emus Documentation

Release

Author

April 11, 2016

Contents:

CHAPTER 1

Quickstart

This guide covers perform typical tasks with the EMUS package. The required data files, as well as a (slightly modified) script containing all of the python commands used below, can be found in the examples directory of the package in the AlaDipeptide_1D directory. The guide will make use of the numpy and matplotlib packages.

1.1 Loading from WHAM-like Formats

The usutils module provides a method that loads data in the format used by WHAM. It outputs the trajectory in collective variable space as well as the $\psi_i j(x_n)$ values.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import usutils as uu
>>>
>>> # Define Simulation Parameters
>>> T = 310                                # Temperature in Kelvin
>>> meta_file = 'wham_meta.txt'            # Path to Meta File
>>> dim = 1                                # 1 Dimensional CV space.
>>> period = 360                           # Dihedral Angles periodicity
>>>
>>> # Load data
>>> psis, cv_trajs = uu.data_from_WHAMmeta('wham_meta.txt',dim,T=T,period=period)
```

1.2 Calculating the PMF

We can now build the EMUS object (this automatically calculates the relative normalization constants according to the first EMUS iteration).

```
>>> from emus import emus
>>> EM = emus(psis,cv_trajs)
```

To calculate the potential of mean force, we provide the number of histogram bins and the range of the collective variable, and call the appropriate method of the EMUS object.

```
>>> domain = ((-180.0,180.))               # Range of dihedral angle values
>>> pmf = EM.calc_pmf(domain,nbins=60)      # Calculate the pmf
```

We can now plot the potential of mean force using pyplot or other tools. Note that this returns the unitless free energy by default: the user can either multiply the pmf by $k_B T$ in postprocessing, or specify $k_B T$ as a parameter for calc_pmf.

```
>>> centers = np.linspace(-177,177,60)  # Center of the histogram bins
>>> plt.plot(centers,pmf)
>>> plt.show()
```

1.3 Estimating Window Partition Functions

Upon creation, the EMUS object already estimates the relative partition function (denoted z) of each window using the EMUS estimator. These are contained in the object, and can be accessed directly.

```
>>> print EM.z
```

The EMUS object also has the ability to calculate the relative partition functions from the MBAR estimator. This requires solving a self-consistent iteration. The nMBAR parameter specifies the maximum number of iterations. Note that truncating early still provides a consistent estimator, and introduces no systematic bias.

```
>>> z_MBAR_1, F_MBAR_1 = EM.calc_zs(nMBAR=1)
>>> z_MBAR_2, F_MBAR_2 = EM.calc_zs(nMBAR=2)
>>> z_MBAR_5, F_MBAR_5 = EM.calc_zs(nMBAR=5)
>>> z_MBAR_1k, F_MBAR_1k = EM.calc_zs(nMBAR=1000)
```

We can plot the unitless window free energies for each max iteration number to see how our estimates converge.

```
>>> plt.plot(-np.log(EM.z),label="Iteration 0")
>>> plt.plot(-np.log(z_MBAR_1),label="Iteration 1")
>>> plt.plot(-np.log(z_MBAR_2),label="Iteration 2")
>>> plt.plot(-np.log(z_MBAR_5),label="Iteration 5")
>>> plt.plot(-np.log(z_MBAR_1k),label="Iteration 1k")
>>> plt.show()
```

The pmf can be constructed using these values for the relative partition functions. [1]

```
>>> MBARpmf = EM.calc_pmf(domain,nbins=60,z=z_MBAR_1k)
```

1.4 Calculating Averages

It is possible to use the EMUS package to calculate the averages of functions. Here, we will calculate the probability that the dihedral takes values between 25 and 100 degrees (this roughly corresponds to the molecule being in the C7 axial basin). This is equivalent to the average of an indicator function that is 1 if the molecule is in the desired configuration and 0 otherwise. First, we construct the timeseries of this function for each umbrella. Note that if the EMUS object was constructed with the collective variable trajectories, they are contained at `EM.cv_trajs`. [2]

```
>>> fdata = [(traj>25) & (traj<100) for traj in EM.cv_trajs]
```

We can now calculate the probability of being in this state.

```
>>> prob_C7ax = EM.calc_obs(fdata)
>>> print prob_C7ax
```

[1] Technically speaking, this is mixing estimators: the z's are being estimated using the MBAR estimator, whereas the pmf is estimated using the MBAR z's, but with the EMUS estimator (MBAR iteration 0). However, in practice the majority of the error comes from estimating the normalization constant. Consequently, the estimator used for estimating the pmf affects the results much less.

[2] The following code relies on True and False having integer comprehensions as 1 and 0, respectively. If, hypothetically speaking, the programmer has done something like in `False=5` earlier in the code, then this won't give the correct answer. Then again, if the programmer did this they are probably a bad person.

CHAPTER 2

Data Structures

This section discusses commonly used variables and naming conventions used throughout the EMUS package. These are not strict standards: the EMUS package attempts to duck-type as much as possible to interpret user input. The documentation below is intended to indicate approximately what form inputs can take, and to aid comprehension of the source code.

2.1 Parameters for Harmonic windows

These variables contain parameters of the window sampling system.

centers Two-dimensional array-like containing the center of each harmonic window in collective variable space. The first index corresponds to the window index, and the second to the collective variable index: `centers[2]` gives the coordinate of the center of the third window.

fks Two-dimensional array-like containing the force constant for each window. The syntax and format is the same as for centers.

kTs One-dimensional array-like with the Boltzmann factor for each window, where the index corresponds to the window index. Alternatively, if the Boltzmann factor is the same for all windows, EMUS will accept a scalar value for.

period One-dimensional array-like encoding the period of the collective variable. `period[i]` gives the periodicity of the i'th collective variable. If the periodicity in that dimension is none, a value of None is used. For instance, `period=[360.,1,None]` specifies that the first collective variable has a period of 360, the second has one of 1, and that the third is aperiodic.

2.2 Data from Sampling

cv_trajs Two-dimensional array-like containing the trajectories in the collective variable space. The first dimension corresponds to the state index and the second to the timepoint. `cv_trajs[i]` gives the the collective variable trajectory of the i'th state, and `cv_trajs[i][n]` gives the value of the collective variable of state i at the nth time point. Note the trajectories can have the same number of time points: `cv_trajs[i]` and `cv_trajs[j]` may have different lengths.

psis Three-dimensional array-like with the values of ψ_j evaluated at each point in the trajectory. As above, the first and second indices are the state index and timepoint, respectively. The third index is the state where ψ is being evaluated: `psis[i][n][j]` returns the value of $\psi_j\left(X_n^i\right)$. If the neighborlist functionality is used, the third index does not need to span over all of the windows, only nearby ones (this introduces slight systematic bias into the estimator).

fdata Two-dimensional array-like containing the values of an observable. The data structure is similar to cv_trajs: `fdata[i][n]` gives the value of the observable evaluated in state i at timepoint n.

neighbors Two-dimensional array-like used for the neighborlist functionality. In practice, it is often known in advance that many combinations of i and j will result in values of ψ that are effectively zero (for instance, two harmonic windows far away in collective variable space). In these situations, it is cheaper to only calculate and store psis for neighboring states, and `psis[i][n][j]` will return the return ψ for the j'th *neighbor*. The `neighbors` data element then gives the true indeces of each neighbor: `neighbors[i][j]` returns the index of the j'th state neighboring state i. For harmonic windows, a neighborlist can be constructed using the neighbors_harmonic function.

iats One-dimensional array-like giving the integrated autocorrelation time for the values of ψ each state (iat). `iats[i]` gives the iat for state i.[#iatnote]_

2.3 Other Conventions

CHAPTER 3

EMUS Modules

Contents:

3.1 autocorrelation module

Tools for analyzing the autocorrelation of a time series

`autocorrelation.`**`autocorrfxn`**(*timeseries*, *lagmax*)

`autocorrelation.`**`icce`**(*timeseries*, *lagmax=None*)
Initial convex correlation time estimator

`autocorrelation.`**`ipce`**(*timeseries*, *lagmax=None*)
Initial positive correlation time estimator

3.2 avar module

Library with routines associated with the asymptotic variance of the first EMUS iteration. These estimates rely on estimates of

`avar.`**`avar_obs`**(*psis*, *z*, *F*, *f1data*, *f2data=None*, *neighbors=None*, *iat_method='ipce'*)
Estimates the asymptotic variance in the ratio of two observables. If f2data is not given, it just calculates the asymptotic variance associated with the average of f1.

Parameters **psis** : 3D data structure

Data structure containing psi values. See documentation in emus.py for a detailed explanation.

z : 1D array

Array containing the normalization constants

F : 2D array

Overlap matrix for the first EMUS iteration.

f1data : 2D data structure

Trajectory of observable in the numerator. First dimension corresponds to the umbrella index and the second to the point in the trajectory.

f2data : 2D data structure, optional

Trajectory of observable in the denominator.

neighbors : 2D array, optional

List showing which states neighbor which. See neighbors_harmonic in usutils for explanation.

iat_method : string, optional

Method used to estimate autocorrelation time. See the documentation for the avar module.

Returns **errvals** : ndarray

Array of length L (no. windows) where the i'th value corresponds to the contribution to the error from window i.

iatvals : ndarray

Array of length L (no. windows) where the i'th value corresponds to the iat for window i.

avar.**avar_obs_diff**(*psis*, *z*, *F*, *f1data*, *g1data*, *f2data=None*, *g2data=None*, *neighbors=None*, *iat_method='ipce'*)

Estimates the asymptotic variance of the estimate of the ratio $< f_1 > / < f_2 > - < g_1 > / < g_2 >$ observables. If f2data and g2data are not given, they are set to 1.

Parameters **psis** : 3D data structure

Data structure containing psi values. See documentation in emus.py for a detailed explanation.

z : 1D array

Array containing the normalization constants

F : 2D array

Overlap matrix for the first EMUS iteration.

f1data : 2D data structure

Trajectory of observable in the numerator in the first term of the difference. First dimension corresponds to the umbrella index and the second to the point in the trajectory.

g1data : 2D data structure

Trajectory of observable in the numerator in the second term of the difference.

f2data : 2D data structure, optional

Trajectory of observable in the denominator in the first term of the difference.

g2data : 2D data structure, optional

Trajectory of observable in the denominator in the second term of the difference.

neighbors : 2D array, optional

List showing which states neighbor which. See neighbors_harmonic in usutils for explanation.

iat_method : string, optional

Method used to estimate autocorrelation time. See the documentation for the avar module.

Returns **errvals** : ndarray

Array of length L (no. windows) where the i'th value corresponds to the contribution to the error from window i.

iatvals : ndarray

Array of length L (no. windows) where the i'th value corresponds to the iat for window i.

`avar.`**`avar_zfe`**(*psis*, *z*, *F*, *um1*, *um2*, *neighbors=None*, *iat_method='ipce'*)

Estimates the asymptotic variance in the free energy difference between windows um2 and um1, -k_B T log(z_2/z_1). In the code, we arbitrarily denote um2 as 'k' and um1 as 'l' for readability.

Parameters **psis** : 3D data structure

Data structure containing psi values. See documentation in emus.py for a detailed explanation.

z : 1D array

Array containing the normalization constants

F : 2D array

Overlap matrix for the first EMUS iteration.

state_1 : int

Index of the first state.

state_2 : ind

Index of the second state.

neighbors : 2D array, optional

list showing which states neighbor which. See neighbors_harmonic in umbrellautils for explanation.

iat_method : string, optional

Method used to estimate autocorrelation time. Default is the initial positive correlation estimator ('ipce'), but also supported is the initial convex correlation estimator ('icce') and the acor algorithm ('acor') See Geyer, Stat. Sci. 1992 and Jonathan Goodman's acor documentation for reference.

Returns **errvals** : ndarray

Array of length L (no. windows) where the i'th value corresponds to the contribution to the error from window i.

iatvals : ndarray

Array of length L (no. windows) where the i'th value corresponds to the iat for window i.

`avar.`**`getAllocations`**(*importances*, *N_is*, *newWork*)

Calculates the optimal allocation of sample points These are the optimal weights for To deal with negative weights, it removes all the negative weights, and calculates the weights for the resulting subproblem.

3.3 emus module

Module containing the emus object.

class `emus.emus`(*psis*, *cv_trajs=None*, *neighbors=None*, *k_B=0.0019872041*)

Class containing methods and data for the EMUS algorithm. An EMUS object has the following data structures which can be interacted with or modified:

self.psis (3D array): array containing values of the biases in each state. self.cv_trajs (2D Array): array containing the trajectories in cv space. None if not used. self.z (1d array): array containing the normalization constants. Calculated according to the first iteration of EMUS. self.F (2d array): F matrix for the first iteration of EMUS. self.iats (1d array): array containing integrated autocorrelation times of $\psi_{ii}(x)$ in each window.

Methods

`avar_zfe`(state_1, state_2)	Calculates the asymptotic variance for the free energy difference between the two stat
`calc_obs`(fdata[, z])	Estimates the average of an observable function.
`calc_pmf`(domain[, cv_trajs, nbins, kT, z])	Calculates the potential of mean force for the system.
`calc_zs`([nMBAR, tol, use_iats, iats])	Calculates the normalization constants for the states.

`avar_zfe`(*state_1*, *state_2*)

Calculates the asymptotic variance for the free energy difference between the two states specified.

Parameters **state_1** : int

Index of the first state.

state_2 : int

Index of the second state.

Returns **errs** : 1D array

Array containing each state's contribution to the asymptotic error. The total asymptotic error is taken by summing the entries.

`calc_obs`(*fdata*, *z=None*)

Estimates the average of an observable function.

Parameters **fdata** : 2d array-like

Two dimensional data structure where the first dimension corresponds to the state index, and the second to the value of the observable at that time point. Must have the same number of data-points as the collective variable trajectory.

z : 1D array, optional

User-provided values for the normalization constants. If not provided, uses values from the first iteration.

Returns **favg** : float

The estimated average of the observable.

`calc_pmf`(*domain*, *cv_trajs=None*, *nbins=100*, *kT=1.0*, *z=None*)

Calculates the potential of mean force for the system.

Parameters **domain** : tuple

Tuple containing the dimensions of the space over which to construct the pmf, e.g. (-180,180) or ((0,1),(-3.14,3.14))

nbins : int or tuple, optional

Number of bins to use. If int, uses that many bins in each dimension. If tuple, e.g. (100,20), uses 100 bins in the first dimension and 20 in the second.

cvtrajectories : 2D array-like, optional

Two dimensional data structure with the trajectories in cv space. The first dimension is the state where the data was collected, and the second is the value in cv space. If not provided, uses trajectory given in the constructor.

z : 1D array, optional

User-provided values for the normalization constants If not provided, uses values from the first iteration of EMUS.

kT : float, optional

Value of kT to scale the PMF by. If not provided, set to 1.0

Returns **pmf** : nd array

Returns the potential of mean force as a d dimensional array, where d is the number of collective variables.

calc_zs (*nMBAR=0*, *tol=1e-08*, *use_iats=False*, *iats=None*)

Calculates the normalization constants for the states.

Parameters **nMBAR** : int, optional (default 0)

Maximum number of MBAR iterations to perform.

tol : float, optional (default 1.0E-8)

If the relative residual falls beneath the tolerance, the MBAR iteration is truncated.

use_iats : bool, optional

If true, estimate integrated autocorrelation time in each MBAR iteration. Likely unnecessary unless dynamics are expected to be drastically different in each state. If iats is provided, the iteration will use those rather than estimating them in each step.

iats : 1D array, optional

Array of size L (no. states) with values of the integrated autocorrelation time estimated in each state. These values will be used in each iteration. Overrides use_iats.

Returns **z** : 1D array

Values for the Normalization constant in each state.

F : 2D array

Matrix to take the eigenvalue of for MBAR.

iats 1D array

Estimated values of the autocorrelation time. Only returned if use_iats is true.

3.4 emusroutines module

Container for the primary EMUS routines.

emusroutines.**calc_obs** (*psis*, *z*, *f1data*, *f2data=None*)

Estimates the value of an observable or ratio of observables.

Parameters **psis** : 3D data structure

Data structure containing psi values. See documentation in emus.py for a detailed explanation.

z : 1D array

Array containing the normalization constants

f1data : 2D data structure

Trajectory of observable in the numerator. First dimension corresponds to the umbrella index and the second to the point in the trajectory.

f2data : 2D data structure, optional

Trajectory of observable in the denominator.

Returns avg : float

The estimate of <f_1>/<f_2>.

`emusroutines.`**`emus_iter`**(*psis*, *Avals=None*, *neighbors=None*, *return_iats=False*, *iat_method='ipce'*)

Performs one step of the the EMUS iteration.

Parameters psis : 3D data structure

Data structure containing psi values. See documentation in emus.py for a detailed explanation.

Avals : 2D matrix, optional

Weights in front of ψ in the overlap matrix.

neighbors : 2D array, optional

List showing which states neighbor which. See neighbors_harmonic in usutils.

return_iats : bool, optional

Whether or not to calculate integrated autocorrelation times of $\psi_i i^*$ for each window.

iat_method : string, optional

Routine to use for calculating said iats. Accepts 'ipce', 'acor', and 'icce'.

Returns z : 1D array

Normalization constants for each state

F : 2D array

The overlap matrix constructed for the eigenproblem.

iats : 1D array

If return_iats chosen, returns the iats that have been estimated.

`emusroutines.`**`make_pmf`**(*cv_trajs*, *psis*, *domain*, *z*, *nbins=100*, *kT=1.0*)

Calculates the free energy surface for an umbrella sampling run.

Parameters cv_trajs : 2D data structure

Data structure containing trajectories in the collective variable space. See documentation in emus object for more detail.

psis : 3D data structure

Data structure containing psi values. See documentation in emus object for a detailed explanation.

domain : tuple

Tuple containing the dimensions of the space over which to construct the pmf, e.g. (-180,180) or ((0,1),(-3.14,3.14)) z (1D array or list): Normalization constants for each state

nbins : int or tuple, optional

Number of bins to use. If int, uses that many bins in each dimension. If tuple, e.g. (100,20), uses 100 bins in the first dimension and 20 in the second.

kT : float, optional

Value of kT to scale the PMF by. If not provided, set to 1.0

Returns pmf : nd array

Returns the potential of mean force as a d dimensional array, where d is the number of collective variables.

3.5 linalg module

Collection of linear algebra routines used in the EMUS algorithm and associated error analysis.

`linalg.`**`groupInverse`**(*M*)

Computes the group inverse of stochastic matrix using the algorithm given by Golub and Meyer in: G. H. Golub and C. D. Meyer, Jr, SIAM J. Alg. Disc. Meth. 7, 273- 281 (1986)

Parameters M : ndarray

A square matrix with index 1.

Returns grpInvM : ndarray

The group inverse of M.

`linalg.`**`old_stationary_distrib`**(*F*, *fix=None*, *residtol=1e-10*, *max_iter=100*)

`linalg.`**`stationary_distrib`**(*F*, *residtol=1e-10*, *max_iter=100*)

Calculates the eigenvector of the matrix F with eigenvalue 1 (if it exists).

Parameters F : ndarray

A matrix known to have a single left eigenvector with eigenvalue 1.

residtol : float or scalar

To improve the accuracy of the computation, the algorithm will “polish” the final result using several iterations of the power method, z^T F = z^T. Residtol gives the tolerance for the associated relative residual to determine convergence.

maxiter : int

Maximum number of iterations to use the power method to reduce the residual. In practice, should never be reached.

Returns z : ndarray

The eigenvector of the matrix F with eigenvalue 1. For a Markov chain stationary distribution, this is the stationary distribution. Normalization is chosen s.t. entries sum to one.

3.6 usutils module

Module containing methods useful for analyzing umbrella sampling calculations that do not rely directly on the EMUS estimator.

`usutils.`**`calc_harmonic_psis`**(*cv_traj*, *centers*, *fks*, *kTs*, *period=None*)

Calculates the values of each bias function from a trajectory of points in a single state.

Parameters **cv_traj** : array-like

Trajectory in collective variable space. Can be 1-dimensional (one cv) or 2-dimensional (many cvs). The first dimension is the time index, and (optional) second corresponds to the collective variable.

centers : array-like

The locations of the centers of each window. The first dimension is the window index, and the (optional) second is the collective variable index.

fks : scalar or 2darray

If array or list, data structure where the first dimension corresponds to the window index and the second corresponds to the collective variable. If scalar, windows are assumed to have that force constant in every dimension.

kTs : scalar or 2darray

1D array with the Boltzmann factor or a single value which will be used in all windows. Default value is the scalar 1.

period : 1D array-like or float, optional

Period of the collective variable e.g. 360 for an angle. If None, all collective variables are taken to be aperiodic. If scalar, assumed to be period of each collective variable. If 1D iterable with each value a scalar or None, each cv has periodicity of that size.

Returns **psis** : 2D array

The values of the bias functions at each point in the trajectory evaluated at the windows given. First axis corresponds to the timepoint, the second to the window index.

`usutils.`**`data_from_WHAMmeta`**(*filepath*, *dim*, *T=None*, *k_B=0.0019872041*, *period=None*)

Reads data saved on disk according to the format used by the WHAM implementation by Grossfield.

Parameters **filepath** : string

The path to the meta file.

dim : int

The number of dimensions of the cv space.

T : scalar, optional

Temperature of the system.

k_B : scalar, optional

Boltzmann Constant for the system. Default is in kCal/mol

period : 1D array-like or float, optional

Variable with the periodicity information of the system. See the Data Structures section of the documentation for a detailed explanation.

Returns **psis** : 2D array

The values of the bias functions at each point in the trajectory evaluated at the windows given. First axis corresponds to the timepoint, the second to the window index.

cv_trajs : 2D array-like

Two dimensional data structure with the trajectories in cv space. The first dimension is the state where the data was collected, and the second is the value in cv space.

`usutils.`**`neighbors_harmonic`**(*centers*, *fks*, *kTs=1.0*, *period=None*, *nsig=4*)

Calculates neighborlist for harmonic windows. Neighbors are chosen such that neighboring umbrellas are no more than nsig standard deviations away on a flat potential.

Parameters centers : 2darray

The locations of the centers of each window. The first dimension is the window index, and the second is the collective variable index.

fks : 2darray or scalar

If array or list, data structure where the first dimension corresponds to the window index and the second corresponds to the collective variable. If scalar, windows are assumed to have that force constant in every dimension.

kTs : 2darray or float

1D array with the Boltzmann factor or a single value which will be used in all windows. Default value is the scalar 1.

period : 1D array-like or float

Period of the collective variable e.g. 360 for an angle. If None, all collective variables are taken to be aperiodic. If scalar, assumed to be period of each collective variable. If 1D iterable with each value a scalar or None, each cv has periodicity of that size.

nsig : scalar

Number of standard deviations of the gaussians to include in the neighborlist.

Returns nbrs : 2d list

List where element i is a list with the indices of all windows neighboring window i.

`usutils.`**`parse_metafile`**(*filepath*, *dim*)

Parses the meta file located at filepath. Assumes Wham-like Syntax.

Parameters filepath : string

The path to the meta file.

dim : int

The number of dimensions of the cv space.

Returns traj_paths : list of strings

A list containing the paths to the trajectories for each window.

centers : 2D array of floats

Array with the center of each harmonic window. See calc_harm_psis for syntax.

fks : 2D array of floats

Array with the force constants for each harmonic window. See calc_harm_psis for syntax.

iats : 1D array of floats or None

Array with the integrated autocorrelation times of each window. None if not given in the meta file

temps : 1D array of floats or None

Array with the temperature of each window in the umbrella sampling calculation. If not given in the meta file, this will just be None.

`usutils.`**`unpackNbrs`**(*compd_array*, *neighbors*, *L*)

Unpacks an array of neighborlisted data. Currently, assumes axis 0 is the compressed axis.

Parameters compd_array : array-like

The compressed array, calculated using neighborlists

neighbors : array-like

The list or array of ints, containing the indices of the neighboring windows

L : int

The total number of windows.

Returns expd_array : array-like

The expanded array of data

test

test

[1] will be “2” (manually numbered), [2] will be “3” (anonymous auto-numbered), and [3] will be “1” (labeled auto-numbered).

[1] This footnote is labeled manually, so its number is fixed.
[2] This footnote will be labeled “3”. It is the second auto-numbered footnote, but footnote label “2” is already used.
[3]

This autonumber-labeled footnote will be labeled “1”. It is the first auto-numbered footnote and no other footnote with label “1” exists. The order of the footnotes is used to determine numbering, not the order of the footnote references.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index