
enspara Documentation

Release 0.0.1

Justin R. Porter, Maxwell I. Zimmerman, Gregory R. Bowman

May 29, 2020

Contents

1	Installation	3
2	Tutorial	5
3	CARDS	11
4	CLI	15
5	Clustering	17
6	Cookbook	21
7	Pocket Detection	23
8	Transition Path Theory	25
9	API	27
10	Indices and tables	33
	Python Module Index	35
	Index	37

Enspara is primarily a python library, but it also contains a few applications for very common Markov-state model related tasks.

- MSM estimation and manipulation
- Turn-key applications for clustering and analysis
- Transition path theory
- Information theory
- Ragged ndarrays

Contents:

CHAPTER 1

Installation

Enspara can be installed from our github repository in the following way:

1. Confirm Anaconda is installed on local machine and create/activate environment.
2. Clone enspara from github to local machine:

```
git clone https://github.com/bowman-lab/enspara
```

3. Install compiled dependencies of enspara (one frequently runs into problems compiling these locally via *pip*, but if you prefer you can just run *setup.py* and *pip* will try to download and compile these packages):

```
conda install -c conda-forge mdtraj=1.8.0
conda install numpy==1.14
conda install cython
conda install mpi4py -c conda-forge
```

4. Enter enspara from home directory and:

```
python setup.py install
```

If setup failed, *conda install* necessary packages and rerun setup command.

5. Check that you've been successful:

```
cd && python -c 'import enspara'
```


In this short tutorial, we'll walk you through a basic example using `enspara` to make an MSM and do a simple analysis.

2.1 Clustering Trajectories

The first step to analyzing MD data is usually clustering. For simple to moderately-complex clustering tasks, we make this pretty straightforward in `enspara`.

With the *Clustering CLI*, you can cluster the data like so:

```
enspara cluster \  
  --trajectories trajectory-*.xtc \  
  --topology fs-peptide.pdb \  
  --algorithm khybrid \  
  --cluster-number 20 \  
  --subsample 10 \  
  --atoms '(name N or name C or name CA)' \  
  --distances ./fs-khybrid-clusters0020-distances.h5 \  
  --center-features ./fs-khybrid-clusters0020-centers.pickle \  
  --assignments ./fs-khybrid-clusters0020-assignments.h5
```

This will cluster all the trajectories into 20 clusters using the k-hybrid algorithm based on backbone (atoms named N, CA or C, per the `MDTraj DSL`) and output the results (distance, center structures, and assignments) to files named as specified on the command line (`fs-khybrid-clusters0020*`).

2.1.1 Clustering Outputs

Clustering this way will output four files,

1. `fs-khybrid-clusters0020-centers.pickle` is a pickle of a python list containing the frames that were at the center of each cluster center. They are given in the order they were discovered by k -centers clustering, with 0 being the first frame in the dataset.

2. `fs-khybrid-clusters0020-assignments.h5` is an h5 file containing the assignment of each frame of all trajectories to a cluster center. The value at t, i gives which cluster center frame i in trajectory t was assigned to. The i values match the centers file.
3. The distances file `fs-khybrid-clusters0020-distances.h5` is an array (numpy in this case) where the value at t, i gives the distance between frame i in trajectory t (matching the order of the trajectories given by the `--trajectories` flag) and the closest cluster center.

Because these are usually much smaller than your inputs, you can usually drop into a ipython shell (or even better, jupyter notebook) to inspect your values:

```
$ ipython
Python 3.6.0 \Continuum Analytics, Inc.\ (default, Dec 23 2016, 12:22:00)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Then you can load up these files,

```
import pickle

import mdtraj as md
from enspara import ra

assigns = ra.load('fs-khybrid-clusters0020-assignments.h5')
dists = ra.load('fs-khybrid-clusters0020-distances.h5')
```

This uses `enspara`'s `RaggedArray` submodule to load the assignments and distances files. In this case, all your trajectories are uniform length, so you will actually get back a numpy ndarray, but in a more realistic situation when trajectories have different lengths, this would be a ragged array.

We can then ask how many frames got assigned to each cluster and make a plot:

```
import matplotlib.pyplot as plt
import numpy as np

plt.bar(x=range(0, 20), height=np.bincount(assigns.flatten()))
plt.xticks(range(assigns.max()+1))
plt.xlim(-1, assigns.max()+1)
plt.ylabel('Number of Frames Assigned')
plt.xlabel('Cluster Index')
```

And also ask what the distribution of frames per cluster looks like,

```
import matplotlib.pyplot as plt
import numpy as np

plt.hist(np.bincount(assigns.flatten()))
plt.ylabel('Number of Clusters')
plt.xlabel('Frames Assigned')
```

And, using the distances file, we can compute the distribution of distances to the nearest cluster center,

```
import matplotlib.pyplot as plt
```

```
plt.hist(dists.flatten(), bins=100)
plt.ylabel('Number of Frames')
plt.xlabel('Distance to Cluster Center')
```

Next, rather than counting raw frames, we can build a Markov State Model, which does a more rigorous statistical analysis on these data.

2.2 Fitting

Once you've clustered your data, you'll want to build a Markov state model.

2.2.1 Implied Timescales

One way of assessing the quality of your MSM is to look at the implied timescales. In particular, this is often used to choose a lag time. Ideally, your MSM's behavior isn't very dependent on your choice of lag time (i.e. it satisfies the Markov assumption), and so this is usually a good thing to check.

```
from enspara import msm
import numpy as np

# make 20 different lag times (integers) evenly spaced between 10 and 750
lag_times = np.linspace(10, 750, num=20).astype(int)

implied_timescales = []
for time in lag_times:
    m = msm.MSM(
        lag_time=time,
        method=msm.builders.transpose)
    m.fit(assigns)

    implied_timescales.append(
        -time / np.log(msm.eigenspectrum(m.tprobs_, n_eigs=3)[0][1:3])
    )
```

This will calculate the top 3 implied timescales across that range of lag times. Let's plot it and see how it looks:

```
import matplotlib.pyplot as plt

implied_timescales = np.vstack(implied_timescales)

for i in range(implied_timescales.shape[1]):
    plt.plot(lag_times, implied_timescales[:, i],
             label='$\lambda_{%s}$' % (i+1))

plt.show()
```

2.2.2 Fitting an MSM

Once you've got a lagtime you're satisfied with, make an MSM the same way as before (we also could have stored the old one).

```
m = msm.MSM(
    lag_time=10,
    method=msm.builders.transpose)
m.fit(assigns)
```

You can then ask questions about the conformational landscape you've sampled. First, let's find out what the equilibrium probability of each state is with the `eq_probs_` parameter fit by the MSM.

```
import matplotlib.pyplot as plt

plt.bar(x=np.arange(m.n_states_), height=m.eq_probs_)
plt.xticks(range(0, m.n_states_+1))

plt.xlabel("State Index")
plt.ylabel("Equilibrium Probability")
```

The MSM also fits a probability of transition between states, which it keeps in the `tprobs_` member variable.

```
plt.imshow(np.log(m.tprobs_.toarray()))
plt.xticks(range(0, m.n_states_+1))
plt.yticks(range(0, m.n_states_+1))
plt.colorbar(label=r'\ln $P(i \rightarrow j)$')
```

Note: `enspara` makes heavy use of sparse matrices! In this case (although not always, depending on what method is used to build it) the MSM has a sparse `tprobs_`. In this example, we'll convert it to a dense array with `toarray()`, but this is a potentially expensive choice for big MSMs!

Next, we'll analyze the features of the conformational landscape to learn about our protein!

2.3 Analysis

In this section, as an example, we're going to analyze some of the results from the simulation to give an example how to use `enspara`, and how to use an MSM to generate understanding about a protein's conformational landscape.

First, let's load the representative centers from the clustering:

```
import pickle
import mdtraj as md

with open('fs-khybrid-clusters0040-centers.pickle', 'rb') as f:
    ctr_structs = md.join(pickle.load(f))
```

The clustering application saves representative cluster centers as a pickle of `mdtraj` trajectory objects. In this case, they all have the same topology, so we can get away with “`md.join`”ing them once pickle has loaded them. (Note, though, that this can be pretty slow for large lists of trajectories.)

Let's say we're interested in hydrogen bonding patterns in this protein. First, we used the [MDTraj implementation of the Kabsch Sander hydrogen bond](#) formulation to compute all the hydrogen bonds in each frame:

```
hbonds = md.kabsch_sander(ctr_structs)
```

This results in a list of $n \times n$ `scipy sparse matrices`, where the element i, j is the predicted hydrogen bond energy between residues i and j . Combining these energies with the probability estimates from our `msm`, we can show the mean energy for all possible pairs of hydrogen bonds throughout the conformational landscape:

```
weighted_hbond_mtx = sum(p*h for p, h in zip(m.eq_probs_, hbonds)).todense()
plt.imshow(weighted_hbond_mtx, cmap='viridis_r')
plt.colorbar()
```

As you can see, the highest energy hydrogen bonds are the i to $i+4$ hydrogen bonds. This isn't too surprising, since this protein is supposed to fold up into a helix.

Let's discretize these energies into a bond formed/bond broken binary vector for each state:

```
all_hbonds = set()

# accumulate all the possible pairs of residues involved in hbonds
for i in range(len(ctr_structs)):
    donors, acceptors = np.where(hbonds[i].todense() != 0)
    all_hbonds.update([(d, a) for d, a in zip(donors, acceptors)])

# make a list so that it's ordered
all_hbonds = list(all_hbonds)

# this matrix of length n_states will have each binary feature vector
hbond_presence = np.zeros((m.n_states_, len(all_hbonds)),
                          dtype='uint8')

# set each value i, j to one if state i has hbond j.
for i in range(len(ctr_structs)):
    donors, acceptors = np.where(hbonds[i].todense() != 0)

    for a, d in zip(donors, acceptors):
        hbond_id = all_hbonds.index((a, d))
        hbond_presence[i, hbond_id] = 1
```

Then, we can ask what the energy of each of these hydrogen bonds is by taking the dot product between the equilibrium probability vector and the feature vector:

```
p_hbond = np.dot(m.eq_probs_, hbond_presence)

plt.bar(np.arange(len(all_hbonds)), height=np.log(p_hbond))
plt.ylabel("Free Energy")
plt.xlabel("HBond ID")
plt.savefig('./hbond-free-energy.svg')
```

Finally, we might like to know about joint behavior of hydrogen bonds. To do this, let's use `enspara's info_theory` module:

```
from enspara.info_theory import weighted_mi

hbond_mi = weighted_mi(features=hbond_presence, weights=m.eq_probs_)
hbond_mi = hbond_mi - np.diag(np.diag(hbond_mi))

plt.imshow(hbond_mi - np.diag(np.diag(hbond_mi)))
plt.colorbar()
```

And we can ask what which pair of hydrogen bonds has the highest MI:

```
hbond1, hbond2 = np.unravel_index(hbond_mi.argmax(), hbond_mi.shape)

def hbond2str(pair, top):
    return ''.join([str(top.residue(i)) for i in pair])

hbond2str(all_hbonds[hbond1], ctr_structs.top), hbond2str(all_hbonds[hbond2], ctr_
↳structs.top)
```

which returns ('ALA14ALA18', 'ALA13ALA17'), which shouldn't be too surprising, since the formation of one hydrogen bond probably pays most of the entropy cost for its neighboring hydrogen bond to form.

There are lots more things you can do with *enspara*, including *transition path theory*, *CARDS*, and *pocket detection*, so make sure to explore our API documentation!

Before you get started, however, let's download some molecular dynamics data! This MD data is of the Fs peptide (Ace-A_5(AAARA)_3A-NME), which is a fairly common model system for studying protein folding. It was prepared by Robert McGibbon for MSMBuilder3.

```
mkdir fs_peptide && cd fs_peptide
wget https://ndownloader.figshare.com/articles/1030363/versions/1 -O fs_peptide.zip
unzip fs_peptide.zip
```

While you're waiting for that download, check the textbook on MSMs, "An Introduction to Markov State Models and Their Application to Long Timescale Molecular Simulation" edited by Greg Bowman, Vijay Pande, and Frank Noe. That book describes the theoretical and empirical groundwork for a lot of what we'll do.

Anyway, once your download is finished, if you `ls`, you should see a bunch of trajectories (*.xtc), a pdb file `fs-peptide.pdb`, and a few other files. If you do, you're ready to move on.

One of the interesting phenomena that simulations can report on is allosteric communication. Identifying which residues communicate with a target site of interest, like an active site, can help isolate the important regions within a protein.

CARDS is a way of capturing this long-range communication from MD simulations, measuring coupling between every pair of dihedrals in an entire protein. The CARDS methodology has been published in [Singh and Bowman, JCTC, 2017](#).

Using CARDS is a two-step process:

1. Using the implementation within `enspara` to *collect cards* by utilizing the command-line script.
2. Analyzing CARDS data using the CARDS-Reader library (URL) as described in the *analysis* section

3.1 Measure correlations with CARDS

CARDS works by decomposing each rotamer into two sets of states - rotameric states and dynamical states. Dynamical states are determined by whether a dihedral remains in a single rotameric state (ordered) or rapidly transitions between multiple rotameric states (disordered). It then computes pairwise correlations between every pair of dihedrals and their rotameric and dynamical states. Thus there are four types of correlations produced by CARDS.

1. Structure-structure (between rotameric states)
2. Disorder-disorder (between dynamical states)
3. Structure-disorder (rotameric states of one dihedral with dynamical states of another)
4. Disorder-structure (dynamical states of one dihedral with rotameric states of another)

`enspara` features a multi-processing implementation of CARDS that is useful for large systems or datasets. Before you get started you will need to have the following:

1. A directory containing all your trajectory files, any format recognized by `MDTraj` is fine
2. A topology file that corresponds to your trajectory files (if it doesn't have the topology pre-written like in `.h5` files). Again, any format recognizable by `MDTraj` is acceptable.

Once you have these two inputs, you must consider how big of a “buffer-zone” you want to use. To prevent counting spurious transitions as part of the correlated motions of your system, CARDS places a *buffer-zone* around each rotameric-barrier, defining “core-regions” within each rotamer. Thus, a rotamer has only had a “true” transition if it enters a new “core region” than the one it previously occupied. Generally we use buffer-zones that are ~15 degrees on each side of the barrier.

You can read more about buffer zones in the publication [describing CARDS](#)

3.1.1 Running CARDS

Let’s run a simple example of the `collect_cards.py` script. In our case we have some directory containing a series of `.xtc` files that are MD Trajectories. We can catch them all using the wild-card `*.xtc`. We also have our `topology.top` file to load in alongside the trajectories.

```
python /home/username/enspara/enspara/apps/collect_cards.py \  
  --trajectories /path/to/input/*.xtc \  
  --topology /path/to/input/topology.top \  
  --buffer-size 15 \  
  --processes 1 \  
  --matrices /home/username/output_path/cards.pickle \  
  --indices /home/username/output_path/inds.csv \  

```

This is a CARDS run that will use a buffer-width of 15 degrees, and a single core. The outputs are 1) a single pickle file containing a dictionary of the four types of matrices that CARDS generates, and 2) An *inds.csv* file that contains the atom indices that correspond to each row/column in the CARDS matrices.

Specifically, the `cards.pickle` output is a dictionary that contains four matrices. The dictionary keys identify which matrix measures which type of correlation,

3.2 Analyze CARDS data

Analysis of CARDS data can be done using the [CARDS-Reader library](#) As published in CARDS-using papers, there are multiple ways CARDS data can be analyzed, including:

1. Extracting Shannon entropies to measure residue disorder
2. Computing a holistic correlations matrix.
3. Target site analysis

3.2.1 Extracting Shannon entropy to measure residue disorder

The Shannon entropy is an information-theoretic metric that can be used to measure disorder in a dataset. It is computed across a dataset by looking at the population of each bin:

$$H(X) = -\sum_x P(x) \log(P(x))$$

In CARDS, it provides a useful insight into how much any single dihedral moves around across the simulation dataset.

Computing Shannon entropy of how a dihedral moves in inherently a structural phenomena, and conveniently is equivalent to the diagonal of the CARDS matrix corresponding to Structure_Structure correlation (between rotameric states). We also want to be able to understand motion on an amino-acid level, rather than a dihedral level.

In CARDS-Reader we can use the `apps/extract_dihedral_entropy.py` found inside the CARDS-Reader library.


```
python /home/username/cardsReader/apps/extract_dihedral_entropy.py \
  --matrices /home/username/output_path/cards.pickle \
  --indices /home/username/output_path/inds.csv \
  --topology /path/to/input/topology.top \
```

In this script you are simply inputting the same topology file as used in *collect_cards.py* and the outputs from *collect_cards.py*.

The output will be two files, *dihedral_entropy.csv* and *residue_entropy.csv* that will have the entropy for each dihedral (AKA just the diagonal), and the residue-level entropy, which is normalized by the maximum amount of entropy a residue can have. In other words, a residue-level entropy of 0.3 means a residue has ~30% of the maximum possible Shannon entropy value it can have.

3.2.2 Computing holistic correlations

To capture the full pattern of communication into a single dihedral matrix, we can sum the four matrices in *cards.pickle* directly into a single *Holistic communication matrix*.

This is a relatively trivial task, but for convenience, CARDS-Reader has an apps script *apps/generate_holistic_matrix.py* that computes this matrix and saves it.

```
python /home/username/cardsReader/apps/generate_holistic_matrix.py \
  --directory /home/username/output_path/cards.pickle \
```

At its core, CARDS is built on the fundamental idea that the overall communication pattern of a system is based on the combined communication of rotameric and disordered states.

$$I_{Holistic} = I_{Structural} + I_{Disordered}$$

The *generate_holistic_matrix.py* script computes both the Structural-Structural matrix (*Structural_MI.csv*), as well as a single disorder-disorder matrix (*totalDisorder_MI.csv*), which is the sum of the other three matrices. It also outputs the total Holistic communication matrix (*holistic_MI.csv*)

This holistic communication matrix is what we can use to probe overall communication patterns in our system, using techniques like *Target site analysis*, or other methods.

These apps make it easy to do the common tasks associated with building an MSM.

4.1 Clustering

Once you have your simulations run, the first thing you'll want to do is cluster your trajectories based on a parameter of interest. Commonly, you'll want to look at the root mean square deviation of the states or the euclidean distance between some sort of feature vector. You can do this using `apps/cluster.py`

This app is documented in *Clustering App*.

4.2 Implied Timescales

Once you've clustered, you might want to know what lag time is appropriate to use to create your MSM. You can plot eigenvalue motion speed as a function of lag time by using `implied_timescales.py`

The app only requires the assignment files.

```
--assignments path/to/directory/with/file.h5
# This is the file containing assignments
```

However, there are many other parameters that can be set as well.

```
--n-eigenvalues integer
# This is the number of eigenvalues that will be computed for each lag time.
# The default is five.
--lag-times min:max:step
# This is the list of lag times (in frames).
# The default is 5:100:2.
--symmetrization method name
# This is the method to use to enforce detailed balance in the counts matrix.
# The default is transpose.
```

```
--trj-ids trajs
# This will only use given trajectories to compute the implied timescales.
# This is useful for handling assignments for shared state space clusterings.
# The default is none.
--processes integer
# This will set the number of cores to use.
# Because eigenvector decompositions are thread-parallelized, this should
# usually be several times smaller than the number of cores available on
# your machine.
# The default is max(1, cpu_count()/4).
--trim truth statement
# This will turn on ergodic trimming
# The default is False.
--plot path/to/directory/file_name.png
# This is how the plot will save.
--logscale
# This will put the y-axis of the plot on a log scale.
```

Your final submit script should be formatted something like this.

```
python /home/username/enspara/enspara/apps/implied_timescales.py \  
--assignments assignments.h5 \  
--n-eigenvalues 5 \  
--processes 2 \  
--plot implied_timescales.png \  
--logscale
```

Once you have your simulations run, generally the first step in building an MSM is clustering your trajectories based on a parameter of interest. Commonly, you'll want to look at the root mean square deviation of the states or the euclidean distance between some sort of feature vector.

In *enspara*, this functionality is available at three levels of detail.

1. *Apps*. Clustering code is available in a command-line application that is capable of handling much of the book-keeping necessary for more complex clustering operations.
2. *Objects*. Clustering code is wrapped into sklearn-style objects that offer simple API access to clustering algorithms and their parameters.
3. *Functions*. Clustering code is ultimately implemented as functions, which offer the highest degree of control over the function's behavior, but also require the most work on the user's part.

5.1 Clustering App

Clustering functionality is available in *enspara* in the script `apps/cluster.py`. Its help output explains at a granular level of detail what it is capable of, and so here we will seek to provide a high-level discussion of how it can be used.

When clustering, you will need to make a few important choices:

1. What type of data will you be clustering? The app accepts trajectories of coordinates as well as arrays of vectors.
2. Which clustering algorithm will you use? We currently implement k-centers and k-hybrid.
3. How "much" clustering will you do? Both k-centers and k-hybrid require the choice of k-centers stopping criteria, and k-hybrid additionally requires the choice of number of k-medoids refinements.
4. How will you compare frames to one another (i.e. what is your distance function)? Options include RMSD (for coordinates), as well as euclidean and manhattan distances.

5.1.1 A Simple Example

One thing `enspara` excels at is generating fine-grained state spaces by clustering using RMSD as a criterion. This is very fast, and is not only thread-parallelized to use all cores on a single computer (hat tip to MDTraj's blazing fast RMSD calculations), but also can be parallelized across many computers with MPI.

In a simple case, such a clustering will look something like this:

```
python /home/username/enspara/enspara/apps/cluster.py \
  --trajectories /path/to/input/trj1.xtc /path/to/input/trj2.xtc \
  --topology /path/to/input/topology.top \
  --algorithm khybrid \
  --cluster-number 1000 \
  --distances /path/to/output/distances.h5 \
  --center-features /path/to/output/centers.pickle \
  --assignments /path/to/output/assignments.h5
```

This will make 1000 clusters using the k-hybrid clustering algorithm based on all the atomic coordinates in `trj1.xtc` and `trj2.xtc`. Based on the clusters it discovers, it will generate three files:

1. Centers file (`centers.pickle`). This file, which is a python list of `mdtraj.Trajectory` trajectory objects, contains the atomic coordinates that were at the center of each center. If 1000 clusters are discovered, this list will have length 1000.
2. Assignments file (`assignments.h5`). This file assigns each frame in the input to each cluster center (even if sub-sampling is specified). If (i, j) in this array has value n , then the j 'th frame of trajectory `:code:`i` above was found to belong to center n (found in the centers file).
3. Distances file (`distance.h5`). This file gives the distance between each frame (i, j) and the center it is assigned to (found in the assignments file).

5.1.2 Atom Selection and Shared State Spaces

It is also possible to cluster proteins with differing topologies into the same state space. To do this, we rely on the `--atoms` flag to select matching atoms between the two topologies. The `--atoms` flag uses the MDTraj DSL selection syntax to specify which atoms will be loaded from each trajectory.

Imagine we have simulations of a wild-type and point mutant. To specify the the different trajectories and topologies, we pass `--trajectories` and `--topology` more than once. Then, we pass `--atoms` to indicate which atoms should be taken. In this example, we will take just the alpha carbons.

```
python /home/username/enspara/enspara/apps/cluster.py \
  --trajectories wt1.xtc wt2.xtc \
  --topology wt.top \
  --trajectories mut1.xtc mut.xtc \
  --topology mut.top \
  --atoms 'name CA' \
  --algorithm khybrid \
  --cluster-number 1000 \
  --distances /path/to/output/distances.h5 \
  --center-features /path/to/output/centers.pickle \
  --assignments /path/to/output/assignments.h5
```

5.1.3 Feature Clustering

Enspara can also operate on inputs that are “features” rather than coordinates. For example, we have published work that uses clusters based on the solvent accessibility of each sidechain, rather than their position. In that featurization each frame is represented by a one-dimensional vector, and the distances between vectors is computed using some distance function, often the euclidean or manhattan distance (both of which have fast implementations in :code‘enspara‘).

In this case, your `cluster.py` invocation will look something like:

```
python /home/username/enspara/enspara/apps/cluster.py \
  --features features.h5 \
  --algorithm khybrid \
  --cluster-radius 1.0 \
  --cluster-distance euclidean \
  --distances /path/to/output/distances.h5 \
  --centers /path/to/output/centers.pickle \
  --assignments /path/to/output/assignments.h5
```

Here, clusters will be generated until the maximum distance of any frame to its cluster center is 1.0 using a Euclidean distance (the `--cluster-number` flag is also accepted). You can also specify a list of npy files

5.1.4 Subsampling and Reassignment

Sometimes, it is useful not to load every frame of your trajectories. This can be necessary for large datasets, where the data exceeds the memory capacity of the computer(s) being used for clustering, and often does not substantially diminish the quality of the clustering. As a general rule of thumb, it is usually safe to subsample such that frames are 1 ns apart. Thus, if frames have been saved every 10 ps, subsampling by a factor 100 is usually safe. This can be achieved with the `--subsample` flag.

```
python /home/username/enspara/enspara/apps/cluster.py \
  --trajectories /path/to/input/trj1.xtc /path/to/input/trj2.xtc \
  --topology /path/to/input/topology.top \
  --algorithm khybrid \
  --subsample 10 \
  --cluster-number 1000 \
  --distances /path/to/output/distances.h5 \
  --center-features /path/to/output/centers.pickle \
  --assignments /path/to/output/assignments.h5
```

However, when clustering is produced with a subset of the data, it is still valuable to use all frames to build a Markov state model, because it improves the statistics in the transition counts matrix. Consequently, even when clustering uses some subset of frames, it is useful to assign every frame in the dataset to a cluster. This process is called “reassignment”.

By default, reassignment automatically occurs after clustering (it can be suppressed with `--no-reassign`). It sequentially loads subsets of the input data (the size of the subset depends on the size of main memory) and uses the cluster centers to determine cluster membership before purging the subset from memory and loading the next.

Notably, reassignment is embarrassingly parallel, whereas clustering is fundamentally less scalable. As a result, one can provide the `--no-reassign` flag to suppress this behavior and use the centers in some other script to do the reassignment (see the `reassign.py` app).

5.2 Clustering Object

Rather than relying on a pre-built script to cluster data, there is also a scikit-learn-like object for the two major clustering algorithms we use, k-hybrid and k-centers. They are `enspara.cluster.hybrid.KHybrid` and `enspara.cluster.kcenters.KCenters`, respectively.

An example of a script that clusters data using this object is:

```
import mdtraj as md

from enspara.cluster import KHybrid
from enspara.util.load import load_as_concatenated

top = md.load('path/to/trj_or_topology').top

# loads a giant trajectory in parallel into a single numpy array.
lengths, xyz = load_as_concatenated(
    ['path/to/trj1', 'path/to/trj2', ...],
    top=top,
    processes=8)

# configure a KHybrid (KCenters + KMedoids) clustering object
# to use rmsd and stop creating new clusters when the maximum
# RMSD gets to 2.5A.
clustering = KHybrid(
    metric=md.rmsd,
    dist_cutoff=0.25)

# md.rmsd requires an md.Trajectory object, so wrap `xyz` in
# the topology.
clustering.fit(md.Trajectory(xyz=xyz, topology=top))

# the distances between each frame in `xyz` and the nearest cluster center
print(clustering.distances_)

# the cluster id for each frame in `xyz`
print(clustering.labels_)

# a list of the `xyz` frame index for each cluster center
print(clustering.center_indices_)
```

5.3 Clustering Functions

Finally, for the finest-grained control over the clustering process, we implement functions that execute the clustering algorithm over given data, often with very detailed control over stopping conditions and calculations. They are `enspara.cluster.hybrid.hybrid` and `enspara.cluster.kcenters.kcenters`, respectively.

This group of handy recipes might be helpful if you're looking to do something pretty specific and pretty common.

6.1 Building an MSM

Using the object-level interface

```
from enspara.msm import MSM, builders

# build the MSM fitter with a lag time of 100 (frames) and
# using the transpose method
msm = MSM(lag_time=100, method=builders.transpose)

# fit the MSM to your assignments (a numpy ndarray or ragged array)
msm.fit(assignments)

print(msm.tcounts_)
print(msm.tprobs_)
print(msm.eq_probs_)
```

Using the function-level interface

```
from enspara.msm import builders
from enspara.msm.transition_matrices import assigns_to_counts, TrimMapping, \
    eq_probs, trim_disconnected

lag_time = 100

tcounts = assigns_to_counts(assigns, lag_time=lag_time)

#if you want to trim states without counts in both directions:
mapping, tcounts = trim_disconnected(tcounts)
```

```
tprobs = builders.transpose(tcounts)
eq_probs_ = eq_probs(tprobs)
```

6.2 Coarse-graining with BACE

Danger: Be warned that our BACE implementation is still experimental, and you should be careful to check your output.

BACE is an algorithm for converting a large, fine-grained Markov state model into a smaller, coarser-grained model.

```
from enspara import array as ra
from enspara import msm

assigs = ra.load('path/to/assignments.h5')

m = msm.MSM(lag_time=20, method=msm.builders.transpose)
m.fit(assigs)

bayes_factors, labels = msm.bace.bace(m.tcounts_, n_macrostates=2, n_procs=8)
```

This code will create two dictionaries, `bayes_factors`, which contains a mapping from number of microstates (up to `n_microstates` as specified in the call to `bace()`) to a the Bayes' factor for the model with that number of microstates, and `labels`, a mapping from number of microstates to a labeling of the initial microstates of `m` into a that number of microstates.

6.3 Changing logging

Enspara uses python's logging module. Each file has its own logger, which are usually set to output files with the module name (e.g. `enspara.cluster.khybrid`).

They can be made louder or quieter on a per-file level by accessing the logger and running `logger.setLevel()`. So the following code sets the log level of `util.load` to `DEBUG`.

```
import logging

logging.getLogger('enspara.util.load').setLevel(logging.DEBUG)
```

We've implemented the classic pocket detection algorithm LIGSITE in `enspara`, and it can also be used to detect expositions.

7.1 Finding pockets with LIGSITE

`Enspara` packages an implementation of the classic pocket-detection algorithm LIGSITE. LIGSITE builds a grid over the protein, and searches for concavities in the protein by extending a number of rays from each grid vertex, and then counts what fraction of them interact with protein. Points that are not inside the protein, but with rays that intersect the protein, are considered to be 'pocket vertices'.

```
import mdtraj as md
from enspara import geometry

pdb = md.load('reagents/m182t-a243-exposon-open.pdb')

# run ligsite
pockets_xyz = enspara.geometry.pockets.get_pocket_cells(struct=pdb)

# build a pdb of hydrogen atoms for each grid point so it can be
# examined in a visualization program (e.g. pymol)
import pandas as pd

top_df = pd.DataFrame()
top_df['serial'] = list(range(pockets_xyz.shape[0]))
top_df['name'] = 'PK'
top_df['element'] = 'H'
top_df['resSeq'] = list(range(pockets_xyz.shape[0]))
top_df['resName'] = 'PCK'
top_df['chainID'] = 0

pocket_top = md.Topology.from_dataframe(top_df, np.array([]))
```

```
pocket_trj = md.Trajectory(xyz=pockets_xyz, topology=pocket_top)
pocket_trj.save('./pockets.pdb')
```

Transition Path Theory

Transition path theory is a mathematical framework to analyze MSMs to find transition pathways.

8.1 Computing Mean First Passage Times (MFPTs)

Enspara implements fast, raw-matrix transition path theory (i.e. there is no dependence on any enspara-specific objects) for use in extracting various parameters derived in TPT. Among these parameters are Mean First Passage Times, which represent the mean time required to reach a particular state from some other specific state.

First, you'll need a transition probability matrix and (optionally) equilibrium probabilities. For this recipe, we'll use the enspara *MSM* class, but any transition probability matrix and equilibrium probability distribution (as a numpy array) works.

```
import msmbuilder.tpt

from enspara import tpt
from enspara.msm import MSM, builders

msm = MSM(lag_time=10, method=builders.transpose)
msm.fit(a)

# mfpts is an array where mfpts[i, j] gives you the mfpt from i to j
mfpts = tpt.mfpts(tprob=msm.tprobs_, populations=msm.eq_probs_)
```

8.2 Extracting maximum flux pathways

You can also extract a maximum flux pathway. First need a transition probability matrix and (optionally) equilibrium probabilities.

```
# assuming we've fit an MSM, as above in the MFPT example
source_state = 1
sink_state = 100

# compute the net flux matrix from our msm
nfm = tpt.net_fluxes(
    msm.tprobs_,
    source_state, sink_state,
    populations=msm.eq_probs_)

path, flux = msmbuilder.tpt.top_path(maximizer_ind, minimizer_ind, nfm.todense())
```

CHAPTER 9

API

`enspara.apps`

`enspara.cards`

`enspara.cluster`

`enspara.geometry`

`enspara.info_theory`

`enspara.mpi`

`enspara.msm`

`enspara.tpt`

`enspara.util`

`enspara.exception`

Custom `enspara`-only exceptions.

9.1 enspara.apps

9.1.1 enspara.apps.implicit_timescales

9.1.2 enspara.apps.reassign

9.1.3 enspara.apps.cluster

9.1.4 enspara.apps.util module

9.2 enspara.cards

9.3 enspara.cluster

9.3.1 enspara.cluster.hybrid module

9.3.2 enspara.cluster.kcenters module

9.3.3 enspara.cluster.kmedoids module

9.3.4 enspara.cluster.util module

9.4 enspara.geometry

9.4.1 enspara.geometry.libdist module

9.4.2 enspara.geometry.pockets module

9.4.3 enspara.geometry.rotamer module

9.5 enspara.info_theory

9.5.1 enspara.info_theory.entropy module

9.5.2 enspara.info_theory.libinfo module

9.5.3 enspara.info_theory.mutual_info module

9.6 enspara.mpi

9.6.1 enspara.mpi.io module

9.6.2 enspara.mpi.ops module

9.7 enspara.msm

9.7.1 enspara.msm.bace module

Danger: Our BACE implementation is still experimental. Use only if you are an expert and know what you are doing.

9.7.2 `enspara.msm.bootstrap` module

9.7.3 `enspara.msm.builders` module

9.7.4 `enspara.msm.libmsm` module

9.7.5 `enspara.msm.msm` module

9.7.6 `enspara.msm.synthetic_data` module

9.7.7 `enspara.msm.timescales` module

9.7.8 `enspara.msm.transition_matrices` module

9.8 `enspara.tpt`

9.8.1 `enspara.tpt.core` module

9.8.2 `enspara.tpt.tpt` module

9.9 `enspara.util`

9.9.1 `enspara.util.array` module

9.9.2 `enspara.util.log` module

9.9.3 `enspara.util.load` module

9.9.4 `enspara.util.parallel` module

9.9.5 `enspara.util.preprocessing` module

9.10 `enspara.exception`

Custom `enspara`-only exceptions.

exception `enspara.exception.ConvergenceWarning`

Bases: `UserWarning`

An iterative procedure has failed to converge after the maximum allowed number of iterations.

exception `enspara.exception.DataInvalid`

Bases: `Exception`

The data looks structurally invalid (mismatched array lengths, negative numbers where natural numbers are expected, etc).

exception `enspara.exception.ImproperlyConfigured`

Bases: `Exception`

The given configuration is incomplete or otherwise not usable.

exception `enspara.exception.InsufficientResourceError`

Bases: `Exception`

The data is structurally valid, but insufficient computational resources were available to complete the operation or request.

exception `enspara.exception.PerformanceWarning`

Bases: `UserWarning`

Something has happened that may have substantial performance implications and may be easy to avoid.

exception `enspara.exception.SuspiciousDataWarning`

Bases: `UserWarning`

The data is usable, but it has a structure or type that is suspicious, and may cause bad behavior down the road.

CHAPTER 10

Indices and tables

- `genindex`
- `modindex`
- `search`

e

`enspara.exception`, 30

C

ConvergenceWarning, 30

D

DataInvalid, 30

E

enspara.exception (module), 30

I

ImproperlyConfigured, 30

InsufficientResourceError, 31

P

PerformanceWarning, 31

S

SuspiciousDataWarning, 31