this is way to download pymbar 4.

1.1.2 Running the tests

Running the tests is a great way to verify that everything is working.

The test suite uses pytest, in addition to statsmodels and pytables, which you can install via conda:

```
$ conda install pytest statsmodels
```

You can then run the tests from within the *pymbar* directory with:

```
$ pytest -v pymbar
```

1.2 Moving from `pymbar` version 3

Pymbar v4.0 contains several changes to improve the API longer term. This, however, breaks the API used in 3.x and previous versions.

The main changes include:

- Making various estimators return dictionaries, not tuples, making it easier to return optional information requested at call time.
- Standardizing on snake_case for function names.
- Making the built-in solvers work to have an interface closer to like `scipy` solvers.

1.2.1 Snake case

Previous version of *pymbar* had mixed cases in functions. We have standardized on snake case, and tried to make the method names that do similar things more consistent. Specific changes include:

- `getFreeEnergyDifferences` is now `compute_free_energy_differences`
- `computeExpectations` is now `compute_expectations`
- `computeMultipleExpectations` is now `compute_multiple_expectations`
- `computePerturbedFreeEnergies` is now `compute_perturbed_free_energies`
- `computeEntropyAndEnthalpy` is now `compute_entropy_and_enthalpy`

In the submodule *timeseries*:

- `statisticalInefficiency` is now `statistical_inefficiency`
- `statisticalInefficiencyMltiple` is now `statistical_inefficiency_multiple`
- `integratedAutocorrelationTime` is now `integrated_autocorrelation_time`
- `normalizedFluctuationCorrelationFunction` is now `normalized_fluctuation_correlation_function`
- `normalizedFluctuationCorrelationFunctionMultiple` is now `normalized_fluctuation_correlation_function_m`
- `subsampleCorrelatedData` is now `subsample_correlated_data`
- `detectEquilibration` is now `detect_equilibration`
- `statisticalInefficiency_fft` is now `statistical_inefficiency_fft`
- `detectEquilibration_binary_search` is now `detect_equilibration_binary_search`

Additionally, the other estimators such as the Bennett Acceptance Ratio and exponential averaging/Zwanzig equation have different, more consistent, call signatures. All other estimators are now in the `other_estimators` module.

- `BAR` is now `bar`
- `EXP` is now `exp`
- `EXPGauss` is now `exp_gauss`
- `PMF` is now `FEP` and is greatly expanded (see *Free energy surfaces with pymbar*).

1.2.2 More consistent return functionality

Previously, different pymbar functions returned different information as tuples. This became problematic when different functions returned different types of information or different numbers of results. We have thus consolidated on an API where all functions return a dictionary.

As an example of both API changes of API, a short bit of code that would load in data and calculate free energies, instead of being

Listing 1: Example of initializing `MBAR` in 3.0.7

```
mbar = MBAR(u_kn, N_k)
results, errors = mbar.getFreeEnergyDifferences()
print(results[0])
print(errors[0])
```

Would now be written as:

Listing 2: Example of initializing `MBAR` in 4.0

```
mbar = MBAR(u_kn, N_k)
results = mbar.compute_free_energy_differences()
print(results['Delta_f'])
print(results['dDelta_f'])
```

Other estimators including `bar` and `exp` also use a dictionary for return data.

The `pymbar.timeseries` submodule return patterns have *not* changed in 4.0, however, and one should refer to the individual function documentations for these return patterns.

```
results = bar(w_F, w_R)
print(f'Free energy difference is {results['Delta_f']:.3f} +- {results['Delta_f']:.3f} kT
↪')


and:
```

```
results = exp(w_F)
print(f"Forward free energy difference is {results['Delta_f']:.3f} +- {results['dDelta_f
↪']:.3f} kT)
results = exp(w_R)
print(f"Reverse free energy difference is {results['Delta_f']:.3f} +- {results['dDelta_f
↪']:.3f} kT)
```

1.2.3 Simulation output

Previously, `pymbar` send all messages to standard out when verbose was set to `True`. `pymbar` now uses the logging module to output this information. If you wish to set messages, even if the verbose is set to `True`, you will need to turn on logging for your script by importing the logging module, and adding the lines:

Listing 3: Enabling logging in pybmar

```
import logging
import sys
logging.basicConfig(stream=sys.stdout, level=logging.INFO)
```

pymbar generally uses the logging levels info for information that previously was set to standard out. Note that for a given method to produce extensive information, even with logging, the verbose flag still needs to be set to true.

1.2.4 Free energy surfaces

Previously, pymbar had a method PMF that estimated a free energy from a series of umbrella samples using a histogram approach. This was sematically problematin in two ways. First, the term PMF (potential of mean force) is somewhat of an ambiguous term, as the potential of mean force has some dependence on the coordinate system in which the mean force is calculated. Since pymbar does not calculate free energies by integration of mean force, this caused some comfusion. To be clearer, we now have renamed the class FES, for "free energy surface".

The inclusion of a PMF function also created some confusion where some authors referred to MBAR as a method to calculate a free energy surface. MBAR can only be used to take biased samples an estimate the unbiased weight of each sample. In order to calculate a free energy surface, one must also find a way to take the set of discrete weighted samples and calculate a continous potential of mean force: see Shirts and Ferguson [1] for a further discussion of the separation of these two distinct tasks in the construction of free energy surfaces. The pymbar code more cleanly separates the calculation of biasing weights associated with umbrella samples, and the estimation of the free energy surface.

For more information on the options for computing free energy surfaces with the code, please see: *Free energy surfaces with pymbar*.

1.2.5 Acceleration

Previous version of pymbar include acceleration using explict C++ inner loops. The C++ interface has become out of date. pymbar optimization routines are now accelerated with jax. This provides approximately a 2x speed up when performed on most CPUs, and additional acceleration when a GPU can be detected (pymbar does not install the appropriate GPU libraries). jax will be installed when pymbar in installed via conda, but pymbar will function with or without jax installed if there are issues with the JAX configuration.

1.2.6 Other changes

Additional changes not affecting the API:

- Removed legacy *old_mbar.py* support.
- Moved testing framework to pytest, added significant numbers of tests.
- Improved code linting using *black*
- Bootstrapping for errors in free energies and expectations is now supported; see *Strategies for solution* for more information.
- Added a *bar_overlap* function to find overlap when using just *bar*
- Fixed an error in computing expectations of small numbers.
- Improved automated adaptive choice of samplers; see *Strategies for solution* for more information.
- Many instances of code cleanup.
- Improved docstring documentation.

1.3 Strategies for solution

1.3.1 Approaches to solving the MBAR equations

The multistate reweighting approach to calculate free energies can be formulated in several ways. The multistate reweighting equations are a set of coupled, implicit equations for the free energies of K states, given samples from these K states. If one can calculate the energies of each of the K states, for each sample, then one can solve for the K free energies satisfying the equations. The solutions are unique only up to an overall constant, which `pymbar` removes by setting the first free energy to zero to 0, leaving *K-1*. free energies.

By rearrangement, this set of self-consistent equations can be written as simultaneous roots to K equations. This set of roots also turns out to be the Jacobian of single maximum likelihood function of all the free energies. We then can find the MBAR solutions by either maximization/minimization techiques, or by root finding.

Because the second derivative of the likelihood is always negative, there is only one possivle solution. However, if there is poor overlap, it is not uncommon that some of the optimal f_k could be in extremely flat region of solution space, and therefore have significant round-off errors resulting in slow or no convergence to the solution, and low overlap can also lead to underflow and overflow leading to crashed solutions.

1.3.2 `scipy` solvers

`pymbar` is set up to use the `scipy.optimize.minimize()` and `scipy.optimize.roots()` functionality to perform this minimization. We use only the gradient-based methods, as the analytical gradient-based optimization is obtainable from the MBAR equations. Available `scipy.optimize.minimize()` methods include `L-BFGS-B`, `dogleg`, `CG`, `BFGS`, `Newton-CG`, `TNC`, `trust-ncg`, `trust-krylov`, `trust-exact`, and `SLSQP`. and `scipy.optimize.roots` options are `hybr` and `lm`. Methods that take a Hessian (`dogleg`, `Newton-CG`, `trust-ncg`, `trust-krylov`, `trust-exact`) are passed the analytical Hessian. Options can be passed to each of these methods through the `MBAR` object initialization interface.

1.3.3 Built-in solutions

In addition to the `scipy` solcers, `pymbar` also includes an adaptive solver designed directly for MBAR. At every step, the adaptive solver calculates both the next iteration of the self-consistent iterative formula presented in Shirts et al. [2], and takes a Newton-Raphson step. In both cases, it calculates the gradients of the points resulting after the two steps, and selects the move that makes the magnitude of the gradient (i.e. the dot product of the gradient with itself) smallest. Far from the solution, the self-consistent iteration tends have the smaller gradient, while closer to the solution, the Newton-Raphson step tends to have the smallest gradient. It always chooses the self-consistent iteration for the first *min_sc_iter* iterations, as if overlap is poor then Newton=Raphson iteration can fail. `min_sc_iter`'s default is 2, but if one is starting from a good guess for the free energies, one could start with `min_sc_iter=0`

1.3.4 Constructing solver protocols

We have found that in general, different solvers have different convergence properties and difference convergence behavior. Even at the same input tolerance level, different algorithms may not yield the same result. Additionally, accurate free energy estimates and other. Although data with states that have significant overlap in configuration states usually converge successfully with all simulation algorithms, different algorithms succeed and fail on different "hard" data sets. We have therefore constructed a very general interface to allow the user to try different algorithms.

`pymbar` uses the concept of a "solver protocol", where one applies a series of methods one or multiple times. In most cases, one will never need to interact with this interface, because several different protocols have already been implemented, and are accessible with string keywords. These are currently `solver_protocl=default` (also the default) and `solver_protocol=robust`.

However, a user has the option of creating their own solver protocol. Solver protocols are created as tuples of dictionaries, where each dictionary is an optimization operation. The user has the ability to continue with the resulting free energies each time, or restarting back from the initialized free energies.

Take for example the default solver protocol, designed to give a high accuracy answer in most circumstances:

```
solver_protocol = (
       dict(method="hybr", continuation=True),
       dict(method="adaptive", options=dict(min_sc_iter=0)),
)
```

Here, the first pass through optimization uses the `scipy.optimize.roots` function `hybr`, and then if it is successful, continues on by running the built-in `adaptive` method, but with no self-consistent iteration choices forced, as described above. If *hybr* fails, it will still attempted to continue on with the resulting free energies, from whatever point it ended up, issue a warning in case that *adaptive* is still unable to solve the result.

The `options` dictionary is passed onto the method, so whatever options the scipy method uses it its documentation, it can be passed on through this approach. `tol` is a direct option to *scipy.optimize* methods, and not passed on through the dictionary, and thus is passed on directly in the solver protocol.

```
solver_protocol = (
        dict(method="hybr"),
        dict(method="L-BFGS-B", tol = 1.0e-5, continuation = True,␣
↪options=dict(maxiter=1000)),
        dict(method="adaptive", tol = 1.0e-12, options=dict(maxiter=1000,min_sc_iter=5))
)
```

In this case, it would first use "hybr" with the default options and tolerance and if successful, exit. If not successful with "hybr", it would continue on to "L-BFGS-B", with 1000 maximum iterations, and a tolerance of 10^{-5} but would not use the results from the "hybr" call. If "L-BFGS-B" was either successful or unsuccessful, it would pass the results to "adaptive", where it would choose the self-consistent iteration the first five times, the numberof maximum iteractions would be 1000, and the tolerance is 10^{-12}.

1.3.5 Initialization

One can initialize the soution process in a number of ways. The simplest is to start from all zeros, which is the default (and also has keyword `initialize=zeros`). If the keyword `f_k_initial` is used, then the length *K*.

Two other options for `initialize` are `BAR` and `average-enthalpies`. `average-enthalpies` which approximates the free energy of each state using the average enthalpy of each states, which will be valid in the limit of no entropy differences beween states. `initialize=BAR` can be used whenever states are given in a natural sequence of overlap, such that state 0 has the most configurational overlap with state 1, state 1 has significant configurational overlap with both states 0 and state 2, and so forth. In the limit there is only overlap between neighboring states, MBAR converges to give the same answer for $\Delta f_k = f_{k+1} - f_k$ that BAR gives. Although BAR also requires an iterative solution, it is a single variable problem, and thus the *K-1* BAR iterations that need to be done are much faster than a single *K-1* dimensional problem. The initial input for state *k* in the solution process is then $f_{k,initial} = \sum_j \Delta f_{j,BAR}$

Note that if both `initialize` and `f_k_initial` are selected, the logic is somewhat different. Specifing `f_k_initial` overwrites `initialize=zeros`, but `initialize=BAR` starts each application of BAR with the (reduced) free energy difference between states *k* and *k+1* in from `f_k_initial`.

1.3.6 Calculating uncertainties

The MBAR equations contain analytical estimates of uncertainties. These are essentially, however, the functional form is bit more complicated, since they include modifications for error propagation with implicit equations.

For free energies and expectations, one includes the analytical uncertainties by adding the keyword `compute_uncertainties=True`.

In some cases, to peform additional error analysis, one might need access to the covariance matrix of $ln f_k$. This is accessed in `results['Theta']`, and included by setting `compute_theta=True`, or if `compute_uncertainties=True` and uncertainty_method is not `bootstrap`.

If `uncertainty_method=bootstrap`, then the analytical error analysis is not performed, and instead bootstrap samples are pulled from the original distribution. Bootstrapping is done on each set of N_k samples from each *K* states individually, rather than on the set as a whole, as the number of samples drawn from each state should not change in the bootstrapping process, or it would be a different process.

For bootstrapping to be used in calculating error estimates, the `MBAR` object must be initialized with the keyword *n_bootstraps*, which must be an integer greater than zero. In general, 50–200 bootstraps should be used to estimate uncertainties with a good degree of accuracy.

Note that users have complete control over the solver sequence for bootstrapped solutions, using the same API as for solvers of the original solution, with keyword `bootstrap_solver_protocol`. As an example, the default bootstrap protocol is:

```
bootstrap_solver_protocol = (dict(method="adaptive", tol = 1e-6, options=dict(min_sc_
↪iter=0,maximum_iterations=100)))
```

The solutions for bootstrapped data should be relatively close to the solutions for the original data set; additionally, they do not need to be quite as accurate, since they are used to compute the variances.

Bootstrapped uncertainties (using `uncertainty_method=bootstrap` is also available for all functions calculating expectations, but again requires initialization with “n_bootstraps” when initalizing the MBAR object.

1.4 The `mbar` module: `MBAR`

The `mbar` module contains the *`MBAR`* class, which implements the multistate Bennett acceptance ratio (MBAR) method [2].

class `pymbar.`**`MBAR`**(*u_kn*, *N_k*, *maximum_iterations=10000*, *relative_tolerance=1e-07*, *verbose=False*, *initial_f_k=None*, *solver_protocol=None*, *initialize='zeros'*, *x_kindices=None*, *n_bootstraps=0*, *bootstrap_solver_protocol=None*, *rseed=None*)

Multistate Bennett acceptance ratio method (MBAR) for the analysis of multiple equilibrium samples.

Notes

Note that this method assumes the data are uncorrelated.

Correlated data must be subsampled to extract uncorrelated (effectively independent) samples.

References

[1] Shirts MR and Chodera JD. Statistically optimal analysis of samples from multiple equilibrium states. J. Chem. Phys. 129:124105, 2008 http://dx.doi.org/10.1063/1.2978177

Initialize multistate Bennett acceptance ratio (MBAR) on a set of simulation data.

Upon initialization, the dimensionless free energies for all states are computed. This may take anywhere from seconds to minutes, depending upon the quantity of data. After initialization, the computed free energies may be obtained by a call to `compute_free_energy_differences()`, or expectation at any state of interest can be computed by calls to `compute_expectations()`.

Parameters

- **u_kn** (*np.ndarray, float, shape=(K, N_max)*) – `u_kn[k,n]` is the reduced potential energy of uncorrelated configuration n evaluated at state `k`.
- **u_kln** (*np.ndarray, float, shape (K, L, N_max)*) – If the simulation is in form `u_kln[k,l,n]` it is converted to `u_kn` format

  ```
  u_kn = [ u_1(x_1) u_1(x_2) u_1(x_3) . . . u_1(x_n)
           u_2(x_1) u_2(x_2) u_2(x_3) . . . u_2(x_n)
                                      . . .
           u_k(x_1) u_k(x_2) u_k(x_3) . . . u_k(x_n)]
  ```

- **N_k** (*np.ndarray, int, shape=(K)*) – `N_k[k]` is the number of uncorrelated snapshots sampled from state `k`. Some may be zero, indicating that there are no samples from that state.

  We assume that the states are ordered such that the first `N_k` are from the first state, the 2nd `N_k` the second state, and so forth. This only becomes important for bar – MBAR does not care which samples are from which state. We should eventually allow this assumption to be overwritten by parameters passed from above, once `u_kln` is phased out.

- **maximum_iterations** (*int, optional*) – Set to limit the maximum number of iterations performed (default 1000)
- **relative_tolerance** (*float, optional*) – Set to determine the relative tolerance convergence criteria (default 1.0e-6)
- **verbosity** (*bool, optional*) – Set to True if verbose debug output is desired (default False)
- **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)
- **solver_protocol** (*list(dict), string or None, optional, default=None*) – List of dictionaries to define a sequence of solver algorithms and options used to estimate the dimensionless free energies. See *pymbar.mbar_solvers.solve_mbar()* for details. If None, use the developers best guess at an appropriate algorithm.

  if the string is “robust”, it tries “L-BFGS-B” and “adaptive”, with relatively large numbers of iterations.

  if “default” or “none”, it will use ‘hybr’ root solver, followed with some rounds of Newton-Raphson if it fails to converge.

  The default will try to solve with an adaptive solver algorithm which alternates between self-consistent iteration and Newton-Raphson, where the method with the smallest gradient is chosen to improve numerical stability.

- **initialize** (*'zeros' or 'BAR', optional, Default: 'zeros'*) – If equal to 'BAR', use bar between the pairwise state to initialize the free energies. Eventually, should specify a path; for now, it just does it zipping up the states.

  The 'BAR' option works best when the states are ordered such that adjacent states maximize the overlap between states. It is up to the user to arrange the states in such an order, or at least close to such an order. If you are uncertain what the order of states should be, or if it does not make sense to think of states as adjacent, then choose 'zeros'.

- **x_kindices** (*np.ndarray, int, shape=(K), default = [0,1,2,3,4...K]*) – Describes which state is each sample *x* is from? Usually doesn't matter, but it does for bar. As a default, we assume the samples are in `K` order (the first `N_k[0]` samples are from the 0th state, the next `N_k[1]` samples from the 1st state, and so forth.
- **n_bootstraps** (*int*) – How many bootstrap free energies will be computed? If None, no bootstraps will be computed. computing uncertainties with bootstraps is only possible if this is > 0. (default: None)
- **bootstrap_solver_protocol** (*list(dict), string or None, optional, default=None*) – We usually just do steps of adaptive sampling without. "robust" would be the backup. Default: dict(method="adaptive", options=dict(min_sc_iter=0)),

Notes

The reduced potential energy `u_kn[k,n] = u_k(x_{ln})`, where the reduced potential energy `u_l(x)` is defined (as in the text) by: `u_k(x) = beta_k [ U_k(x) + p_k V(x) + mu_k' n(x) ]` where

`beta_k = 1/(k_B T_k)` is the inverse temperature of condition `k`, where k_B is Boltzmann's constant

`U_k(x)` is the potential energy function for state `k`

`p_k` is the pressure at state `k` (if an isobaric ensemble is specified)

`V(x)` is the volume of configuration `x`

`mu_k` is the M-vector of chemical potentials for the various species, if a (semi)grand ensemble is specified, and `'` denotes transpose

`n(x)` is the M-vector of numbers of the various molecular species for configuration `x`, corresponding to the chemical potential components of `mu_m`.

`x_n` indicates that the samples are from `k` different simulations of the `n` states. These simulations need only be a subset of the k states.

The configurations `x_ln` must be uncorrelated. This can be ensured by subsampling a correlated timeseries with a period larger than the statistical inefficiency, which can be estimated from the potential energy timeseries `{u_k(x_ln)}_{n=1}^{N_k}` using the provided utility *pymbar.timeseries.statistical_inefficiency()*. See the help for this function for more information.

Examples

```
>>> from pymbar import testsystems
>>> (x_n, u_kn, N_k, s_n) = testsystems.HarmonicOscillatorsTestCase().sample(mode=
↪'u_kn')
>>> mbar = MBAR(u_kn, N_k)
```

property W_nk

Retrieve the weight matrix W_nk from the MBAR algorithm.

Necessary because they are stored internally as log weights.

Returns
weights – NxK matrix of weights in the MBAR covariance and averaging formulas

Return type
np.ndarray, float, shape=(N, K)

compute_covariance_of_sums(*d_ij*, *K*, *a*)

We wish to calculate the variance of a weighted sum of free energy differences. for example `var(\sum a_i df_i)`.

We explicitly lay out the calculations for four variables (where each variable is a logarithm of a partition function), then generalize.

The uncertainty in the sum of two weighted differences is

```
var(a1(f_i1 - f_j1) + a2(f_i2 - f_j2)) =
    a1^2 var(f_i1 - f_j1) +
    a2^2 var(f_i2 - f_j2) +
    2 a1 a2 cov(f_i1 - f_j1, f_i2 - f_j2)
cov(f_i1 - f_j1, f_i2 - f_j2) =
    cov(f_i1,f_i2) -
    cov(f_i1,f_j2) -
    cov(f_j1,f_i2) +
    cov(f_j1,f_j2)
```

call:

```
f_i1 = a
f_j1 = b
f_i2 = c
f_j2 = d
a1^2 var(a-b) + a2^2 var(c-d) + 2a1a2 cov(a-b,c-d)
```

we want `2cov(a-b,c-d) = 2cov(a,c)-2cov(a,d)-2cov(b,c)+2cov(b,d)`, since `var(x-y) = var(x) + var(y) - 2cov(x,y)`, then, `2cov(x,y) = -var(x-y) + var(x) + var(y)`. So, we get

```
2cov(a,c) = -var(a-c) + var(a) + var(c)
-2cov(a,d) = +var(a-d) - var(a) - var(d)
-2cov(b,c) = +var(b-c) - var(b) - var(c)
2cov(b,d) = -var(b-d) + var(b) + var(d)
```

adding up, finally :

```
2cov(a-b,c-d) = 2cov(a,c)-2cov(a,d)-2cov(b,c)+2cov(b,d) =
    - var(a-c) + var(a-d) + var(b-c) - var(b-d)

a1^2 var(a-b)+a2^2 var(c-d)+2a1a2cov(a-b,c-d) =
    a1^2 var(a-b)+a2^2 var(c-d)+a1a2 [-var(a-c)+var(a-d)+var(b-c)-var(b-d)]

var(a1(f_i1 - f_j1) + a2(f_i2 - f_j2)) =
    a1^2 var(f_i1 - f_j1) + a2^2 var(f_i2 - f_j2) + 2a1 a2 cov(f_i1 - f_j1, f_
↪i2 - f_j2)
= a1^2 var(f_i1 - f_j1) + a2^2 var(f_i2 - f_j2) + a1 a2 [-var(f_i1 - f_i2) +␣
↪var(f_i1 - f_j2) + var(f_j1-f_i2) - var(f_j1 - f_j2)]
```

assume two arrays of free energy differences, and and array of constant vectors a. we want the variance `var(\sum_k a_k (f_i,k - f_j,k))` Each set is separated from the other by an offset K same process applies with the sum, with the single var terms and the pair terms

Parameters

- **d_ij** (*a matrix of standard deviations of the quantities f_i - f_j*) –
- **K** (*The number of states in each 'chunk', has to be constant*) –
- **outputs** (KxK variance matrix for the sums or differences `\sum a_i df_i`) –

compute_effective_sample_number(*verbose=False*)

Compute the effective sample number of each state; essentially, an estimate of how many samples are contributing to the average at given state. See pymbar/examples for a demonstration.

It also counts the efficiency of the sampling, which is simply the ratio of the effective number of samples at a given state to the total number of samples collected. This is printed in verbose output, but is not returned for now.

Returns

N_eff – estimated number of samples contributing to estimates at each state i. An estimate to how many samples collected just at state i would result in similar statistical efficiency as the MBAR simulation. Valid for both sampled states (in which the weight will be greater than N_k[i], and unsampled states.

Return type

np.ndarray, float, shape=(K)

Parameters

verbose (*print out information about the effective number of samples*) –

Notes

Using Kish (1965) formula (Kish, Leslie (1965). Survey Sampling. New York: Wiley)

As the weights become more concentrated in fewer observations, the effective sample size shrinks. from http://healthcare-economist.com/2013/08/22/effective-sample-size/

```
effective number of samples contributing to averages carried out at state i
    =  (\sum_{n=1}^N w_in)^2 / \sum_{n=1}^N w_in^2
    =  (\sum_{n=1}^N w_in^2)^-1
```

the effective sample number is most useful to diagnose when there are only a few samples contributing to the averages.

Examples

```
>>> from pymbar import testsystems
>>> [x_kn, u_kln, N_k, s_n] = testsystems.HarmonicOscillatorsTestCase().sample()
>>> mbar = MBAR(u_kln, N_k)
>>> N_eff = mbar.compute_effective_sample_number()
```

compute_entropy_and_enthalpy(*u_kn=None*, *uncertainty_method=None*, *verbose=False*, *warning_cutoff=1e-10*)

Decompose free energy differences into enthalpy and entropy differences.

Compute the decomposition of the free energy difference between states 1 and N into reduced free energy differences, reduced potential (enthalpy) differences, and reduced entropy (S/k) differences.

Parameters

- **u_kn** (*float, NxK array*) – The energies of the state that are being used.
- **uncertainty_method** (*str , optional*) – Choice of method used to compute asymptotic covariance method, or None to use default See help for computeAsymptoticCovarianceMatrix() for more information on various methods. if method = "bootstrap" then uncertainty over bootstrap samples is used. with bootstraps. (default: None)
- **warning_cutoff** (*float, optional*) – Warn if squared-uncertainty is negative and larger in magnitude than this number (default: 1.0e-10)

Returns

- *Results dictionary with the following keys*
- **'Delta_f'** (*np.ndarray, float, shape=(K, K)*) – results['Delta_f'] is the dimensionless free energy difference f_j - f_i
- **'dDelta_f'** (*np.ndarray, float, shape=(K, K)*) – uncertainty in results['Delta_f']
- **'Delta_u'** (*np.ndarray, float, shape=(K, K)*) – results['Delta_u'] is the reduced potential energy difference u_j - u_i
- **'dDelta_u'** (*np.ndarray, float, shape=(K, K)*) – uncertainty in results['Delta_u']
- **'Delta_s'** (*np.ndarray, float, shape=(K, K)*) – results['Delta_s'] is the reduced entropy difference S/k between states i and j (s_j - s_i)
- **'dDelta_s'** (*np.ndarray, float, shape=(K, K)*) – uncertainty in results['Delta_s']

Examples

```
>>> from pymbar import testsystems
>>> x_n, u_kn, N_k, s_n = testsystems.HarmonicOscillatorsTestCase().sample(mode=
↪'u_kn')
>>> mbar = MBAR(u_kn, N_k)
>>> results = mbar.compute_entropy_and_enthalpy()
```

compute_expectations(*A_n*, *u_kn=None*, *output='averages'*, *state_dependent=False*, *compute_uncertainty=True*, *uncertainty_method=None*, *warning_cutoff=1e-10*, *return_theta=False*)

Compute the expectation of an observable of a phase space function.

Compute the expectation of an observable of a single phase space function A(x) at all states where potentials are generated.

Parameters

- **A_n** (*np.ndarray, float*) – A_n (N_max np float64 array) - A_n[n] = A(x_n)
- **u_kn** (*np.ndarray*) – u_kn (energies of state of interest length N) default is self.u_kn
- **output** (*string, optional*) – 'averages' outputs expectations of observables and 'differences' outputs a matrix of differences in the observables.
- **compute_uncertainty** (*bool, optional*) – If False, the uncertainties will not be computed (default : True)
- **uncertainty_method** (*string, optional*) – Choice of method used to compute asymptotic covariance method, or None to use default See help for _computeAsymptoticCovarianceMatrix() for more information on various methods. if uncertainty_method = "bootstrap", then uncertainty over bootstrap samples is used. (default: None)
- **warning_cutoff** (*float, optional*) – Warn if squared-uncertainty is negative and larger in magnitude than this number (default: 1.0e-10)
- **state_dependent** (*bool, whether the expectations are state-dependent.*) –

Returns

- *Results dictionary with the following keys*
- **'mu'** (*np.ndarray, float*) – if output is 'averages' A_i (K np float64 array) - A_i[i] is the estimate for the expectation of A(x) for state i. if output is 'differences'
- **'sigma'** (*np.ndarray, float*) – dA_i (K np float64 array) - dA_i[i] is uncertainty estimate (one standard deviation) for A_i[i] or dA_ij (K np float64 array) - dA_ij[i,j] is uncertainty estimate (one standard deviation) for the difference in A beteen i and j or None, if compute_uncertainty is False.
- **'Theta'** (*((KxK np float64 array): Covariance matrix of log weights*)

References

See Section IV of [1].

Examples

```
>>> from pymbar import testsystems
>>> (x_n, u_kn, N_k, s_n) = testsystems.HarmonicOscillatorsTestCase().
↪sample(mode='u_kn')
>>> mbar = MBAR(u_kn, N_k)
>>> A_n = x_n
>>> results = mbar.compute_expectations(A_n)
>>> A_n = u_kn[0,:]
>>> results = mbar.compute_expectations(A_n, output='differences')
```

compute_expectations_inner(*A_n, u_ln, state_map, uncertainty_method=None, warning_cutoff=1e-10, return_theta=False*)

Compute the expectations of multiple observables of phase space functions in multiple states.

Compute the expectations of multiple observables of phase space functions [A_0(x),A_1(x),...,A_i(x)] along with the covariances of their estimates at multiple states.

Intended as an internal function to keep all the optimized and robust expectation code in one place, but will leave it open to allow for later modifications and uses.

It calculates all input observables at all states which are specified by the list of states in the state list.

Parameters

- `A_n` (*np.ndarray, float, shape=(I, N)*) – A_in[i,n] = A_i(x_n), the value of phase observable i for configuration n
- `u_ln` (*np.ndarray, float, shape=(L, N)*) – u_ln[l,n] is the reduced potential of configuration n at state l if u_ln = None, we use self.u_kn
- `state_map` (*np.ndarray, int, shape (2,NS) or shape(1,NS)*) – If state_map has only one dimension where NS is the total number of states we want to simulate things a. The list will be of the form `[[0,1,2],[0,1,1]]`. This particular example indicates we want to output the properties of three observables total: the first property A[0] at the 0th state, the 2nd property A[1] at the 1th state, and the 2nd property A[1] at the 2nd state. This allows us to tailor our output to a large number of different situations.
- `uncertainty_method` (*string, optional*) – Choice of method used to compute asymptotic covariance method, or None to use default See help for _computeAsymptoticCovarianceMatrix() for more information on various methods. The exception is the "bootstrap" option, which required n_boostraps being defined at initialization of MBAR. (default: None)
- `warning_cutoff` (*float, optional*) – Warn if squared-uncertainty is negative and larger in magnitude than this number (default: 1.0e-10)
- `return_theta` (*bool, optional*) – Whether or not to return the theta matrix. Can be useful for complicated differences of observables.

Returns

- **result_vals** (*dictionary*)
- *Keys in the result_vals dictionary*
- **'observables'** (*np.ndarray, float, shape = (S)*) – results_vals['observables'][i] is the estimate for the expectation of A_state_map[i](x) at the state specified by u_n[state_map[i],:]
- **'Theta'** (*np.ndarray, float, shape = (K+len(state_list), K+len(state_list)) the covariance matrix of log weights.*)
- **'Amin'** (*np.ndarray, float, shape = (S), needed for reconstructing the covariance one level up.*)
- **'f'** (*np.ndarray, float, shape = (K+len(state_list)), 'free energies' of the new states (i.e. ln (<A>-Amin+logfactor)) as the log form is easier to work with.*)
- **'bootstrapped_observables'** (*np.ndarray, float, shape = (n_boostraps,S), array of the observables bootstrapped over random samples.*)
- **'bootstrapped_f'** (*np.ndarray, float, shape = (n_boostraps,S), free energies 'f' bootstrapped over random samples.*)

Notes

Situations this will be used for:

- Multiple observables, single state (called though compute_multiple_expectations)
- Single observable, multiple states (called through compute_expectations)

  This has two cases : observables that don't change with state, and observables that do change with state. For example, the set of energies at state k consist in energy function of state 1 evaluated at state 1, energies of state 2 evaluated at state 2, and so forth.

- Computing only free energies at new states.

Examples

```
>>> from pymbar import testsystems
>>> (x_n, u_kn, N_k, s_n) = testsystems.HarmonicOscillatorsTestCase().
↪sample(mode='u_kn')
>>> mbar = MBAR(u_kn, N_k)
>>> A_n = np.array([x_n,x_n**2,x_n**3])
>>> u_n = u_kn[:2,:]
>>> state_map = np.array([[0,0],[1,0],[2,0],[2,1]],int)
>>> results = mbar.compute_expectations_inner(A_n, u_n, state_map)
```

compute_free_energy_differences(*compute_uncertainty=True*, *uncertainty_method=None*, *warning_cutoff=1e-10*, *return_theta=False*)

Compute and return the dimensionless free energy differences and uncertainties among all thermodynamic states.

Parameters

- **compute_uncertainty** (*bool, optional*) – If False, the uncertainties will not be computed (default : True)
- **uncertainty_method** (*string, optional*) – Choice of method used to compute asymptotic covariance method, or None to use default. See help for _computeAsymptoticCovarianceMatrix() for more information on various methods. (default : svd) The exception is the "bootstrap" option, which requires n_boostraps being defined upon initialization of MBAR.
- **warning_cutoff** (*float, optional*) – Warn if squared-uncertainty is negative and larger in magnitude than this number (default : 1.0e-10)
- **return_theta** (*bool, optional*) – Whether or not to return the theta matrix. Can be useful for complicated differences.

Returns

- *Results dictionary with the following keys*
- **'Delta_f'** (*np.ndarray, float, shape=(K, K)*) – Deltaf_ij[i,j] is the estimated free energy difference
- **'dDelta_f'** (*np.ndarray, float, shape=(K, K)*) – If compute_uncertainty==True, dDeltaf_ij[i,j] is the estimated statistical uncertainty (one standard deviation) in Deltaf_ij[i,j]. Otherwise not included.
- **'Theta'** (*np.ndarray, float, shape=(K, K)*) – The theta_matrix if return_theta==True, otherwise not included.

Notes

Computation of the covariance matrix may take some time for large K.

The reported statistical uncertainty should, in the asymptotic limit, reflect one standard deviation for the normal distribution of the estimate. The true free energy difference should fall within the interval [-df, +df] centered on the estimate 68% of the time, and within the interval [-2 df, +2 df] centered on the estimate 95% of the time. This will break down in cases where the number of samples is not large enough to reach the asymptotic normal limit.

See Section III of Reference [1].

Examples

```
>>> from pymbar import testsystems
>>> (x_n, u_kn, N_k, s_n) = testsystems.HarmonicOscillatorsTestCase().
↪sample(mode='u_kn')
>>> mbar = MBAR(u_kn, N_k)
>>> results = mbar.compute_free_energy_differences()
```

`compute_multiple_expectations`(*A_in*, *u_n*, *compute_uncertainty=True*, *compute_covariance=False*, *uncertainty_method=None*, *warning_cutoff=1e-10*, *return_theta=False*)

Compute the expectations of multiple observables of phase space functions.

Compute the expectations of multiple observables of phase space functions [A_0(x),A_1(x),…,A_i(x)] at a single state, along with the error in the estimates and the uncertainty in the estimates. The state is specified by the choice of u_n, which is the energy of the n samples evaluated at a the chosen state.

Parameters

- **`A_in`** (*`np.ndarray, float, shape=(I, k, N)`*) – A_in[i,n] = A_i(x_n), the value of phase observable i for configuration n at state of interest
- **`u_n`** (*`np.ndarray, float, shape=(N)`*) – u_n[n] is the reduced potential of configuration n at the state of interest
- **`compute_uncertainty`** (*`bool, optional, default=True`*) – If True, calculate the uncertainty
- **`compute_covariance`** (*`bool, optional, default=False`*) – If True, calculate the covariance
- **`uncertainty_method`** (*`string, optional`*) – Choice of method used to compute asymptotic covariance method, or None to use default See help for computeAsymptoticCovarianceMatrix() for more information on various methods. if method = "bootstrap" then uncertainty over bootstrap samples is used. with bootstraps. (default: None)
- **`warning_cutoff`** (*`float, optional`*) – Warn if squared-uncertainty is negative and larger in magnitude than this number (default : 1.0e-10)

Returns

- *Results dictionary with the following keys*
- **'mu'** (*np.ndarray, float, shape=(I)*) – result_vals['mu'] is the estimate for the expectation of A_i(x) at the state specified by u_kn

- **'sigma'** (*np.ndarray, float, shape = (I)*) – result_vals['sigma'] is the uncertainty in the expectation of A_state_map[i](x) at the state specified by u_n[state_map[i],:] or None if compute_uncertainty is False
- **'covariances'** (*np.ndarray, float, shape=(I, I)*) – result_vals['covariances'] is the COVARIANCE in the estimates of A_i[i] and A_i[j]: we can't actually take a square root or None if compute_covariance is False
- **'Theta'** (*np.ndarray, float, shape=(I, I), covariances of the log weights, useful for some additional calculations.*)

Examples

```
>>> from pymbar import testsystems
>>> (x_n, u_kn, N_k, s_n) = testsystems.HarmonicOscillatorsTestCase().
↪sample(mode='u_kn')
>>> mbar = MBAR(u_kn, N_k)
>>> A_in = np.array([x_n,x_n**2,x_n**3])
>>> u_n = u_kn[0,:]
>>> results = mbar.compute_multiple_expectations(A_in, u_kn)
```

`compute_overlap()`

Compute estimate of overlap matrix between the states.

Parameters
`None` –

Returns

- *Results dictionary with the following keys*
- **'scalar'** (*np.ndarray, float, shape=(K, K)*) – One minus the largest nontrival eigenvalue (largest is 1 or -1)
- **'eigenvalues'** (*np.ndarray, float, shape=(K)*) – The sorted (descending) eigenvalues of the overlap matrix.
- **'matrix'** (*np.ndarray, float, shape=(K, K)*) – Estimated state overlap matrix : O[i,j] is an estimate of the probability of observing a sample from state i in state j

Notes

```
W.T * W \approx \int (p_i p_j /\sum_k N_k p_k)^2 \sum_k N_k p_k dq^N
    = \int (p_i p_j /\sum_k N_k p_k) dq^N
```

Multiplying elementwise by N_i, the elements of row i give the probability for a sample from state i being observed in state j.

Examples

```
>>> from pymbar import testsystems
>>> (x_kn, u_kn, N_k, s_n) = testsystems.HarmonicOscillatorsTestCase().
↪sample(mode='u_kn')
>>> mbar = MBAR(u_kn, N_k)
>>> results = mbar.compute_overlap()
```

compute_perturbed_free_energies(*u_ln*, *compute_uncertainty=True*, *uncertainty_method=None*, *warning_cutoff=1e-10*)

Compute the free energies for a new set of states.

Here, we desire the free energy differences among a set of new states, as well as the uncertainty estimates in these differences.

Parameters

- **u_ln** (*np.ndarray, float, shape=(L, Nmax)*) – u_ln[l,n] is the reduced potential energy of uncorrelated configuration n evaluated at new state k. Can be completely indepednent of the original number of states.
- **compute_uncertainty** (*bool, optional, default=True*) – If False, the uncertainties will not be computed (default: True)
- **uncertainty_method** (*string, optional*) – Choice of method used to compute asymptotic covariance method, or None to use default See help for computeAsymptoticCovarianceMatrix() for more information on various methods. if method = "bootstrap" then uncertainty over bootstrap samples is used. with bootstraps. (default: None)
- **warning_cutoff** (*float, optional*) – Warn if squared-uncertainty is negative and larger in magnitude than this number (default: 1.0e-10)

Returns

- *Results dictionary with the following entries.*
- **'Delta_f'** (*np.ndarray, float, shape=(L, L)*) – result_vals['Delta_f'] = f_j - f_i, the dimensionless free energy difference between new states i and j
- **'dDelta_f'** (*np.ndarray, float, shape=(L, L)*) – result_vals['dDelta_f'] is the estimated statistical uncertainty in result_vals['Delta_f'] or not included if *compute_uncertainty* is False

Examples

```
>>> from pymbar import testsystems
>>> (x_n, u_kn, N_k, s_n) = testsystems.HarmonicOscillatorsTestCase().
↪sample(mode='u_kn')
>>> mbar = MBAR(u_kn, N_k)
>>> results = mbar.compute_perturbed_free_energies(u_kn)
```

weights()

Retrieve the weight matrix W_nk from the MBAR algorithm.

Necessary because they are stored internally as log weights.

Returns

weights – NxK matrix of weights in the MBAR covariance and averaging formulas

Return type
np.ndarray, float, shape=(N, K)

1.5 Free energy surfaces with pymbar

1.5.1 Free energy surfces

`pymbar` can be used to estimate free energy surfaces using samples from K biased simulations. It is important to note that MBAR itself is not enough to generate a free energy surface. MBAR takes a set of samples from K different states, and can compute the weight that should be given to each sample in in the unbiased state, i.e. the state in which one desires to compute the free energy surface. Thus, there can be no MBAR estimator of the free energy surface; that would consist only in a set of weighted delta functions. This is done by initializing the `pymbar.FES` class, which takes u_{kn} and N_k matrices and passes them to MBAR.

The second step that needs to be carried out is to determine the best approximation of the continuous function that the samples are estimated from. `pymbar.FES` supports several methods to estimate this continuous function. `generate_fes`, given an initialized MBAR object, a set of points, the energies at that point, and a method, generates an object that contains the FES information. The current options are `histogram`, `kde`, and spline. `histogram` behaves as one might expect, creating a free energy surface as a histogram, and refer to `FES.rst` for additional information. `kde` creates a kernel density approximation, using the `sklearn.neighbors.KernelDensity function`, and parameters can be passed to that function using the `kde_parameters` keyword. Finally, the `spline` method uses a maximum likelihood approach to calculate the spline most consistent with the input data, using the formalism presented in Shirts et al. [1]. The `spline` functionality includes the ability to perform Monte Carlo sampling in the spline parameters to generate confidence intervals for the points in the spline curve.

`histogram` and `kde` methods can generate multidimesional free energy surfaces, while `splines` for now is limited to a single free energy surface.

The method *get_fes* return values of the free energy surface at the specified coordinates, and when available, returns the uncertainties in the values as well.

Examples *parallel-tempering-2d* and *umbrella-sampling* have been rewnamed *parallel-tempering-2dfes and `umbrella-sampling* and rewritten to demonstrate the new functionality.

class `pymbar.FES`(*u_kn*, *N_k*, *verbose=False*, *mbar_options=None*, *timings=True*, ***kwargs*)

Methods for generating free energy surfaces (profile) with statistical uncertainties.

Notes

Note that this method assumes the data are uncorrelated.

Correlated data must be subsampled to extract uncorrelated (effectively independent) samples.

References

[1] Shirts MR and Chodera JD. Statistically optimal analysis of samples from multiple equilibrium states. J. Chem. Phys. 129:124105, 2008 http://dx.doi.org/10.1063/1.2978177

[2] Shirts MR and Ferguson AF. Statistically optimal continuous free energy surfaces from umbrella sampling and multistate reweighting https://arxiv.org/abs/2001.01170

Initialize a free energy surface calculation by performing multistate Bennett acceptance ratio (MBAR) on a set of simulation data from umbrella sampling at K states.

Upon initialization, the dimensionless free energies for all states are computed. This may take anywhere from seconds to minutes, depending upon the quantity of data.

This also creates an internal mbar object that is used to create the free energy surface.

Parameters

- **u_kn** (*np.ndarray, float, shape=(K, N_max)*) – u_kn[k,n] is the reduced potential energy of uncorrelated configuration n evaluated at state k.
- **N_k** (*np.ndarray, int, shape=(K)*) – N_k[k] is the number of uncorrelated snapshots sampled from state k. Some may be zero, indicating that there are no samples from that state.

  We assume that the states are ordered such that the first N_k are from the first state, the 2nd N_k the second state, and so forth. This only becomes important for bar – MBAR does not care which samples are from which state. We should eventually allow this assumption to be overwritten by parameters passed from above, once u_kln is phased out.
- **mbar_options** (*dict*) – The following options supported by mbar (see MBAR documentation)

  maximum_iterations : int, optional relative_tolerance : float, optional verbosity : bool, optional initial_f_k : np.ndarray, float, shape=(K), optional solver_protocol : list(dict) or None, optional, default=None initialize : 'zeros' or 'BAR', optional, Default: 'zeros' x_kindices : array of ints, shape=(K), which state index each sample is from.

Examples

```
>>> from pymbar import testsystems
>>> (x_n, u_kn, N_k, s_n) = testsystems.HarmonicOscillatorsTestCase().sample(mode=
↪'u_kn')
>>> fes = FES(u_kn, N_k)
```

generate_fes(*u_n, x_n, fes_type='histogram', histogram_parameters=None, kde_parameters=None, spline_parameters=None, n_bootstraps=0, seed=-1*)

Given an intialized MBAR object, a set of points, the desired energies at that point, and a method, generate an object that contains the FES information.

Parameters

- **u_n** (*np.ndarray, float, shape=(N)*) – u_n[n] is the reduced potential energy of snapshot n of state for which the FES is to be computed. Often, it will be one of the states in of u_kn, used in initializing the FES object, but we want to allow more generality.
- **x_n** (*np.ndarray, float, shape=(N,D)*) – x_n[n] is the d-dimensional coordinates of the samples, where D is the reduced dimensional space.
- **fes_type** (*str*) – options = 'histogram', 'kde', 'spline'
- **histogram_parameters** (*dictionary*) – Input dictionary with the following keys:

  bin_edges: list of ndim np.ndarray, each array shaped ndum+1
  The bin edges. Compatible with *bin_edges* output of np.histogram.

  kde_parameters
  all the parameters from sklearn (https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KernelDensity.html). Defaults will be used if nothing changed.

  spline_parameters

’spline_weights’
which type of fit to use: ‘biasedstates’ - sum of log likelihood over all weighted states ‘unbiasedstate’ - log likelihood of the single unbiased state ‘simplesum’ - sum of log likelihoods from the biased simulation. Essentially equivalent to vFEP (York et al.)

’optimization_algorithm’:
’Custom-NR’ - a custom Newton-Raphson that is particularly fast for close data, but can fail ‘Newton-CG’ - scipy Newton-CG, only Hessian based method that works correctly because of data ordering. ‘ ‘ - scipy gradient based methods that work, but are generally slower (CG, BFGS, L-LBFGS-B, TNC, SLSQP)

’fkbias’
[array of functions] Return the Kth bias potential for each function

’nspline’
[int] Number of spline points

’kdegree’
[int] Degree of the spline. Default is cubic (‘3’)

’objective’
[string] ’ml’,’map’ # whether to fit the maximum likelihood or the maximum a posteriori

- **n_bootstraps** (*int, 0 or > 1, Default: 0*) – Number of bootstraps to create an uncertainty estimate. If 0, no bootstrapping is done. Required if one uses uncertainty_method = ‘bootstrap’ in get_fes
- **seed** (*int, Default = -1*) – Set the randomization seed. Settting should get the randomization (assuming the same calls are made in the same order) to return the same numbers. This is local to this class and will not change any other random objects.

Returns
if ‘timings’ is set to True in __init__, returns the time taken to generate the FES

Return type
dict, optional

Notes

- **fes_type = ‘histogram’:**
  - This method works by computing the free energy of localizing the system to each bin for the given potential by aggregating the log weights for the given potential.
  - To estimate uncertainties, the NxK weight matrix W_nk is augmented to be Nx(K+nbins) in order to accomodate the normalized weights of states …
  - the potential is given by u_n within each bin and infinite potential outside the bin. The uncertainties with respect to the bin of lowest free energy are then computed in the standard way.

Examples

```
>>> # Generate some test data
>>> from pymbar import testsystems
>>> from pymbar import FES
>>> x_n, u_kn, N_k, s_n = testsystems.HarmonicOscillatorsTestCase().sample(mode=
↪'u_kn',seed=0)
>>> # Select the potential we want to compute the FES for (here, condition 0).
>>> u_n = u_kn[0, :]
>>> # Sort into nbins equally-populated bins
>>> nbins = 10 # number of equally-populated bins to use
>>> import numpy as np
>>> N_tot = N_k.sum()
>>> x_n_sorted = np.sort(x_n) # unroll to n-indices
>>> bins = np.append(x_n_sorted[0::int(N_tot/nbins)], x_n_sorted.max()+0.1)
>>> bin_widths = bins[1:] - bins[0:-1]
>>> # Compute FES for these unequally-sized bins.
>>> fes = FES(u_kn, N_k)
>>> histogram_parameters = dict()
>>> histogram_parameters['bin_edges'] = [bins]
>>> _ = fes.generate_fes(u_n, x_n, fes_type='histogram', histogram_parameters =␣
↪histogram_parameters)
>>> results = fes.get_fes(x_n)
>>> f_i = results['f_i']
>>> for i, x_n in enumerate(x_n):
>>>     print(x_n, f_i[i])
>>> mbar = fes.get_mbar()
>>> print(mbar.f_k)
>>> print(N_k)
```

get_confidence_intervals(*xplot*, *plow*, *phigh*, *reference='zero'*)

Parameters

- **xplot** – data points we want to plot at
- **plow** – lowest percentile
- **phigh** – highest percentile

Returns

plow
[ndarray of float] len(xplot) value of the parameter at plow percentile of the distribution at each x in xplot.

phigh
[ndarray of float] value of the parameter at phigh percentile of the distribution at each x in xplot.

median
[ndarray of float] value of the parameter at the median of the distribution at each x in xplot.

values
[ndarray of float] shape [niterations//sample_every, len(xplot)] of the FES saved during the MCMC sampling at each input value of xplot.

Return type
Dictionary of results. Contains

`get_fes`(*x*, *reference_point='from-lowest'*, *fes_reference=None*, *uncertainty_method=None*)
Returns values of the FES at the specified x points.

Parameters

- **x** (*numpy.ndarray of D dimensions, where D is the dimensionality of the FES defined.*) –
- **reference_point** (*str, optional*) – Method for reporting values and uncertainties (default : 'from-lowest')
  - 'from-lowest' - the uncertainties in the free energy difference with lowest point on FES are reported
  - 'from-specified' - same as from lowest, but from a user specified point
  - 'from-normalization' - the normalization sum_i p_i = 1 is used to determine uncertainties spread out through the FES
  - 'all-differences' - the nbins x nbins matrix df_ij of uncertainties in free energy differences is returned instead of df_i
- **uncertainty_method** (*str, optional*) – Method for computing uncertainties (default: None)
- **fes_reference** – an N-d point specifying the reference state. Ignored except with uncertainty method `from_specified`

Returns

'f_i'
[np.ndarray, float, shape=(K)] result_vals['f_i'][i] is the dimensionless free energy of the x_i point, relative to the reference point

'df_i'
[np.ndarray, float, shape=(K)] result_vals['df_i'][i] is the uncertainty in the difference of x_i with respect to the reference point Only included if uncertainty_method is not None

Return type
dict

`get_information_criteria`(*type='akaike'*)
returns the Akaike or Bayesian Informatiton Criteria for the model if it exists.

Parameters
type (*string*) – either 'Akaike' (or 'akaike' or 'aic') or 'Bayesian' (or 'bayesian' or 'bic')

Returns
value of information criteria

Return type
float

`get_kde`()
return the KernelDensity object if it exists.

Return type
sklearn KernelDensity object

`get_mbar()`

return the MBAR object being used by the FES

Return type
MBAR object

`get_mc_data()`

convenience function to retrieve MC data

Parameters
`None` –

Returns
samples : samples of the parameters with size [len(parameters) times niterations/sample_every] logposteriors : log posteriors (which might be defined with respect to some reference) as a time series size [# points] mc_parameters : dictionary of parameters that were run with (see definitons in sample_parameter_distrbution) acceptance_ratio : overall acceptance ratio of the MC chain nequil : the start of the "equilibrated" data set (i.e. nequil-1 is the number that werer thrown out) g_logposterior : statistical efficiency of the log posterior g_parameters : statistical efficiency of the parametere g : statistical efficiency used for subsampling

Return type
dict

`sample_parameter_distribution`(*x_n*, *mc_parameters=None*, *decorrelate=True*, *verbose=True*)

Samples the valus of the spline parameters with MC.

Parameters

- `x_n` (*numpy.ndarray of D dimensions*) – D is the dimensionality of the FES defined.
- `mc_parameters` (*dictionary*) –

  niteratons
  [int] number of iterations of the Monte Carlo procedure

  fraction_change
  [float] which fraction of the range of input parameters is used to make new MC moves.

  sample_every
  [int] the frequency in steps at which the MC timeseries is saved.

  print_every
  [int] the frequency in steps aat which the MC timeseries is saved to log.

  logprior
  [function,] the function of parameters, the Function must take a single argument, an array the size of parameters in in the same order as a used by the internal functions.

- `decorrelate` (*boolean*) – Whether to decorrelate the time series of output points
- `verbose` (*boolean*) – Whether to print high levels of information to the logger

Return type
None

1.6 Other estimators

`pymbar` implements other reweighting estimators, specifically the Bennett Acceptance Ratio, exponential averaging, and a Gaussian approximation to exponential averaging.

Please reference the following if you use this code in your research:

[1] Shirts MR and Chodera JD. Statistically optimal analysis of samples from multiple equilibrium states. J. Chem. Phys. 129:124105, 2008. http://dx.doi.org/10.1063/1.2978177

This module contains implementations of

- bar - bidirectional estimator for free energy differences / Bennett acceptance ratio estimator
- exp - unidirectional estimator for free energy differences based on Zwanzig relation / exponential averaging
- exp_gauss - unidirectional estimator for free energy differences based on Zwanzig relation / exponential averaging, assuming the distribution is Gaussian.

`pymbar.other_estimators.`**`bar`**(*w_F*, *w_R*, *DeltaF=0.0*, *compute_uncertainty=True*, *uncertainty_method='BAR'*, *maximum_iterations=500*, *relative_tolerance=1e-12*, *verbose=False*, *method='false-position'*, *iterated_solution=True*)

Compute free energy difference using the Bennett acceptance ratio (BAR) method.

Parameters

- **`w_F`** (*`np.ndarray`*) – w_F[t] is the forward work value from snapshot t. t = 0. . . (T_F-1) Length T_F is deduced from vector.
- **`w_R`** (*`np.ndarray`*) – w_R[t] is the reverse work value from snapshot t. t = 0. . . (T_R-1) Length T_R is deduced from vector.
- **`DeltaF`** (*`float, optional, default=0.0`*) – DeltaF can be set to initialize the free energy difference with a guess
- **`compute_uncertainty`** (*`bool, optional, default=True`*) – if False, only the free energy is returned
- **`uncertainty_method`** (*`string, optional, default="BAR"`*) – There are two possible uncertainty estimates for BAR. One agrees with MBAR for two states exactly, and is indicated by "MBAR". The other estimator, which is the one originally derived for BAR, only agrees with MBAR in the limit of good overlap, and is designated 'BAR' See code comments below for derivations of the two methods.
- **`maximum_iterations`** (*`int, optional, default=500`*) – Can be set to limit the maximum number of iterations performed
- **`relative_tolerance`** (*`float, optional, default=1E-11`*) – Can be set to determine the relative tolerance convergence criteria (defailt 1.0e-11)
- **`verbose`** (*`bool`*) – Should be set to True if verbse debug output is desired (default False)
- **`method`** (*`str, optional, defualt='false-position'`*) – Choice of method to solve bar nonlinear equations: one of 'bisection', 'self-consistent-iteration' or 'false-position' (default : 'false-position').
- **`iterated_solution`** (*`bool, optional, default=True`*) – whether to fully solve the optimized bar equation to consistency, or to stop after one step, to be equivalent to transition matrix sampling.

Returns

'Delta_f'
[float] Free energy difference

'dDelta_f'
[float] Estimated standard deviation of free energy difference

Return type
dict

References

[1] Shirts MR, Bair E, Hooker G, and Pande VS. Equilibrium free energies from nonequilibrium measurements using maximum-likelihood methods. PRL 91(14):140601, 2003.

Notes

The false position method is used to solve the implicit equation.

Examples

Compute free energy difference between two specified samples of work values.

```
>>> from pymbar import testsystems
>>> [w_F, w_R] = testsystems.gaussian_work_example(mu_F=None, DeltaF=1.0, seed=0)
>>> results = bar(w_F, w_R)
>>> print('Free energy difference is {:.3f} +- {:.3f} kT'.format(results['Delta_f'],
↪ results['dDelta_f']))
Free energy difference is 1.088 +- 0.050 kT
```

Test completion of various other schemes.

```
>>> results = bar(w_F, w_R, method='self-consistent-iteration')
>>> results = bar(w_F, w_R, method='false-position')
>>> results = bar(w_F, w_R, method='bisection')
```

`pymbar.other_estimators.`**`bar_overlap`**`(`*w_F*, *w_R*`)`

Compute overlap between forward and backward ensembles (using MBAR definition of overlap)

Parameters

- **w_F** (*np.ndarray*) – w_F[t] is the forward work value from snapshot t. t = 0…(T_F-1) Length T_F is deduced from vector.
- **w_R** (*np.ndarray*) – w_R[t] is the reverse work value from snapshot t. t = 0…(T_R-1) Length T_R is deduced from vector.

Returns
overlap – The overlap: 0 denotes no overlap, 1 denotes complete overlap

Return type
float

`pymbar.other_estimators.`**`bar_zero`**`(`*w_F*, *w_R*, *DeltaF*`)`

A function that when zeroed is equivalent to the solution of the Bennett acceptance ratio.

from http://journals.aps.org/prl/pdf/10.1103/PhysRevLett.91.140601
D_F = M + w_F - Delta F D_R = M + w_R - Delta F

we want:

sum_N_F (1+exp(D_F))^-1 = sum N_R N_R <(1+exp(-D_R))^-1> ln sum N_F (1+exp(D_F))^-1>_F = ln sum N_R exp((1+exp(-D_R))^(-1)>_R ln sum N_F (1+exp(D_F))^-1>_F - ln sum N_R exp((1+exp(-D_R))^(-1)>_R = 0

Parameters

- **w_F** (*np.ndarray*) – w_F[t] is the forward work value from snapshot t. t = 0…(T_F-1) Length T_F is deduced from vector.
- **w_R** (*np.ndarray*) – w_R[t] is the reverse work value from snapshot t. t = 0…(T_R-1) Length T_R is deduced from vector.
- **DeltaF** (*float*) – Our current guess

Returns

fzero – a variable that is zeroed when DeltaF satisfies bar.

Return type

float

Examples

Compute free energy difference between two specified samples of work values.

```
>>> from pymbar import testsystems
>>> [w_F, w_R] = testsystems.gaussian_work_example(mu_F=None, DeltaF=1.0, seed=0)
>>> DeltaF = bar_zero(w_F, w_R, 0.0)
```

pymbar.other_estimators.**exp**(*w_F*, *compute_uncertainty=True*, *is_timeseries=False*)

Estimate free energy difference using one-sided (unidirectional) exponential averaging (EXP).

Parameters

- **w_F** (*np.ndarray, float*) – w_F[t] is the forward work value from snapshot t. t = 0…(T-1) Length T is deduced from vector.
- **compute_uncertainty** (*bool, optional, default=True*) – if False, will disable computation of the statistical uncertainty (default: True)
- **is_timeseries** (*bool, default=False*) – if True, correlation in data is corrected for by estimation of statisitcal inefficiency (default: False) Use this option if you are providing correlated timeseries data and have not subsampled the data to produce uncorrelated samples.

Returns

- **'Delta_f'** (*float*) – Free energy difference
- **'dDelta_f'** (*float*) – Estimated standard deviation of free energy difference

Notes

If you are prodividing correlated timeseries data, be sure to set the 'timeseries' flag to True

Examples

Compute the free energy difference given a sample of forward work values.

```
>>> from pymbar import testsystems
>>> [w_F, w_R] = testsystems.gaussian_work_example(mu_F=None, DeltaF=1.0, seed=0)
>>> results = exp(w_F)
>>> print('Forward free energy difference is {:.3f} +- {:.3f} kT'.format(results[
↪'Delta_f'], results['dDelta_f']))
Forward free energy difference is 1.088 +- 0.076 kT
>>> results = exp(w_R)
>>> print('Reverse free energy difference is {:.3f} +- {:.3f} kT'.format(results[
↪'Delta_f'], results['dDelta_f']))
Reverse free energy difference is -1.073 +- 0.082 kT
```

`pymbar.other_estimators.`**`exp_gauss`**`(`*w_F*, *compute_uncertainty=True*, *is_timeseries=False*`)`

Estimate free energy difference using gaussian approximation to one-sided (unidirectional) exponential averaging.

Parameters

- **w_F** (*np.ndarray, float*) – w_F[t] is the forward work value from snapshot t. t = 0…(T-1) Length T is deduced from vector.
- **compute_uncertainty** (*bool, optional, default=True*) – if False, will disable computation of the statistical uncertainty (default: True)
- **is_timeseries** (*bool, default=False*) – if True, correlation in data is corrected for by estimation of statisitcal inefficiency (default: False) Use this option if you are providing correlated timeseries data and have not subsampled the data to produce uncorrelated samples.

Returns

'Delta_f'
[float] Free energy difference between the two states

'dDelta_f'
[float] Estimated standard deviation of free energy difference between the two states

Return type

Results dictionary with keys

Notes

If you are providing correlated timeseries data, be sure to set the 'timeseries' flag to `True`

Examples

Compute the free energy difference given a sample of forward work values.

```
>>> from pymbar import testsystems
>>> [w_F, w_R] = testsystems.gaussian_work_example(mu_F=None, DeltaF=1.0, seed=0)
>>> results = exp_gauss(w_F)
>>> print('Forward Gaussian approximated free energy difference is {:.3f} +- {:.3f}␣
↪kT'.format(results['Delta_f'], results['dDelta_f']))
Forward Gaussian approximated free energy difference is 1.049 +- 0.089 kT
>>> results = exp_gauss(w_R)
>>> print('Reverse Gaussian approximated free energy difference is {:.3f} +- {:.3f}␣
↪kT'.format(results['Delta_f'], results['dDelta_f']))
Reverse Gaussian approximated free energy difference is -1.073 +- 0.080 kT
```

1.7 The timeseries module pymbar.timeseries

The *pymbar.timeseries* module contains tools for dealing with timeseries data. The MBAR method is only applicable to uncorrelated samples from probability distributions, so we provide a number of tools that can be used to decorrelate simulation data.

1.7.1 Automatically identifying the equilibrated production region

Most simulations start from initial conditions that are highly unrepresentative of equilibrated samples that occur late in the simulation. We can improve our estimates by discarding these initial regions to "equilibration" (also known as "burn-in"). We recommend a simple scheme described in Ref. [3], which identifies the production region as the final contiguous region containing the *largest* number of uncorrelated samples. This scheme is implemented in the *detect_equilibration()* method:

```
from pymbar import timeseries
t0, g, Neff_max = timeseries.detect_equilibration(A_t) # compute indices of uncorrelated␣
↪timeseries
A_t_equil = A_t[t0:]
indices = timeseries.subsample_correlated_data(A_t_equil, g=g)
A_n = A_t_equil[indices]
```

In this example, the *detect_equilibration()* method is used on the correlated timeseries `A_t` to identify the sample index corresponding to the beginning of the production region, `t_0`, the statistical inefficiency of the production region `[t0:]`, `g`, and the effective number of uncorrelated samples in the production region, `Neff_max`. The production (equilibrated) region of the timeseries is extracted as `A_t_equil` and then subsampled using the *subsample_correlated_data()* method with the provided statistical inefficiency `g`. Finally, the decorrelated samples are stored in `A_n`.

Note that, by default, the statistical inefficiency is computed for every time origin in a call to *detect_equilibration()*, which can be slow. If your dataset is more than a few hundred samples, you may want to evaluate only every `nskip` samples as potential time origins. This may result in discarding slightly more data than strictly necessary, but may not have a significant impact if the timeseries is long.

```
nskip = 10 # only try every 10 samples for time origins
t0, g, Neff_max = timeseries.detect_equilibration(A_t, nskip=nskip)
```

1.7.2 Subsampling timeseries data

If there is no need to discard the initial transient to equilibration, the *subsample_correlated_data()* method can be used directly to identify an effectively uncorrelated subset of data.

```
from pymbar import timeseries
indices = timeseries.subsample_correlated_data(A_t_equil)
A_n = A_t_equil[indices]
```

Here, the statistical inefficiency `g` is computed automatically.

1.7.3 Other utility timeseries functions

A number of other useful functions for computing autocorrelation functions from one or more timeseries sampled from the same process are also provided.

A module for extracting uncorrelated samples from correlated timeseries data.

This module provides various tools that allow one to examine the correlation functions and integrated autocorrelation times in correlated timeseries data, compute statistical inefficiencies, and automatically extract uncorrelated samples for data analysis.

Please reference the following if you use this code in your research:

[1] Shirts MR and Chodera JD. Statistically optimal analysis of samples from multiple equilibrium states. J. Chem. Phys. 129:124105, 2008 http://dx.doi.org/10.1063/1.2978177

[2] J. D. Chodera, W. C. Swope, J. W. Pitera, C. Seok, and K. A. Dill. Use of the weighted histogram analysis method for the analysis of simulated and parallel tempering simulations. JCTC 3(1):26-41, 2007.

`pymbar.timeseries.`**`detect_equilibration`**(*A_t*, *fast=True*, *nskip=1*)

Automatically detect equilibrated region of a dataset using a heuristic that maximizes number of effectively uncorrelated samples.

Parameters

- **A_t** (*np.ndarray*) – timeseries
- **nskip** (*int, optional, default=1*) – number of samples to sparsify data by in order to speed equilibration detection

Returns

- **t** (*int*) – start of equilibrated data
- **g** (*float*) – statistical inefficiency of equilibrated data
- **Neff_max** (*float*) – number of uncorrelated samples

Notes

If your input consists of some period of equilibration followed by a constant sequence, this function treats the trailing constant sequence as having Neff = 1.

Examples

Determine start of equilibrated data for a correlated timeseries.

```
>>> from pymbar import testsystems
>>> A_t = testsystems.correlated_timeseries_example(N=1000, tau=5.0) # generate a␣
↪test correlated timeseries
>>> [t, g, Neff_max] = detect_equilibration(A_t) # compute indices of uncorrelated␣
↪timeseries
```

Determine start of equilibrated data for a correlated timeseries with a shift.

```
>>> from pymbar import testsystems
>>> A_t = testsystems.correlated_timeseries_example(N=1000, tau=5.0) + 2.0 #␣
↪generate a test correlated timeseries
>>> B_t = testsystems.correlated_timeseries_example(N=10000, tau=5.0) # generate a␣
↪test correlated timeseries
>>> C_t = np.concatenate([A_t, B_t])
>>> [t, g, Neff_max] = detect_equilibration(C_t, nskip=50) # compute indices of␣
↪uncorrelated timeseries
```

`pymbar.timeseries.`**`detect_equilibration_binary_search`**(*A_t*, *bs_nodes=10*)

Automatically detect equilibrated region of a dataset using a heuristic that maximizes number of effectively uncorrelated samples.

Parameters

- **A_t** (*np.ndarray*) – timeseries
- **bs_nodes** (*int > 4*) – number of geometrically distributed binary search nodes

Returns

- **t** (*int*) – start of equilibrated data
- **g** (*float*) – statistical inefficiency of equilibrated data
- **Neff_max** (*float*) – number of uncorrelated samples

Notes

Finds the discard region (t) by a binary search on the range of possible lagtime values, with logarithmic spacings. This will give a local maximum. The global maximum is not guaranteed, but will likely be found if the N_eff[t] varies smoothly.

`pymbar.timeseries.`**`integrated_autocorrelation_time`**(*A_n*, *B_n=None*, *fast=False*, *mintime=3*)

Estimate the integrated autocorrelation time.

See also:

`statisticalInefficiency`

`pymbar.timeseries.`**`integrated_autocorrelation_timeMultiple`**(*A_kn*, *fast=False*)

Estimate the integrated autocorrelation time from multiple timeseries.

See also:

`statistical_inefficiency_multiple`

`pymbar.timeseries.`**`normalized_fluctuation_correlation_function`**(*A_n*, *B_n=None*, *N_max=None*, *norm=True*)

Compute the normalized fluctuation (cross) correlation function of (two) stationary timeseries.

C(t) = (<A(t) B(t)> - <A><B>) / (<AB> - <A><B>)

This may be useful in diagnosing odd time-correlations in timeseries data.

Parameters

- **A_n** (*np.ndarray*) – A_n[n] is nth value of timeseries A. Length is deduced from vector.
- **B_n** (*np.ndarray*) – B_n[n] is nth value of timeseries B. Length is deduced from vector.
- **N_max** (*int, default=None*) – if specified, will only compute correlation function out to time lag of N_max
- **norm** (*bool, optional, default=True*) – if False will return the unnormalized correlation function D(t) = <A(t) B(t)>

Returns

C_n – C_n[n] is the normalized fluctuation auto- or cross-correlation function for timeseries A(t) and B(t).

Return type

np.ndarray

Notes

The same timeseries can be used for both A_n and B_n to get the autocorrelation statistical inefficiency. This procedure may be slow. The statistical error in C_n[n] will grow with increasing n. No effort is made here to estimate the uncertainty.

References

[1] J. D. Chodera, W. C. Swope, J. W. Pitera, C. Seok, and K. A. Dill. Use of the weighted histogram analysis method for the analysis of simulated and parallel tempering simulations. JCTC 3(1):26-41, 2007.

Examples

Estimate normalized fluctuation correlation function.

```
>>> from pymbar import testsystems
>>> A_t = testsystems.correlated_timeseries_example(N=10000, tau=5.0)
>>> C_t = normalized_fluctuation_correlation_function(A_t, N_max=25)
```

`pymbar.timeseries.`**`normalized_fluctuation_correlation_function_multiple`**(*A_kn*, *B_kn=None*, *N_max=None*, *norm=True*, *truncate=False*)

Compute the normalized fluctuation (cross) correlation function of (two) timeseries from multiple timeseries samples.

C(t) = (<A(t) B(t)> - <A><B>) / (<AB> - <A><B>) This may be useful in diagnosing odd time-correlations in timeseries data.

Parameters

- **A_kn** (*Python list of numpy arrays*) – A_kn[k] is the kth timeseries, and A_kn[k][n] is nth value of timeseries k. Length is deduced from arrays.
- **B_kn** (*Python list of numpy arrays*) – B_kn[k] is the kth timeseries, and B_kn[k][n] is nth value of timeseries k. B_kn[k] must have same length as A_kn[k]
- **N_max** (*int, optional, default=None*) – if specified, will only compute correlation function out to time lag of N_max
- **norm** (*bool, optional, default=True*) – if False, will return unnormalized D(t) = <A(t) B(t)>
- **truncate** (*bool, optional, default=False*) – if True, will stop calculating the correlation function when it goes below 0

Returns
C_n[n] – The normalized fluctuation auto- or cross-correlation function for timeseries A(t) and B(t).

Return type
np.ndarray

Notes

The same timeseries can be used for both A_n and B_n to get the autocorrelation statistical inefficiency. This procedure may be slow. The statistical error in C_n[n] will grow with increasing n. No effort is made here to estimate the uncertainty.

References

[1] J. D. Chodera, W. C. Swope, J. W. Pitera, C. Seok, and K. A. Dill. Use of the weighted histogram analysis method for the analysis of simulated and parallel tempering simulations. JCTC 3(1):26-41, 2007.

Examples

Estimate a portion of the normalized fluctuation autocorrelation function from multiple timeseries of different length.

```
>>> from pymbar import testsystems
>>> N_k = [1000, 2000, 3000, 4000, 5000]
>>> tau = 5.0 # exponential relaxation time
>>> A_kn = [ testsystems.correlated_timeseries_example(N=N, tau=tau) for N in N_k ]
>>> C_n = normalized_fluctuation_correlation_function_multiple(A_kn, N_max=25)
```

pymbar.timeseries.**statistical_inefficiency**(*A_n*, *B_n=None*, *fast=False*, *mintime=3*, *fft=False*)

Compute the (cross) statistical inefficiency of (two) timeseries.

Parameters

- **A_n** (*np.ndarray, float*) – A_n[n] is nth value of timeseries A. Length is deduced from vector.
- **B_n** (*np.ndarray, float, optional, default=None*) – B_n[n] is nth value of timeseries B. Length is deduced from vector. If supplied, the cross-correlation of timeseries A and B will be estimated instead of the autocorrelation of timeseries A.

- **fast** (*bool, optional, default=False*) – f True, will use faster (but less accurate) method to estimate correlation time, described in Ref. [1] (default: False). This is ignored when B_n=None and fft=True.
- **mintime** (*int, optional, default=3*) – minimum amount of correlation function to compute (default: 3) The algorithm terminates after computing the correlation time out to mintime when the correlation function first goes negative. Note that this time may need to be increased if there is a strong initial negative peak in the correlation function.
- **fft** (*bool, optional, default=False*) – If fft=True and B_n=None, then use the fft based approach, as implemented in statistical_inefficiency_fft().

Returns
g – g is the estimated statistical inefficiency (equal to 1 + 2 tau, where tau is the correlation time). We enforce g >= 1.0.

Return type
np.ndarray,

Notes

The same timeseries can be used for both A_n and B_n to get the autocorrelation statistical inefficiency. The fast method described in Ref [1] is used to compute g.

References

[1] J. D. Chodera, W. C. Swope, J. W. Pitera, C. Seok, and K. A. Dill. Use of the weighted histogram analysis method for the analysis of simulated and parallel tempering simulations. JCTC 3(1):26-41, 2007.

Examples

Compute statistical inefficiency of timeseries data with known correlation time.

```
>>> from pymbar.testsystems import correlated_timeseries_example
>>> A_n = correlated_timeseries_example(N=100000, tau=5.0)
>>> g = statistical_inefficiency(A_n, fast=True)
```

pymbar.timeseries.**statistical_inefficiency_fft**(*A_n*, *mintime=3*)

Compute the (cross) statistical inefficiency of (two) timeseries.

Parameters

- **A_n** (*np.ndarray, float*) – A_n[n] is nth value of timeseries A. Length is deduced from vector.
- **mintime** (*int, optional, default=3*) – minimum amount of correlation function to compute (default: 3) The algorithm terminates after computing the correlation time out to mintime when the correlation function first goes negative. Note that this time may need to be increased if there is a strong initial negative peak in the correlation function.

Returns
g – g is the estimated statistical inefficiency (equal to 1 + 2 tau, where tau is the correlation time). We enforce g >= 1.0.

Return type
np.ndarray,

Notes

The same timeseries can be used for both A_n and B_n to get the autocorrelation statistical inefficiency. The fast method described in Ref [1] is used to compute g.

References

[1] J. D. Chodera, W. C. Swope, J. W. Pitera, C. Seok, and K. A. Dill. Use of the weighted histogram analysis method for the analysis of simulated and parallel tempering simulations. JCTC 3(1):26-41, 2007.

`pymbar.timeseries.`**`statistical_inefficiency_multiple`**(*A_kn*, *fast=False*, *return_correlation_function=False*)

Estimate the statistical inefficiency from multiple stationary timeseries (of potentially differing lengths).

Parameters

- **`A_kn`** (*`list of np.ndarrays`*) – A_kn[k] is the kth timeseries, and A_kn[k][n] is nth value of timeseries k. Length is deduced from arrays.
- **`fast`** (*`bool, optional, default=False`*) – f True, will use faster (but less accurate) method to estimate correlation time, described in Ref. [1] (default: False)
- **`return_correlation_function`** (*`bool, optional, default=False`*) – if True, will also return estimates of normalized fluctuation correlation function that were computed (default: False)

Returns

- **g** (*np.ndarray,*) – g is the estimated statistical inefficiency (equal to 1 + 2 tau, where tau is the correlation time). We enforce g >= 1.0.
- **Ct** (*list (of tuples)*) – Ct[n] = (t, C) with time t and normalized correlation function estimate C is returned as well if return_correlation_function is set to True

Notes

The autocorrelation of the timeseries is used to compute the statistical inefficiency. The normalized fluctuation autocorrelation function is computed by averaging the unnormalized raw correlation functions. The fast method described in Ref [1] is used to compute g.

References

[1] J. D. Chodera, W. C. Swope, J. W. Pitera, C. Seok, and K. A. Dill. Use of the weighted histogram analysis method for the analysis of simulated and parallel tempering simulations. JCTC 3(1):26-41, 2007.

Examples

Estimate statistical efficiency from multiple timeseries of different lengths.

```
>>> from pymbar import testsystems
>>> N_k = [1000, 2000, 3000, 4000, 5000]
>>> tau = 5.0 # exponential relaxation time
>>> A_kn = [ testsystems.correlated_timeseries_example(N=N, tau=tau) for N in N_k ]
>>> g = statistical_inefficiency_multiple(A_kn)
```

Also return the values of the normalized fluctuation autocorrelation function that were computed.

```
>>> [g, Ct] = statistical_inefficiency_multiple(A_kn, return_correlation_
↪function=True)
```

pymbar.timeseries.**subsample_correlated_data**(*A_t*, *g=None*, *fast=False*, *conservative=False*, *verbose=False*)

Determine the indices of an uncorrelated subsample of the data.

Parameters

- **A_t** (*np.ndarray*) – A_t[t] is the t-th value of timeseries A(t). Length is deduced from vector.
- **g** (*float, optional*) – if provided, the statistical inefficiency g is used to subsample the timeseries – otherwise it will be computed (default: None)
- **fast** (*bool, optional, default=False*) – fast can be set to True to give a less accurate but very quick estimate (default: False)
- **conservative** (*bool, optional, default=False*) – if set to True, uniformly-spaced indices are chosen with interval ceil(g), where g is the statistical inefficiency. Otherwise, indices are chosen non-uniformly with interval of approximately g in order to end up with approximately T/g total indices
- **verbose** (*bool, optional, default=False*) – if True, some output is printed

Returns

indices – the indices of an uncorrelated subsample of the data

Return type

list of int

Notes

The statistical inefficiency is computed with the function computeStatisticalInefficiency().

Examples

Subsample a correlated timeseries to extract an effectively uncorrelated dataset.

```
>>> from pymbar import testsystems
>>> A_t = testsystems.correlated_timeseries_example(N=10000, tau=5.0) # generate a␣
↪test correlated timeseries
>>> indices = subsample_correlated_data(A_t) # compute indices of uncorrelated␣
↪timeseries
>>> A_n = A_t[indices] # extract uncorrelated samples
```

Extract uncorrelated samples from multiple timeseries data from the same process.

```
>>> # Generate multiple correlated timeseries data of different lengths.
>>> T_k = [1000, 2000, 3000, 4000, 5000]
>>> K = len(T_k) # number of timeseries
>>> tau = 5.0 # exponential relaxation time
>>> A_kt = [ testsystems.correlated_timeseries_example(N=T, tau=tau) for T in T_k ]
↪# A_kt[k] is correlated timeseries k
>>> # Estimate statistical inefficiency from all timeseries data.
>>> g = statistical_inefficiency_multiple(A_kt)
>>> # Count number of uncorrelated samples in each timeseries.
>>> N_k = np.array([ len(subsample_correlated_data(A_t, g=g)) for A_t in A_kt ]) #␣
↪N_k[k] is the number of uncorrelated samples in timeseries k
>>> N = N_k.sum() # total number of uncorrelated samples
>>> # Subsample all trajectories to produce uncorrelated samples
>>> A_kn = [ A_t[subsample_correlated_data(A_t, g=g)] for A_t in A_kt ] # A_kn[k]␣
↪is uncorrelated subset of trajectory A_kt[t]
>>> # Concatenate data into one timeseries.
>>> A_n = np.zeros([N], np.float32) # A_n[n] is nth sample in concatenated set of␣
↪uncorrelated samples
>>> A_n[0:N_k[0]] = A_kn[0]
>>> for k in range(1,K): A_n[N_k[0:k].sum():N_k[0:k+1].sum()] = A_kn[k]
```

1.8 Utilities : `pymbar.utils`

These functions are some miscellaneous functions used by other parts of the `pymbar` library.

exception `pymbar.utils.`**`BoundsError`**

Could not determine bounds on free energy

exception `pymbar.utils.`**`ConvergenceError`**

Convergence could not be achieved.

exception `pymbar.utils.`**`DataError`**

Data is inconsistent.

exception `pymbar.utils.`**`ParameterError`**

An error in the input parameters has been detected.

exception `pymbar.utils.`**`TypeCastPerformanceWarning`**

`pymbar.utils.`**`check_w_normalized`**(*W*, *N_k*, *tolerance=0.0001*)

Check the weight matrix W is properly normalized. The sum over N should be 1, and the sum over k by N_k should aslo be 1

Parameters

- **W** (*np.ndarray, shape=(N, K), dtype='float'*) – The normalized weight matrix for snapshots and states. W[n, k] is the weight of snapshot n in state k.
- **N_k** (*np.ndarray, shape=(K), dtype='int'*) – N_k[k] is the number of samples from state k.
- **tolerance** (*float, optional, default=1.0e-4*) – Tolerance for checking equality of sums

Returns

None – Returns a None object if test passes

Return type

NoneType

Raises

`ParameterError` – Appropriate message if W is not normalized within tolerance.

`pymbar.utils.`**`ensure_type`**`(`*val*, *dtype*, *ndim*, *name*, *length=None*, *can_be_none=False*, *shape=None*, *warn_on_cast=True*, *add_newaxis_on_deficient_ndim=False*`)`

Typecheck the size, shape and dtype of a numpy array, with optional casting.

Parameters

- **`val`** (*`{np.ndaraay, None}`*) – The array to check
- **`dtype`** (*`{nd.dtype, str}`*) – The dtype you'd like the array to have
- **`ndim`** (*`int`*) – The number of dimensions you'd like the array to have
- **`name`** (*`str`*) – name of the array. This is used when throwing exceptions, so that we can describe to the user which array is messed up.
- **`length`** (*`int, optional`*) – How long should the array be?
- **`can_be_none`** (*`bool`*) – Is `val == None` acceptable?
- **`shape`** (*`tuple, optional`*) – What should be shape of the array be? If the provided tuple has Nones in it, those will be semantically interpreted as matching any length in that dimension. So, for example, using the shape spec `(None, None, 3)` will ensure that the last dimension is of length three without constraining the first two dimensions
- **`warn_on_cast`** (*`bool, default=True`*) – Raise a warning when the dtypes don't match and a cast is done.
- **`add_newaxis_on_deficient_ndim`** (*`bool, default=True`*) – Add a new axis to the beginining of the array if the number of dimensions is deficient by one compared to your specification. For instance, if you're trying to get out an array of `ndim == 3`, but the user provides an array of `shape == (10, 10)`, a new axis will be created with length 1 in front, so that the return value is of shape `(1, 10, 10)`.

Notes

The returned value will always be C-contiguous.

Returns

typechecked_val – If *val=None* and *can_be_none=True*, then this will return None. Otherwise, it will return val (or a copy of val). If the dtype wasn't right, it'll be casted to the right shape. If the array was not C-contiguous, it'll be copied as well.

Return type

np.ndarray, None

`pymbar.utils.`**`kln_to_kn`**`(`*kln*, *N_k=None*, *cleanup=False*`)`

Convert KxKxN_max array to KxN max array

Parameters

- **`u_kln`** (*`np.ndarray, float, shape=(KxLxN_max)`*) –
- **`N_k`** (*`np.array, optional`*) – the N_k matrix from the previous formatting form

- **cleanup** (*bool, optional*) – optional command to clean up, since u_kln can get very large

Returns
u_kn

Return type
np.ndarray, float, shape=(LxN)

`pymbar.utils.`**`kn_to_n`**(*kn*, *N_k=None*, *cleanup=False*)

Convert KxN_max array to N array

Parameters

- **u_kn** (*np.ndarray, float, shape=(KxN_max)*) –
- **N_k** (*np.array, optional*) – the N_k matrix from the previous formatting form
- **cleanup** (*bool, optional*) – optional command to clean up, since u_kln can get very large

Returns
u_n

Return type
np.ndarray, float, shape=(N)

`pymbar.utils.`**`logsumexp`**(*a*, *axis=None*, *b=None*, *use_numexpr=True*)

Compute the log of the sum of exponentials of input elements.

Parameters

- **a** (*array_like*) – Input array.
- **axis** (*None or int, optional, default=None*) – Axis or axes over which the sum is taken. By default *axis* is None, and all elements are summed.
- **b** (*array-like, optional*) – Scaling factor for exp(*a*) must be of the same shape as *a* or broadcastable to *a*.
- **use_numexpr** (*bool, optional, default=True*) – If True, use the numexpr library to speed up the calculation, which can give a 2-4X speedup when working with large arrays.

Returns
res – The result, `log(sum(exp(a)))` calculated in a numerically more stable way. If *b* is given then `log(sum(b*exp(a)))` is returned.

Return type
ndarray

See also:

`numpy.logaddexp`, `numpy.logaddexp2`, `scipy.special.logsumexp`

Notes

This is based on `scipy.special.logsumexp` but with optional numexpr support for improved performance.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] Michael R. Shirts and Andrew L. Ferguson. Statistically optimal continuous free energy surfaces from biased simulations and multistate reweighting. *Journal of Chemical Theory and Computation*, 16(7):4107–4125, 2020.

[2] Michael R. Shirts and John D. Chodera. Statistically optimal analysis of samples from multiple equilibrium states. *Journal of Chemical Physics*, 129:124105, 2008.

[3] John D. Chodera. A simple method for automated equilibration detection in molecular simulations. *Journal of Chemical Theory and Computation*, 12:1799, 2016.

PYTHON MODULE INDEX

INDEX