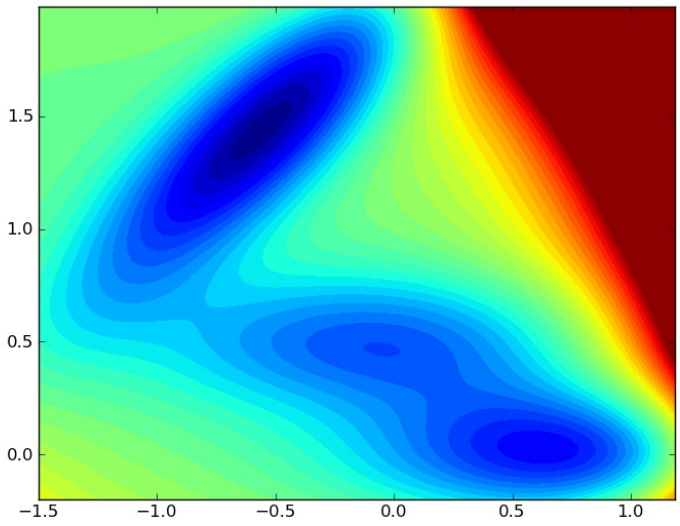
MullerMSM : Markov State Models on the Muller potential -- a MSMBuilder Tutorial



Overview

This package allows you to sample trajectories on the Muller-Brown potential and then use MSMBuilder to build Markov state models to describe the dynamics.

The intent is to provide a simple teaching tool illustrating the process of building Markov state models in a simple and intuitive environment.

Installation

Note: MullerMSM requires MSMBuilder2.5.1 or later, which can be downloaded from http://simtk.org/home/msmbuilder.

Easy way

The easy way to install this package is using `pip`, the python package manager (http://pypi.python.org/pypi/pip/).

```
$ pip install git+git://github.com/rmcgibbo/mullermsm.git
```

Slightly harder way

The package is hosted at https://github.com/rmcgibbo/mullermsm. You can download the package either with the `git` version control system:

```
$ git clone git://github.com/rmcgibbo/mullermsm.git
```

Or by downloading a zip file from `http://github.com/rmcgibbo/mullermsm/zipball/master`

MullerMSM also requires the python package `theano`, which can be installed with

```
$ pip theano
```

To install the MullerMSM package, run

```
$ python setup.py install
```

Using MullerMSM

The MullerMSM package installs three scripts

```
mullermsm_propagate.py
mullermsm_plot_trajectories.py
mullermsm_plot_assignments.py
mullermsm voronoi.py
```

The first script, `mullermsm_propagate.py`, propagates trajectories on the Muller potential energy surface using a simple Langevin integrator, and saves the results in MSMBuilder's format.

The second script, `mullermsm_plot_trajectories.py`, plots the trajectories.

The third script, `mullermsm_plot_assignments.py`, plots the trajectories with where each point is colored based on its micro or macrostate membership, and can help to visualize a macrostate MSM.

The fourth script, `mullermsm_voronoi.py`, helps to visualize the microstates and macrostates of your MSMs.

(1) Simulating Trajectories on the Muller Potential Energy Surface

First, make a new directory to do your work in, and cd there

```
$ mkdir ~/muller
$ cd ~/muller
```

To generate trajectories on the muller potential, use the script `mullermsm_propagate.py`. To simulate ten trajectories of a length 10,000 steps, use the following command.

```
$ mullermsm propagate.py -n 10 -t 10000
```

This script will save the trajectories in MSMBuilder's format, and will create a `ProjectInfo.h5` file for MSMBuilder. You can view the trajectories with `mullermsm_plot_trajectories.py`.

To show all of the trajectories, use the command.

```
$ mullermsm_plot_trajectories.py
```

(2) Clustering the trajectories into a microstate MSM

Now that we've generated the trajectories, we'll use the standard MSMBuilder pipeline to cluster the trajectories into microstates. The only complication is that we need to instruct MSMBuilder to use the right distance metric for clustering. Instead of RMSD, which might be the appropriate distance metric for three-dimensional proteins, we simply want to use the two-dimensional euclidean distance. We do that by suppling a "custom" distance metric.

MSMBuilder provides a number of different clustering algorithms, which each take a variety of different parameters. For this tutorial, we'll use the simplest algorithm, called "kcenters", and tell it to produce 100 microstates.

To do this, call the following script.

```
$ Cluster.py custom -i metric.pickl kcenters -k 100
```

(3) Viewing our trajectories and microstates

The MullerMSM package provides two scripts to help you view your trajectories and your microstates.

To view all of your trajectories plotted in 2D, execute the command

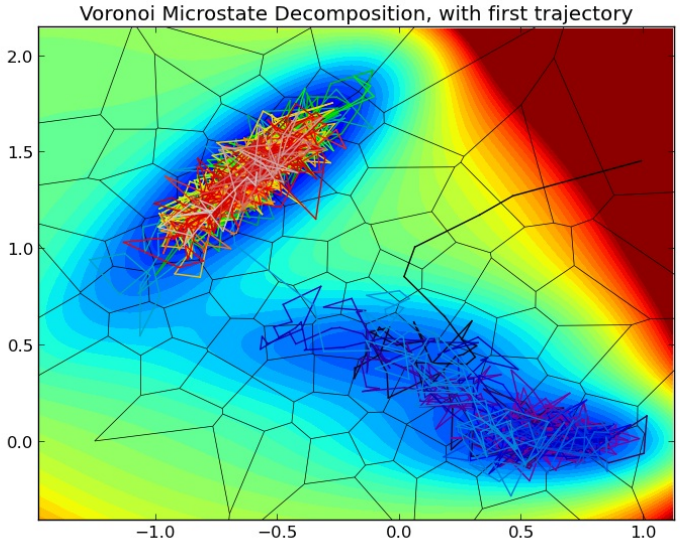
```
$ mullermsm_plot_trajectories.py
```

You should see a plot like this. Each point represents the coordinates of one frame of your simulation, and

the color background shows the underlying potential energy surface.

To view your microstates, execute the command. This shows the regions on the potential energy surface which correspond to each microstate, with the first trajectory plotted on top. The color scheme for the trajectory indicates the progression of time.

```
$ mullermsm voronoi.py
```



(4) Choosing a lag time by implied timescale analysis

MSMs model the dynamics of your system as a memoryless process at a discrete time interval. This interval is called the lag time, and choosing the appropriate lag time key parameter choice in constructing a Markov Model. The choice of lag time presents something known as a bias-variance tradeoff in statistics. Picking a short lag time can lead to biased estimators of the properties of the system, because it takes a nonzero amount of time for the system to "forget" its passed history, a property that is assumed by a markov model. However, picking a long lag time unnecessarily discards data, and makes the statistical estimates of the model properties noisy. A long lagtime also produces a model with less microscopic insight.

In general, we'd like to pick the shortest lag time we can without introducing an unnecessarily large bias into the model. We generally do this by calculating the models "implied relaxation timescales" at a variety of lagtime, and looking for the point at which these timescales begin to converge with respect to lag time.

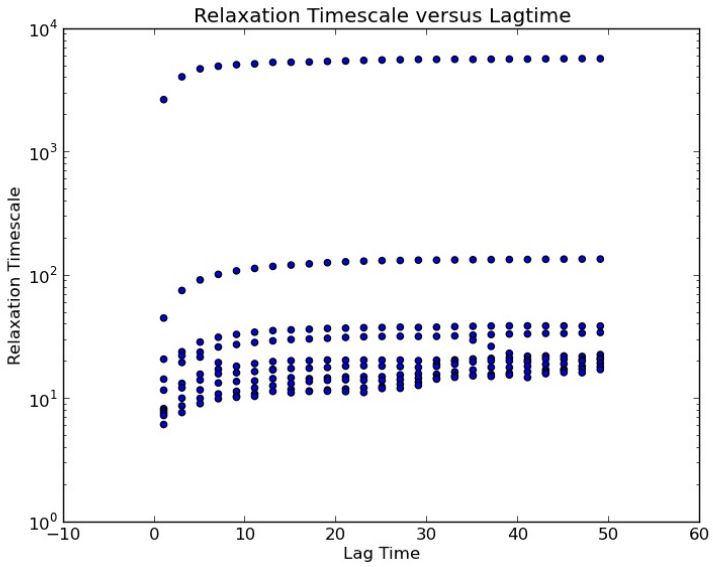
To calculate the implied timescales at lag times from 1 to 50 (with an interval of 2), use the command:

```
$ CalculateImpliedTimescales.py -l 1,50 -i 2
```

To visualize the implied timescales, use the command

```
$ PlotImpliedTimescales.py
```

This analysis is a standard part of MSMBuilder and not specific to MullerMSM.



Visual inspection of this plot reveals that the model begins to converge at a lag time of roughly 5-10 units.

(5) Building a macrostate model to capture the main dynamical processes

The implied timescale plot shows two dominant relaxation timescales, which suggests that a three state macrostate model may be able to capture the essential dynamics.

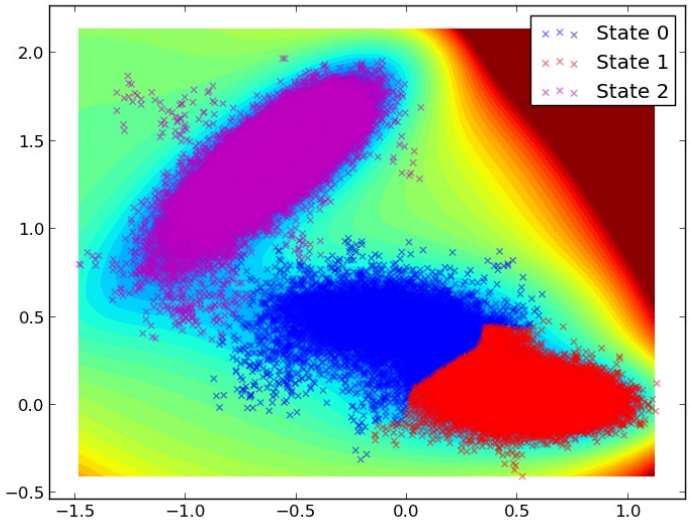
To course grain our microstate model into a three model, we'll first build a full microstate MSM at a lag time of 5, and then coarse grain that microstate model using the PCCA+ algorithm.

```
$ BuildMSM.py -l 5
$ PCCA.py -n 3 -o Macro3 -A PCCA+
```

MullerMSM provides a script that we can use to plot this macrostate model, by plotting all the points from our trajectories with their color indicating which macrostate they are a member of. To use this script, execute the following command

```
$ mullermsm_plot_assignments.py -a Macro3/MacroAssignments.h5
```

You should see plot similar to the one below, showing that MSMBuilder identified the three dominant free energy basins.



(6) Implied timescale analysis for the macrostate MSM

To see how well the macrostate MSM can quantitatively reproduce the data, lets look at the implied

timescales of macrostate model.

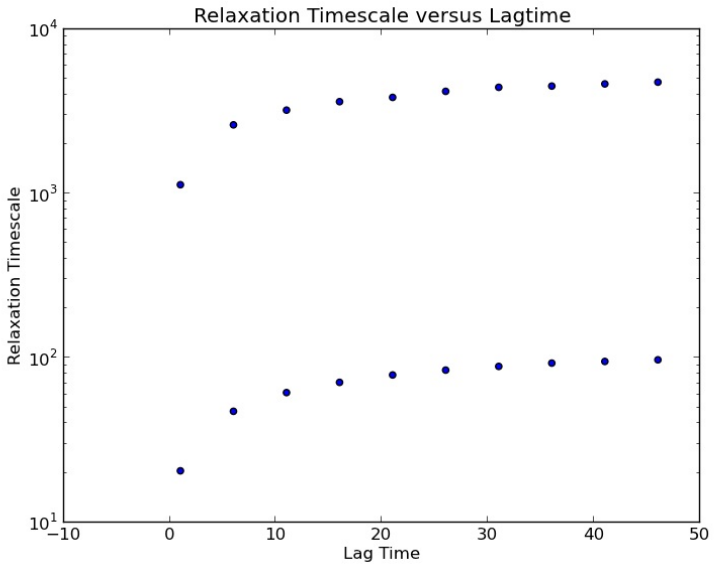
The following command will calculate the implied timescales for your three state MSM.

```
$ CalculateImpliedTimescales.py -l 1,50 -i 5 \
   -o Macro3/ImpliedTimescales.dat -e 2 -a Macro3/MacroAssignments.h5
```

You can plot these timescales with the command

```
$ PlotImpliedTimescales.py -i Macro3/ImpliedTimescales.dat
```

You'll notice that we see only two curves. This is because the three state model can only capture at most two dynamical processes, as the within-state dynamics are not captured at the macrostate level. Nonetheless, the two slowest dynamical processes are faithfully reproduced using this model.



(7) Exercises

- The macrostate model implied timescales converge slower than those of the microstate model. Why do you think this is?
- What would you expect to see from a two-state macrostate model? Try it out!