pymbar Documentation

Release 3.0.3

John D. Chodera, Michael Shirts, Kyle A. Beauchamp, Levi N. Nad

Feb 28, 2018

Contents

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

CHAPTER 1

Documentation

1.1 Getting started

1.1.1 Installing `pymbar`

conda (recommended)

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

:: $ conda install -c omnia pymbar

pip

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

1.1.2 Running the tests

Running the tests is a great way to verify that everything is working. The test suite uses nose, in addition to statsmodels and pytables, which you can install via `conda`:

:: $ conda install nose statsmodels pytables

You can then run the tests with:

:: $ nosetests -vv pymbar

1.2 The `mbar` module: `MBAR`

The `mbar` module contains the *MBAR* class, which implements the multistate Bennett acceptance ratio (MBAR) method [shirts-chodera:jcp:2008:mbar]. A module implementing the multistate Bennett acceptance ratio (MBAR) method for the analysis of equilibrium samples from multiple arbitrary thermodynamic states in computing equilibrium expectations, free energy differences, potentials of mean force, and entropy and enthalpy contributions.

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

- MBAR - multistate Bennett acceptance ratio estimator

class pymbar.mbar.**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*, ***kwargs*)

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 *getFreeEnergyDifferences()*, or expectation at any state of interest can be computed by calls to *computeExpectations()*.

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) 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.

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. Its 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 'zeroes'.

(default: 'zeros', unless specific values are passed in.)

x_kindices

Which state is each `x` from? Usually doesn't matter, but does for BAR. 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.

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/(kB T_k)` is the inverse temperature of condition `k`, where kB 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 time-series 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.statisticalInefficiency()*. 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)
```

W_nk

Retrieve the weight matrix W_nk from the MBAR algorithm.

Necessary because they are stored internally as log weights.

Returns **weights** : np.ndarray, float, shape=(N, K)

NxK matrix of weights in the MBAR covariance and averaging formulas

computeCovarianceOfSums(*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`

computeEffectiveSampleNumber(*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.

Parameters **verbose** : print out information about the effective number of samples

Returns **N_eff** : np.ndarray, float, shape=(K)

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.

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.computeEffectiveSampleNumber()
```

computeEntropyAndEnthalpy(*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 : 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: None)

warning_cutoff : float, optional

Warn if squared-uncertainty is negative and larger in magnitude than this number (default: 1.0e-10)

Returns **result_vals** : dictionary

Keys in the result_vals dictionary:

'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.computeEntropyAndEnthalpy()
```

computeExpectations (*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. (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 result_vals : dictionary

Possible keys in the result_vals dictionary:

'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.computeExpectations(A_n)
>>> A_n = u_kn[0,:]
>>> results = mbar.computeExpectations(A_n, output='differences')
```

computeExpectationsInner(*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

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. (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

Possible 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+1)) as the log form is easier to work with.

Notes

Situations this will be used for :

- Multiple observables, single state (called though computeMultipleExpectations)
- Single observable, multiple states (called through computeExpectations)

  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.
- Would require additional work to work with potentials of mean force, because we need to ignore the functions that are zero when integrating.

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.computeExpectationsInner(A_n, u_n, state_map)
```

computeMultipleExpectations(*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.

Returns result_vals : dictionary

Possible keys in the result_vals dictionary:

'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.computeMultipleExpectations(A_in, u_kn)
```

computeOverlap()

Compute estimate of overlap matrix between the states.

Returns result_vals : dictonary

Possible keys in the result_vals dictionary:

'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.

'O' : 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 pprox \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.computeOverlap()
```

computePMF (*u_n*, *bin_n*, *nbins*, *uncertainties='from-lowest'*, *pmf_reference=None*)

Compute the free energy of occupying a number of bins.

This implementation computes the expectation of an indicator-function observable for each bin.

Parameters u_n : np.ndarray, float, shape=(N)

u_n[n] is the reduced potential energy of snapshot n of state k for which the PMF is to be computed.

bin_n : np.ndarray, float, shape=(N)

bin_n[n] is the bin index of snapshot n of state k. bin_n can assume a value in range(0,nbins)

nbins : int

The number of bins

uncertainties : string, optional

Method for reporting uncertainties (default: 'from-lowest')

- 'from-lowest' - the uncertainties in the free energy difference with lowest point on PMF 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 PMF
- 'all-differences' - the nbins x nbins matrix df_ij of uncertainties in free energy differences is returned instead of df_i

pmf_reference : int, optional

the reference state that is zeroed when uncertainty = 'from-specified'

Returns result_vals : dictionary

Possible keys in the result_vals dictionary:

'f_i' : np.ndarray, float, shape=(K)

result_vals['f_i'][i] is the dimensionless free energy of state i, relative to the state of lowest free energy

'df_i' : np.ndarray, float, shape=(K)

result_vals['df_i'][i] is the uncertainty in the difference of f_i with respect to the state of lowest free energy

Notes

- All bins must have some samples in them from at least one of the states – this will not work if bin_n.sum(0) == 0. Empty bins should be removed before calling computePMF().
- 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 where
- the potential is given by u_kn 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
>>> (x_n, u_kn, N_k, s_n) = testsystems.HarmonicOscillatorsTestCase().
↪sample(mode='u_kn')
>>> # Initialize MBAR on data.
>>> mbar = MBAR(u_kn, N_k)
>>> # Select the potential we want to compute the PMF 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]
>>> bin_n = np.zeros(x_n.shape, np.int64)
>>> bin_n = np.digitize(x_n, bins) - 1
>>> # Compute PMF for these unequally-sized bins.
>>> results = mbar.computePMF(u_n, bin_n, nbins)
>>> # If we want to correct for unequally-spaced bins to get a PMF on uniform␣
↪measure
>>> f_i_corrected = results['f_i'] - np.log(bin_widths)
```

computePerturbedFreeEnergies (*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. (default: None)

warning_cutoff : float, optional

Warn if squared-uncertainty is negative and larger in magnitude than this number (default: 1.0e-10)

Returns result_vals : dictionary

Possible keys in the result_vals dictionary:

'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.computePerturbedFreeEnergies(u_kn)
```

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

Get 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)

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 result_vals : dictionary

Possible keys in the result_vals dictionary:

'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.getFreeEnergyDifferences()
```

getWeights()

Retrieve the weight matrix W_nk from the MBAR algorithm.

Necessary because they are stored internally as log weights.

Returns weights : np.ndarray, float, shape=(N, K)

NxK matrix of weights in the MBAR covariance and averaging formulas

1.3 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.

1.3.1 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. [chodera:jctc:2016:automatic-equilibration-detection], which identifies the production region as the final contiguous region containing the *largest* number of uncorrelated samples. This scheme is implemented in the *detectEquilibration()* method:

```
from pymbar import timeseries
[t0, g, Neff_max] = timeseries.detectEquilibration(A_t) # compute indices of␣
↪uncorrelated timeseries
```

```
A_t_equil = A_t[t0:]
indices = timeseries.subsampleCorrelatedData(A_t_equil, g=g)
A_n = A_t_equil[indices]
```

In this example, the *detectEquilibration()* 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 *subsampleCorrelatedData()* 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 *detectEquilibration()*, 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.detectEquilibration(A_t, nskip=nskip)
```

1.3.2 Subsampling timeseries data

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

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

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

1.3.3 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.`**`detectEquilibration`**(*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] = detectEquilibration(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] = detectEquilibration(C_t, nskip=50) # compute indices of␣
↪uncorrelated timeseries
```

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

Estimate the integrated autocorrelation time.

See also:

statisticalInefficiency

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

Estimate the integrated autocorrelation time from multiple timeseries.

See also:

statisticalInefficiencyMultiple

`pymbar.timeseries.`**`normalizedFluctuationCorrelationFunction`**(*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** : np.ndarray

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

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 = normalizedFluctuationCorrelationFunction(A_t, N_max=25)
```

pymbar.timeseries.**normalizedFluctuationCorrelationFunctionMultiple**(*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]** : np.ndarray

The normalized fluctuation auto- or cross-correlation function for timeseries A(t) and B(t).

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 = normalizedFluctuationCorrelationFunctionMultiple(A_kn, N_max=25)
```

pymbar.timeseries.**statisticalInefficiency**(*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 statisticalInefficiency_fft().

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.

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 = statisticalInefficiency(A_n, fast=True)
```

pymbar.timeseries.**statisticalInefficiencyMultiple**(*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 = statisticalInefficiencyMultiple(A_kn)
```

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

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

pymbar.timeseries.**statisticalInefficiency_fft**(*A_n*, *mintime=3*, *memsafe=None*)

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.

memsafe: bool, optional, default=None (in depreciation)

If this function is used several times on arrays of comparable size then one might benefit from setting this option to False. If set to True then clear np.fft cache to avoid a fast increase in memory consumption when this function is called on many arrays of different sizes.

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.

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.**subsampleCorrelatedData**(*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** : list of int

the indices of an uncorrelated subsample of the data

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 = subsampleCorrelatedData(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 = statisticalInefficiencyMultiple(A_kt)
>>> # Count number of uncorrelated samples in each timeseries.
>>> N_k = np.array([ len(subsampleCorrelatedData(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[subsampleCorrelatedData(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]
```

1.4 The testsystems Module: `pymbar.testsystems`

The pymbar.testsystems module contains a number of test systems with analytically or numerically computable expectations or free energies we use to validate its implementation. These test systems are also convenient to use if you want to easily generate synthetic data to experiment with the capabilities of ``pymbar`.

class pymbar.testsystems.harmonic_oscillators.**HarmonicOscillatorsTestCase**(*O_k=(0, 1, 2, 3, 4), K_k=(1, 2, 4, 8, 16), beta=1.0*)

Test cases using harmonic oscillators.

Examples

Generate energy samples with default parameters.

```
>>> testcase = HarmonicOscillatorsTestCase()
>>> [x_kn, u_kln, N_k, s_n] = testcase.sample()
```

Retrieve analytical properties.

```
>>> analytical_means = testcase.analytical_means()
>>> analytical_variances = testcase.analytical_variances()
>>> analytical_standard_deviations = testcase.analytical_standard_deviations()
>>> analytical_free_energies = testcase.analytical_free_energies()
>>> analytical_x_squared = testcase.analytical_observable('position^2')
```

Generate energy samples with default parameters in one line.

```
>>> (x_kn, u_kln, N_k, s_n) = HarmonicOscillatorsTestCase().sample()
```

Generate energy samples with specified parameters.

```
>>> testcase = HarmonicOscillatorsTestCase(O_k=[0, 1, 2, 3, 4], K_k=[1, 2, 4, 8,␣
↪16])
>>> (x_kn, u_kln, N_k, s_n) = testcase.sample(N_k=[10, 20, 30, 40, 50])
```

Test sampling in different output modes.

```
>>> (x_kn, u_kln, N_k) = testcase.sample(N_k=[10, 20, 30, 40, 50], mode='u_kln')
>>> (x_n, u_kn, N_k, s_n) = testcase.sample(N_k=[10, 20, 30, 40, 50], mode='u_kn')
```

Generate test case with exponential distributions.

Parameters **O_k** : np.ndarray, float, shape=(n_states)

Offset parameters for each state.

K_k : np.ndarray, float, shape=(n_states)

Force constants for each state.

beta : float, optional, default=1.0

Inverse temperature.

Notes

We assume potentials of the form U(x) = (k / 2) * (x - o)^2 Here, k and o are the corresponding entries of O_k and K_k. The equilibrium distribution is given analytically by p(x;beta,K) = sqrt[(beta K) / (2 pi)] exp[-beta K (x-x_0)**2 / 2] The dimensionless free energy is therefore f(beta,K) = - (1/2) * ln[(2 pi) / (beta K)]

classmethod evenly_spaced_oscillators(*n_states*, *n_samples_per_state*, *lower_O_k=1.0*, *upper_O_k=5.0*, *lower_k_k=1.0*, *upper_k_k=3.0*)

Generate samples from evenly spaced harmonic oscillators.

Parameters **n_states** : np.ndarray, int

number of states

n_samples_per_state : np.ndarray, int

number of samples per state. The total number of samples n_samples will be equal to n_states * n_samples_per_state

lower_O_k : float, optional, default=1.0

Lower bound of O_k values

upper_O_k : float, optional, default=5.0

Upper bound of O_k values

lower_k_k : float, optional, default=1.0

Lower bound of O_k values

upper_k_k : float, optional, default=3.0

Upper bound of k_k values

Returns name: str

Name of testsystem

testsystem : TestSystem

The testsystem object

x_n : np.ndarray, shape=(n_samples)

Coordinates of the samples

u_kn : np.ndarray, shape=(n_states, n_samples)

Reduced potential energies

N_k : np.ndarray, shape=(n_states)

Number of samples drawn from each state

s_n : np.ndarray, shape=(n_samples)

State of origin of each sample

sample(*N_k=[10, 20, 30, 40, 50], mode='u_kn', seed=None*)

Draw samples from the distribution.

Parameters **N_k** : np.ndarray, int

number of samples per state

mode : str, optional, default='u_kn'

If 'u_kln', return K x K x N_max matrix where u_kln[k,l,n] is reduced potential of sample n from state k evaluated at state l. If 'u_kn', return K x N_tot matrix where u_kn[k,n] is reduced potential of sample n (in concatenated indexing) evaluated at state k.

seed: int, optional, default=None. Provides control over the random seed for replicability.

Returns if mode == 'u_kn':

x_n : np.ndarray, shape=(n_states*n_samples), dtype=float

x_n[n] is sample n (in concatenated indexing)

u_kn : np.ndarray, shape=(n_states, n_states*n_samples), dtype=float

u_kn[k,n] is reduced potential of sample n (in concatenated indexing) evaluated at state k.

N_k : np.ndarray, shape=(n_states), dtype=float

N_k[k] is the number of samples generated from state k

s_n : np.ndarray, shape=(n_samples), dtype='int'

s_n is the state of origin of x_n

x_kn : np.ndarray, shape=(n_states, n_samples), dtype=float

1D harmonic oscillator positions

u_kln : np.ndarray, shape=(n_states, n_states, n_samples), dytpe=float, only if mode='u_kln'

u_kln[k,l,n] is reduced potential of sample n from state k evaluated at state l.

N_k : np.ndarray, shape=(n_states), dtype=int32

N_k[k] is the number of samples generated from state k

pymbar.testsystems.timeseries.**correlated_timeseries_example**(*N=10000, tau=5.0, seed=None*)

Generate synthetic timeseries data with known correlation time.

Parameters **N** : int, optional

length (in number of samples) of timeseries to generate

tau : float, optional

correlation time (in number of samples) for timeseries

seed : int, optional

If not None, specify the numpy random number seed.

Returns **dih** : np.ndarray, shape=(num_dihedrals), dtype=float

dih[i,j] gives the dihedral angle at traj[i] correponding to indices[j].

Notes

Synthetic timeseries generated using bivariate Gaussian process described by Janke (Eq. 41 of Ref. [1]).

As noted in Eq. 45-46 of Ref. [1], the true integrated autocorrelation time will be given by tau_int = (1/2) coth(1 / 2 tau) = (1/2) (1+rho)/(1-rho) which, for tau >> 1, is approximated by tau_int = tau + 1/(12 tau) + O(1/tau^3) So for tau >> 1, tau_int is approximately the given exponential tau.

References

[R11]

Examples

Generate a timeseries of length 10000 with correlation time of 10.

```
>>> A_t = correlated_timeseries_example(N=10000, tau=10.0)
```

Generate an uncorrelated timeseries of length 1000.

```
>>> A_t = correlated_timeseries_example(N=1000, tau=1.0)
```

Generate a correlated timeseries with correlation time longer than the length.

```
>>> A_t = correlated_timeseries_example(N=1000, tau=2000.0)
```

class pymbar.testsystems.exponential_distributions.**ExponentialTestCase**(*rates=[1, 2, 3, 4, 5], beta=1.0*)

Test cases using exponential distributions.

Examples

Generate energy samples with default parameters.

```
>>> testcase = ExponentialTestCase()
>>> [x_kn, u_kln, N_k] = testcase.sample()
```

Retrieve analytical properties.

```
>>> analytical_means = testcase.analytical_means()
>>> analytical_variances = testcase.analytical_variances()
>>> analytical_standard_deviations = testcase.analytical_standard_deviations()
>>> analytical_free_energies = testcase.analytical_free_energies()
>>> analytical_x_squared = testcase.analytical_x_squared()
```

Generate energy samples with default parameters in one line.

```
>>> [x_kn, u_kln, N_k] = ExponentialTestCase().sample()
```

Generate energy samples with specified parameters.

```
>>> testcase = ExponentialTestCase(rates=[1., 2., 3., 4., 5.])
>>> [x_kn, u_kln, N_k] = testcase.sample(N_k=[10, 20, 30, 40, 50])
```

Test sampling in different output modes.

```
>>> [x_kn, u_kln, N_k] = testcase.sample(N_k=[10, 20, 30, 40, 50], mode='u_kln')
>>> [x_n, u_kn, N_k, s_n] = testcase.sample(N_k=[10, 20, 30, 40, 50], mode='u_kn')
```

Generate test case with exponential distributions.

Parameters **rates** : np.ndarray, float, shape=(n_states)

Rate parameters (e.g. lambda) for each state.

beta : float, optional, default=1.0

Inverse temperature.

Notes

We assume potentials of the form U(x) = lambda x.

analytical_free_energies()

Return the FE: -log(Z)

classmethod evenly_spaced_exponentials(*n_states*, *n_samples_per_state*, *lower_rate=1.0*, *upper_rate=3.0*)

Generate samples from evenly spaced exponential distributions.

Parameters **n_states** : np.ndarray, int

number of states

n_samples_per_state : np.ndarray, int

number of samples per state. The total number of samples n_samples will be equal to n_states * n_samples_per_state

lower_O_k : float, optional, default=1.0

Lower bound of O_k values

upper_O_k : float, optional, default=5.0

Upper bound of O_k values

lower_k_k : float, optional, default=1.0

Lower bound of O_k values

upper_k_k : float, optional, default=3.0

Upper bound of k_k values

Returns name: str

Name of testsystem

testsystem : TestSystem

The testsystem object

x_n : np.ndarray, shape=(n_samples)

Coordinates of the samples

u_kn : np.ndarray, shape=(n_states, n_samples)

Reduced potential energies

N_k : np.ndarray, shape=(n_states)

Number of samples drawn from each state

s_n : np.ndarray, shape=(n_samples)

State of origin of each sample

sample(*N_k=(10, 20, 30, 40, 50), mode='u_kln', seed=None*)

Draw samples from the distribution.

Parameters **N_k** : np.ndarray, int

number of samples per state

mode : str, optional, default='u_kln'

If 'u_kln', return K x K x N_max matrix where u_kln[k,l,n] is reduced potential of sample n from state k evaluated at state l. If 'u_kn', return K x N_tot matrix where u_kn[k,n] is reduced potential of sample n (in concatenated indexing) evaluated at state k.

seed: int, optional, default=None. Provides control over the random seed for replicability.

Returns if mode == 'u_kn':

x_n : np.ndarray, shape=(n_states*n_samples), dtype=float

x_n[n] is sample n (in concatenated indexing)

u_kn : np.ndarray, shape=(n_states, n_states*n_samples), dtype=float

u_kn[k,n] is reduced potential of sample n (in concatenated indexing) evaluated at state k.

N_k : np.ndarray, shape=(n_states), dtype=float

N_k[k] is the number of samples generated from state k

s_n : np.ndarray, shape=(n_samples), dtype='int'

s_n is the state of origin of x_n

x_kn : np.ndarray, shape=(n_states, n_samples), dtype=float

1D harmonic oscillator positions

u_kln : np.ndarray, shape=(n_states, n_states, n_samples), dytpe=float, only if mode='u_kln'

u_kln[k,l,n] is reduced potential of sample n from state k evaluated at state l.

N_k : np.ndarray, shape=(n_states), dtype=float

N_k[k] is the number of samples generated from state k `pymbar.testsystems.gaussian_work.`**`gaussian_work_example`**(*N_F=200*, *N_R=200*, *mu_F=2.0*, *DeltaF=None*, *sigma_F=1.0*, *seed=None*)

Generate samples from forward and reverse Gaussian work distributions.

Parameters N_F : int, optional

number of forward measurements (default: 200)

N_R : float, optional

number of reverse measurements (default: 200)

mu_F : float, optional

mean of forward work distribution (default: 2.0)

DeltaF : float, optional

the free energy difference, which can be specified instead of mu_F (default: None)

sigma_F : float, optional

variance of the forward work distribution (default: 1.0)

seed : int, optional

If not None, specify the numpy random number seed. Old state is restored after completion.

Returns w_F : np.ndarray, dtype=float

forward work values

w_R : np.ndarray, dtype=float

reverse work values

Notes

By the Crooks fluctuation theorem (CFT), the forward and backward work distributions are related by

P_R(-w) = P_F(w) exp[DeltaF - w]

If the forward distribution is Gaussian with mean mu_F and std dev sigma_F, then

P_F(w) = (2 pi)^{-1/2} sigma_F^{-1} exp[-(w - mu_F)^2 / (2 sigma_F^2)]

With some algebra, we then find the corresponding mean and std dev of the reverse distribution are

mu_R = - mu_F + sigma_F^2 sigma_R = sigma_F exp[mu_F - sigma_F^2 / 2 + Delta F]

where all quantities are in reduced units (e.g. divided by kT).

Note that mu_F and Delta F are not independent! By the Zwanzig relation,

E_F[exp(-w)] = int dw exp(-w) P_F(w) = exp[-Delta F]

which, with some integration, gives

Delta F = mu_F + sigma_F^2/2

which can be used to determine either mu_F or DeltaF.

Examples

Generate work values with default parameters.

```
>>> [w_F, w_R] = gaussian_work_example()
```

Generate 50 forward work values and 70 reverse work values.

```
>>> [w_F, w_R] = gaussian_work_example(N_F=50, N_R=70)
```

Generate work values specifying the work distribution parameters.

```
>>> [w_F, w_R] = gaussian_work_example(mu_F=3.0, sigma_F=2.0)
```

Generate work values specifying the work distribution parameters, specifying free energy difference instead of mu_F.

```
>>> [w_F, w_R] = gaussian_work_example(mu_F=None, DeltaF=3.0, sigma_F=2.0)
```

Generate work values with known seed to ensure reproducibility for testing.

```
>>> [w_F, w_R] = gaussian_work_example(seed=0)
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[R11] Janke W. Statistical analysis of simulations: Data correlations and error estimation. In ‘Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms’. NIC Series, VOl. 10, pages 423-445, 2002.

Python Module Index

p

Index

A

C

D

E

G

H

I

M

N

P

S

W