TUTORIAL: TOOLS FOR STRING METHOD CALCULATIONS.

INTRODUCTION.

In this document we are interested in describing several steps and tools to solve one single problem: find the Minimal Free Energy Path (MFEP) between two states in a system defined by selected collective variables. Specifically, we are dealing with a version of this problem that tries to find the best optimal path followed by a ligand when binding its receptor. And the tool we will use is the On The Fly (OTF) string method [ref].

In general, to find the MFEP is better to start the exploration from a path that is as close as possible to the MFEP. In our case, we started this project with a system in which we were fortunate enough to have initial data to make an informed estimation of the initial path as we had unbiased molecular dynamics (MD) simulations featuring spontaneous binding events between the ligand and the receptor. Therefore, this MD data provided us the justification to divide the work in two main parts:

1. use the MD binding data to make an informed estimation for an initial path through the calculation of the principal curve, and
2. calculate the MFEP from iteration of the principal curve with the OTF string method.

PART I - PRINCIPAL CURVE.

I.a - Building a toy system: di-alanine + DIH.

To ilustrate the use of the principal curve calculation we will build a toy system formed by di-alanine and a small ligand named DIH. We already performed the parametrization files for the ligand and the final system that was also neutralize to charge zero and we added a box of water. The resulting topologies and coordinate files were named system.top and system.coor, respectively.

The goal of this small system is to performed a fast exploratory MD run and extract te center of mass of the ligand DIH. For simplicity the di-alanine was kept fixed as we are interested in exploring the trajectory of the ligand. Specifically, we will extract the center of mass of DIH and we will analyze it with the calculation of the principal curve.

Again for simplicity we provide namd input files for this system in case you want to run similar simulations. Specifically the file *miniHeat.namd* perform minimazion+heating and the file *MD.namd* performs molecular dynamics. To keep the same frame of reference for the OTF string method calculations later we should fix (or restrain) the spatial motions of the receptor, that should not drift away or rotate during the simulation.

I.b - Getting the coordinates of the center of mass of the ligand DIH.

Next we proceed to extract the coordinates of the center of mass of the ligand DIH from the MD trajectory. Here we provide the generic tcl script getfromdcd.tcl already prepared to extract this information from the trajectory *md1.coor.dcd* . Simply load vmd:

```
$ vmd -dispdev none -pdb system.pdb
```

and from the vmd console type:

```
vmd> source getfromdcd.tcl
vmd> run
vmd> exit
```

now you can inspect the file COM_DIH.txt containing the coordinates of the center of mass for the ligand DIH.

```
$ vim COM_DIH.txt
16.878963470458984 17.74061393737793 14.654891967773438
16.563583374023438 17.448957443237305 14.392303466796875
...
```

I.c - Calculating the principal curve.

For the calculation of the principal curve that pases through the points from the file COM_DIH.txt we can now use the tool *pCurve.py*. For the interested reader we tailored an ipython notebook with details of our implementation. For practical purposes is for now enough to say that before calculating the principal curve, *pCurve.py* automatically reads from the file the number of rows (corresponding to time stamps during the MD simulation) and the number of columns (corresponding to the number to collective variables on arbitrary dimension on which we want to calculate the principal curve). The use of the tool is in general:

```
$ ./pCurve.py INPUT Min Mout smooth Niter ...+ploting options
```

here there is a screen shot of the help information for this tool accessed with:

```
$ ./pCurve.py -h
```

```
usage: pcurve.py [-h] [-CV CV2PLOT] [-plotdata VAL]
                 INPUT Min Mout smooth Niter

Calculates the principal curve (pcurve) of a group of 'M' collective variables
(columns) measured 'N' times (e.g., extracted from a MD trajectory, rows). The
method evolves an initial guessed string of 'Min' points to find the pcurve.
At each new iteration, the new points on the string are estimated as the mean
value of the DIM-dimensional voronoi cells. The new points are subsequently
reparametrized to a string with equidistant points which is also smoothed
before the new iteration. The final converged string (or pcurve) can be
optionally reparaterized to a higher number of points 'Mout'. Several plots
are produced including the convergence of the iterations and an estimated free
energy along the pcurve. The final state of the calculations is saved to the
file 'pcurve_state.dat', which serves as input to other tools (e.g.,,
map2curve.py).

positional arguments:
  INPUT          Input file
  Min            Number of points on the path (or string) located on the free
                 energy landscape and joining the initial anf final states of
                 the system.
  Mout           The points in Min can be reparametrize to Mout points after
                 convergence
  smooth         A factor between [0,1] defining the strength of the smoothing
                 performed after each reparametrization.
  Niter          Number of iterations updating the strings.

optional arguments:
  -h, --help     show this help message and exit
  -CV CV2PLOT    Indexes between [0, (DIM-1)] of the CVs to be visualized (2D
                 and 3D options are possible).
  -plotdata VAL  Default is 'yes'. If the value 'no' is specified then the
                 intermediate plots will be produced with the curves only.

Thanks for using pcurve.py!!
```

Although the user can use this tool to calculate the principal curve in a N-dimensional space, only 2D and 3D ploting options are available and for this to work properly the user must additionally indicate which collective variables (columns in the input file starting with index 0) are going to be visualize. For instance, the plotting options "-CV 0 -CV4 -plotdata yes" will produce 2D plots using columns 0 and 4; options "-CV 2 -CV4 -CV50 -plotdata yes" will produce 3D plots using columns 2, 4, and 50. Next we will show some results obtained with the provided input file and the following comand line:

```
$ ./pCurve.py COM_DIH.txt 5 10 0.8 20 -CV 0 -CV 1 -CV 2 -plotdata
yes
```

I.d - Checking the generated principal curves.

The results of a principal curve calculation are saved at the end of the run in as serialized python objects in a file named *pcurve_state.dat*. Serialization using the python module pickle is just a convenient way to saved python objects that can be re-imported again in a python program; in fact if the user want to access to a summary of this data again, save ascii versions of the data, or replot the results present in the file pcurve_state.dat we provide an additional tool named *check_pcurve.py*,

```
$ ./check_pcurve.py -h
```

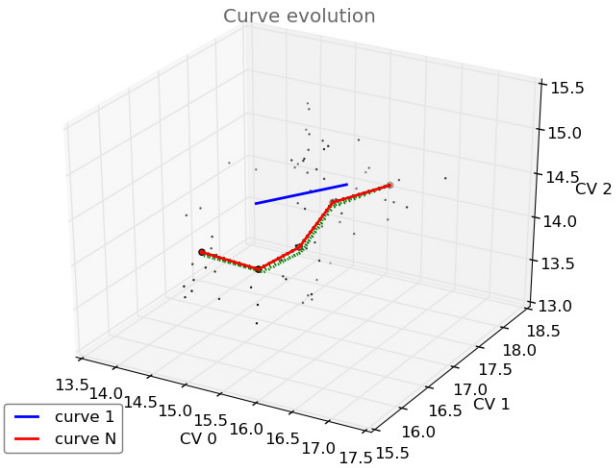


Figure 1: This is the first plot produced by *pCurve.py*. Here the user can have an idea of the evolution of the iterations as we represent the starting curve (blue) the subsequent iterations (green) the the final curve (red). The convergence of the procedure can also be explored in an additional plot (not shown).

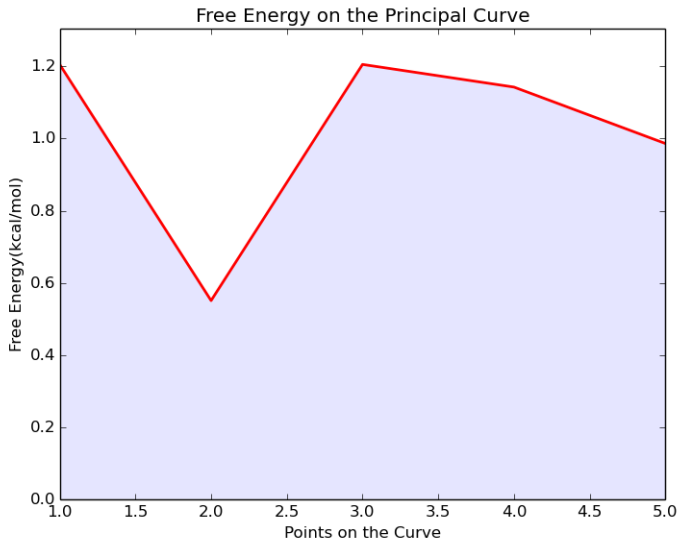


Figure 2: Another output visualization of *pCurve.py* is the free energy entropy along the curve.

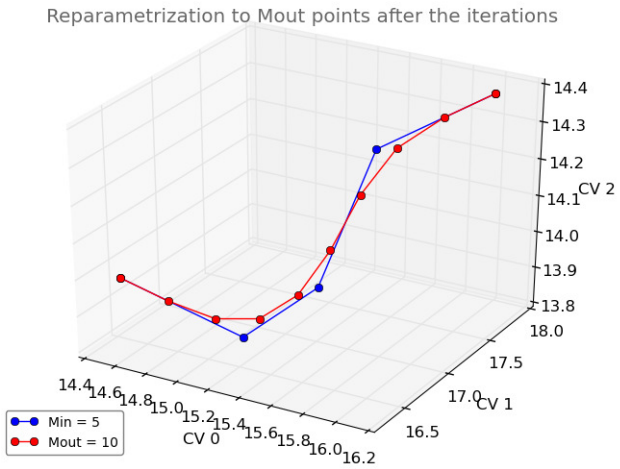


Figure 3: From *pCurve.py* we also obtained the reparametrization to Mout points (plus smoothing) of the obtained converged principal curve.

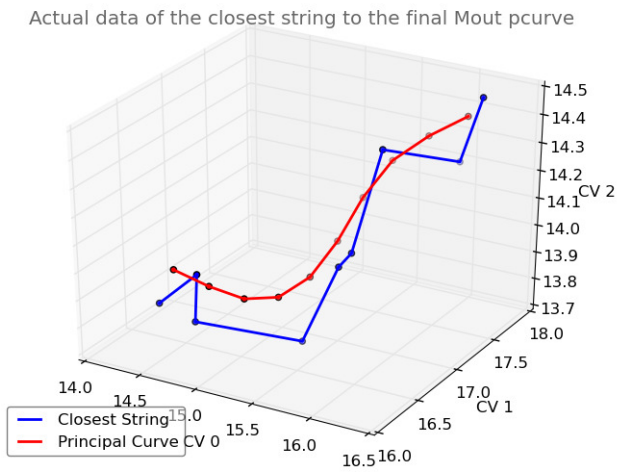


Figure 4: As the points that form the principal curve do not in general belong to the initial dataset present in the input file, it can sometimes be useful to have a 'pseudo'-principal curve formed by the points of the input data that are closest to the calculated principal curve.

```
usage: check_pcurve.py [-h] [-plot PL] [-save SV] INPUT

This tool displays (and/or saves) the content of a pcurve state file (output
from pcurve.py).

positional arguments:
  INPUT       Input pcurve_state file.

optional arguments:
  -h, --help  show this help message and exit
  -plot PL    Plot data? [yes/no].
  -save SV    Save curves? [yes/no].

Thanks for using check_pcurve.py!!
```

Then we cal use the following comand line to first, visualize a summary of the parameters that were used to build the principal curve saved in the file *pcurve_state.dat* (see bellow), and second, see some plots related to this curve;

```
#####################################################
############# pcurve run parameters  ################
#####################################################

N      (Size of the CV vectors)            =  68
DIM    (# of CVs)                          =  3
Min    (#of points on the string)          =  5
Mout   (# of final reparametrized points)  =  10
smooth (strength of the smoothing [0-1])   =  0.5
Niter  (# of iterations)                   =  30

#####################################################
we are ploting...
```

Additionally, if we set the 'save' input argument of *check_pcurve.py* to 'yes', then two additional types of data will be produced:

1. Three files with the 'Mout' coordinates of the center of mass from the reparametrized principal curve , and
2. one file named *closeII.dat* with the indices of the rows in the input file that are closest to the saved principal curve.

With this information we can prepare the conditions to run the OTF string method calculations (see **PART II**)

I.e - Troubleshooting principal curves: the case of clustered data.

Occasionally when we are trying to calculate the principal curve of a dataset, clustered data can introduce problems in the algorithm and in these cases it can happen that the principal curve crosses the space between two clusters in which we really did not observed any dataset point. If this result is not the desired one, we can alternatively deal with this situation by

1. making an unsupervised identification of the clusters and,
2. use extra domain information from the data (e.g., time stamps) for decision making with the final goal of creating a global principal curve from local principal curves calculated using selected data clusters. To help in these cases we provide a few additional tools, but first we will generate test data for this section using the the tool *fakeData.py*

To help in these cases we provide a few additional tools, but first we will generate test data for this section using the the tool *fakeData.py*

```
$ ./fakeData.py -h
```

```
usage: fakeData.py [-h] N M OUTPUT

This tool creates N randomly distributed gaussian clouds of data. Each cloud
has M points. The final collection of clouds is saved in the output file using
3 columns. This data generator is used in the tutorial to produce fake data
for cluster analysis or to estimate the correct number of clusters using
nclust.py.

positional arguments:
  N           Number of clusters.
  M           Number of points in each cluster.
  OUTPUT      Output filename prefix

optional arguments:
  -h, --help  show this help message and exit

Thanks for using fakeData.py!!
```

With this tool we can easily generate a 3D testing dataset with 6 clusters and 400 points per cluster and save the 3 columns in a file *rawclusters6* by the command,

```
$ ./fakeData.py 6 400 rawclusters
```

This tool will also produce a plot of the data set, but if we want to visualize the dataset and also find the 6 centers of each cluster we can use the tool *showCenters.py* that implements a simplified version of k-means and plot the original dataset (rawclusters6) and the 6 associated centers.

```
$ ./showCenters.py -h
```

```
usage: showCenters.py [-h] [-CV CV2PLOT] [-plotdata VAL] INPUT K Niter

Plots the cluster centers and the data. The number of cluster center K should
be previously calculated using ncluster.py). The data should also correspond
to the data used with ncluster.py: a group of 'DIM' collective variables (CVs)
measured 'N' times (e.g., extracted from a MD trajectory). The method, evolves
an initial guess formed by K points to find the K center of the clusters. At
each new iteration, the new points are estimated as the mean value of the DIM-
dimensional voronoi cells. This is equivalent to a k-means cluster
calculation.

positional arguments:
  INPUT           Input file
  K               Number of clusters.
  Niter           Number of iterations updating the strings.

optional arguments:
  -h, --help      show this help message and exit
  -CV CV2PLOT     Indexes between [0, (DIM-1)] of the CVs to be visualized (2D
                  and 3D options are possible).
  -plotdata VAL   Default is 'yes'. If the value 'no' is specified then the
                  intermediate plots will be produced with the curves only.

Thanks for using showCenters.py!!
```

Accordingly the command,

```
$ ./showCenters.py rawClusters6 6 30 -CV 0 -CV 1 -CV 2 -plotdata yes
```

will use 30 iterations of the k-means algorithm to find the centers of 6 clusters in the file *rawclusters6*. A plot of the clusters can be seen in the next figure.

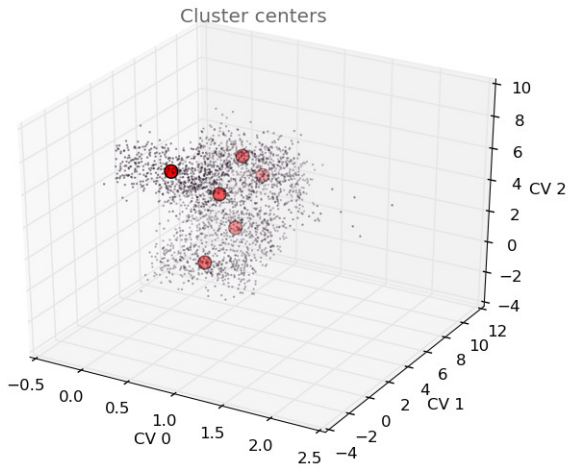


Figure 5: K-means located the positions of the 6 clusters centers (red) from the input file rawclusters6 using the tool *showCenters.py*

Additionally this tool will also produce

1. a file named *centers.dat* containing the coordinates of the centers found by the k-means algorithm,
2. files with coordinates with points from the original dataset; these files will be named group[i] $(1 < i < q)$ where q is the number of possible combinations between two clusters. For instance, file *group1* will have points from clusters with indices 0 and 1. For combination of 2 from 6 different clusters we will therefore obtain 15 groups. We can subsequently calculate the principal curve between any combination of clusters to create a global path.

The scenario described so far in this section uses generated data in which we already know by construction the correct number of clusters. In the general case, however this number has to be determined and for this task we provided the tool *nCluster.py* that implements a distortion measure. For the interested reader we tailored an ipython notebook explaining and showcasing the details of our implementation. The help file of the tool is

```
$ ./nCluster.py -h
```

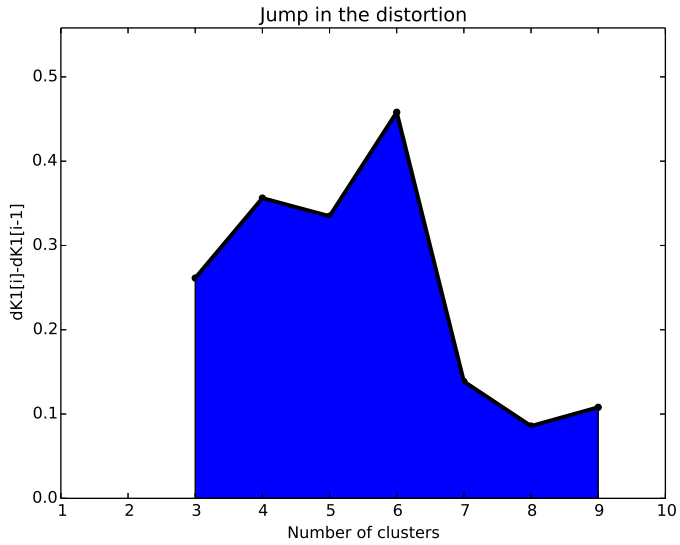


Figure 6: Jump in the distortion to calculate the correct number of clusters using the tool *nCluster.py*; note that the largest jump is observed for a number of clusters equal to 6, which we know that is correct in this example by construction.

```
usage: nCluster.py [-h] [-CV CV2PLOT] [-plotdata VAL] INPUT maxC Niter

Calculates the optimal number of clusters in a dataset formed by a group of
'M' collective variables (columns in the input file) measured 'N' times (rows
in the input file). The procedure evolves an initial guess formed by K cluster
centers. At each new iteration, the new centers are estimated as the mean
value of the M-dimensional voronoi cells (k-means method). Next we calculate
the 'distortion' of the K cluster partition (dKi). We repeat the same
procedure for a range of cluster partitions Ki (2 < Ki < maxC). The highest
'jump' in the cluster partitions (Jk_i = dK_i**(-M/2) - dK_(i-1)**(-M/2))
represents the optimal partition and therefore the correct number of clusters
in the data set.

positional arguments:
  INPUT          Input file
  maxC           Maximal mumber of clusters (2 < Ki < maxC).
  Niter          Number of iterations updating the strings.

optional arguments:
  -h, --help     show this help message and exit
  -CV CV2PLOT    Indexes between [0, (DIM-1)] of the CVs to be visualized (2D
                 and 3D options are possible).
  -plotdata VAL  Default is 'yes'. If the value 'no' is specified then the
                 intermediate plots will be produced with the curves only.

Thanks for using nCluster.py!!
```

Now we can test our tool with the file *rawclusters6* that we know to have 6 cluster by construction. We can therefore use our tool with 30 iteration for each test partition for a maximun of 10 partitions or number of clusters with the following command line,

```
$ ./nCluster.py rawClusters6 10 30 -CV 0 -CV 1 -CV 2 -plotdata yes
```

The last thing we need to explore is the order of the clusters. If the data that we

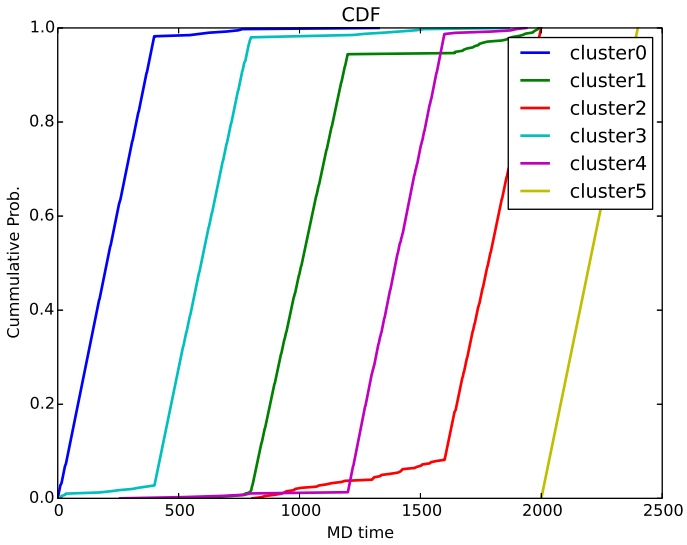
are exploring comes from a molecular dynamics trajectory then we will asume that each row in the data file corresponds to a time stamp; therefore, every cluster will be formed by points that have time stamps. The next reasoning for asign order to cluster is very simple: if probabilistically most of the points in a cluster have a low time stamp then this cluster should be used earlier than others to build local principal curves. As a probabilistic measure we use the the cummulative distribution functions (CDFs) of the points in the cluster taking into account their time stamps. We provide a tool to do this calculation named *clusterCDF.py*,

```
$ ./clusterCDF.py -h
```

```
usage: clusterCDF.py [-h] [-CV CV2PLOT] [-plotdata VAL] INPUT K Niter

This tool find the temporal order of the clusters in a dataset using the
cumulative distributions functions (CDFs) of the points in each cluster. To do
this task, it is assumed that the row index for each data point in the input
file corresponds to a time stamp (i.e., as in a MD simulation). Resuming, from
this calculation we obtain the reordering of the clusters in a sequence
indicated by their time stamps and therefore the obtained ordering should be
maintained when building local principal curves between subsequent pairs of
clusters in time. In this tool we first evolve an initial guess formed by K
points to find the K center of the clusters (using k-means). Next, the
clusters CDFs are calculated and ploted. The temporal location of the CDFs in
the final plot corresponds to the temporal order of the clusters.

positional arguments:
  INPUT          Input file
  K              Number of clusters.
  Niter          Number of iterations updating the strings.

optional arguments:
  -h, --help     show this help message and exit
  -CV CV2PLOT    Indexes between [0, (DIM-1)] of the CVs to be visualized (2D
                 and 3D options are possible).
  -plotdata VAL  Default is 'yes'. If the value 'no' is specified then the
                 intermediate plots will be produced with the curves only.

Thanks for using clustersCDF.py!!
```

Following the example we have been building we can now find the ordering of the clusters in the file *rawclusters6* with this command line,

```
$ ./clusterCDF.py rawClusters6 6 30 -CV 0 -CV 1 -CV 2 -plotdata
yes
```



Calculation of CDFs taking into account the time stamps of the points in each clusters. In this simple case, if a *global* principal curve needs to be built reconstructing temporal cluster information then we should first calculate *local* principal curves between [cluster0-cluster3], [cluster3-cluster1], [cluster1-cluster4], [cluster4-cluster2], and [cluster2-cluster5]. Finally we should put all the *local* curves into a *global* principal curve as desired.

I.f Further analysis on principal curves: density.

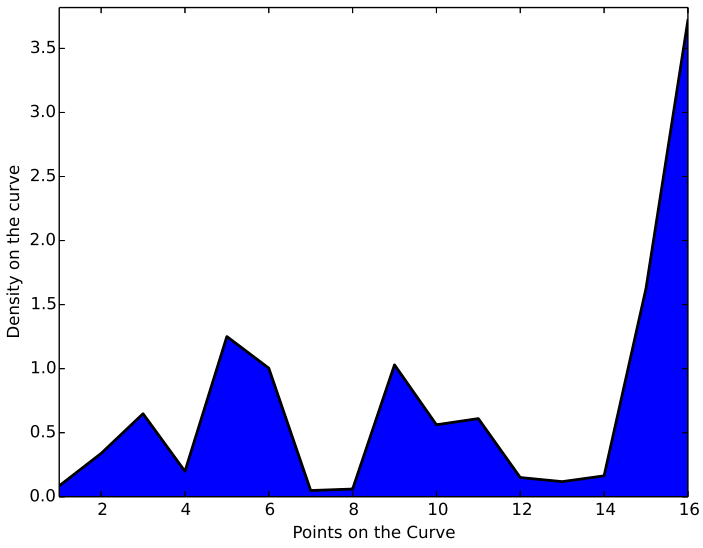
Additionally we also provide a tool to explore the density of data points around each image of the generated principal curve. This tool in fact uses the densities of concentric spheres around each image to extrapolate the value "on the image". the help for this tool is provided bellow:

```
$ pointDensity3D.py -h
```

```
usage: pointDensity3D.py [-h] file1 file2 file3 choice

Estimates the density of points on the principal curve in the collective
variable space (only 3D case, e.g., coordinates of the center of mass). As the
data to estimate the density on the curve is necesarily discrete, here we
extrapolate the density pattern from larger distances until reaching the
points on the principal curve. Shortly, we use an equation like ln(d) =
ln(C)+ln(e(-lr)) = ln(C)-lr, where d=density, r=radius, l=decay constant. We
estimate the slope of the model and extrapolate for the case in which the
distance to the curve is zero, that is, the density 'on the curve'. The
procedure is repeated for each point of the principal curve to render a
density profile that is both, plotted and saved.

positional arguments:
  file1       Input pcurve_state file.
  file2       Original CV file.
  file3       Output file.
  choice      Use Min or Mout? choice '0' for Min and '1' for Mout.

optional arguments:
  -h, --help  show this help message and exit

Thanks for using pointDensity3D.py!!
```

An example output image is shown bellow:

```
$ ./pointDensity3D.py pcurve_state.dat LIGAND_COM_ALL.txt out.out
1
```



Density on a principal curve obtained for the binding of the ligand DIH to the protein PNP through the path01 (gate). As collective variables we used the coordinates of the center of mass of DIH. When executing the command line for this figure, the first output on the computer terminal is a resume of the parameters used to calculate the principal curve (previously saved in the file *pcurve_state.dat*)

I.g Additional analyses on the principal curve: mapping other variables.

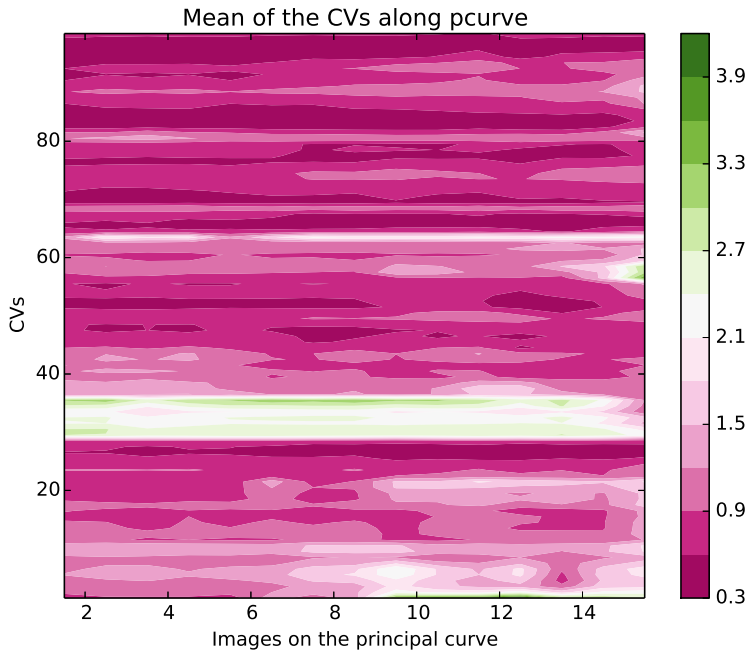
Once we have arrived to convergence to a principal curve, we might also be interested in exploring the behaviour of other variables along the curve. We provide a tool for this purpose called *map2pcurve.py*:

```
$ map2pcurve.py -h
```

```
usage: map2pcurve.py [-h] pcurveFile cvFile iniCV endCV

This tool maps collective variables (CVs) from an input file on a principal
curve. Each CV in the data file should be place in a column. The rows in the
CV data file should coincide in time with the CVs used to estimate the
pricipal curve (as a particular case, the same CV data file can be used with
both pcurve.py and map2pcurve.py). The initial and final indexes of the CVs to
be mapped should be provided. The information of the principal curve is
automatically loaded from the file 'pcurve_state.dat' (output from 'pcurve.py'
). To extract CVs from MD trajectories using VMD see the file getFromDCD.tcl.

positional arguments:
  pcurveFile  Input file with the data from a calculated principal curve.
  cvFile      Input file with CVs.
  iniCV       Index (column) of the first CV to be mapped.
  endCV       Index (column) of the last CV to be mapped.

optional arguments:
  -h, --help  show this help message and exit

Thanks for using map2pcurve.py!!
```

In the next example we use a file containing the rmsd per residue of the backbone atoms of protein PNP extracted from the same MD trajectory used to find the principal curve (*rmsd_backbone_all.dat*). Although in the CV file we measured many collective variables, in this example we will map only the collective variables from columns 1 to 100, additionally only the mean values of the CVs will be plotted, however the tool *map2pcurve.py* produces several similar plots.

```
$ ./map2pcurve.py pcurve_state.dat rmsd_backbone_all.dat 1 100
```



PART II - OTF STRING METHOD CALCULATIONS.

II.a - Preparing files for OTF string method calculations: center of mass as collective variable.

We start this section by giving a brief introduction on how to prepare input files for a string method calculation. For the rest of the section we will assume that we are using as collective variables the coordinates of the center of mass of the ligand as it binds the receptor. The steps that we will follow in this part of the tutorial can be used by the interested reader on the toy system we build on section **I.a**, running a longer MD simulation and using the principal curve obtained at then end of section **I.d**. However, we prefer to show the results of a more interesting system because it will give us the chance to used additional analysis tools. Specifically, the system in this section is the trimeric protein PNP used as a receptor and the same DIH ligand already used in section **I.a**.

To run a calculation with the OTF string method following this tutorial you will need:

1. A file that here we will name *initialPcurve.dat* with the value of the collective variables of our system, in that file each row represent a point along of the principal curve for a total of Mout points. Therefore, *initialPcurve.dat* is a subset of the initial file of coordinates (e.g., *rawclusters6* in the example of **PART I**); specifically, using the rows which indices are in the file *closeII.dat* (see section **I.d**).

2. 'Mout' sets of initial MD files for namd (i.e., coordinates, velocities, topol-

ogy, box size); each set should correspond to a row in the collective variable file *initialPcurve.dat*. Ideally, the collective variables on the coordinate file corresponding to point i of the principal curve should be equal to the value on row i in *initialPcurve.dat*. Each of the Mout systems should be independently energy minimized, heated, and equilibrated taking the additional care of restraining the position of both, the ligand and receptor before running OTF calculations.

3. Indicate the correct parameters to run the OTF string method (see next section).

A few notes should be added to clarify the point 2 above. If we want to start OTF calculations using as initial guess a principal curve obtained from an unbiased MD trajectory, then we can reasonably argue that the unbiased binding trajectory is close to the MFEP and therefore we can count in this case great starting data to find the best path, consequently, we should expect that the OTF calculations converged in a few number of iterations. In this case and provided that we have build the system following section **I.a**, then we can obtained the coordinates for each point on the principal curve by extracting frames from the unbiased MD that we used to find the principal curve. That is, to get the coordinate (and velocity) files for for the OTF calculation we should extract the MD frames with the indices saved in the file *closeII.dat*.

A second case that we will mention now but that will not be followed extensively in this tutorial is related to a situation in which we don't have a MD trajectory to estimate the principal curve. In that case is up to the user to find a reasonable starting curve before reconstructing the system. One possible sequence of stepts could be (a) put the ligand at several distances from the receptor following your chosen initial guess, to each distance will correspond a point in the initial string before starting the OTF calculations, (b) add a water box and the ions to every system controlling that in each case the quantity of water molecules and ions are identical every time, (c) perform energy minimization+heating+equilibration on each system, (d) obtain the value of the relevant collective variable for each system and save them in a file *initialPcurve.dat* (equivalent to item 1 above).

II.b Setting basic run parameters for the OTF string method calculation.

To run calculations with the OTF string method we provide an example folder with the necesary scripts and a README file describing the individual files and directories.

```
$ ls -ltr
```

The folders and files are functional provided that the user place their input files for the system and modify the parameters for the OTF calculation. The main file for the OTF calculation is a bash script that sets general run parameters, creates a framework for the OTF iterations, and calls other programs. Bellow we show the header of the main script named *OTF_string.sh*:

```
#!/bin/bash
#################################################################################################
#####  Initializations  #########################################################################
#################################################################################################
p1=1                                        # first point on the string
pN=3                                        # last point on the string
Nit=1000                                    # No. of iterations
Nsave=10                                    # save coordinates every Nsave iterations
LD_LIBRARY_PATH="/path1/:$LD_LIBRARY_PATH";  # path to namd library (tobe used with CUDA)
export LD_LIBRARY_PATH
namd='/path2/namd2'                         # path to namd executable

#################################################################################
###  comment the following lines if this run is a continuation of a previous one ###
cp INPUT/* OUTPUT/                                                            ###
cp INPUT/initialPcurve.dat string0.dat       # values of the initial string     ###
#################################################################################
```

As we can see from this figure, here we set (a) the path for the namd executable file and associated libraries (needed to run on CUDA capable devices), (b) the initial and final points of the string that we want to iterate, and (c) the total number of iterations. Besides this, if the calculation we are performing does not converges within the selected number of iterations and we need to continue the run, then we should comment a couple of lines in second part of the header in *OTF_string.sh* to avoid overwritting previous results. Additionally we need to increase the total number of iterations (e.g., to 2000 in this case) and to update the starting point in the master loop for iterations inside the script (e.g., to 1000 in this case). Further modifications to the code are not complex; we provide example files for the calculation of the OTF mstring method using 2 centers of mass and for further modifications the user can start by checking this pseudo-code containing the general framework of the method:

```
# Loop on the OTF iterations
for ((i=1; i<=$N; i++))
do
        >Get the CVs from old string (previous iteration)
        # Loop on the points of the string
        for ((j=1; j<=$M; j++))
        do
                >Prepare namd configuration file
                >Get coordinates, velocities, box size from the previous iteration
                >run MD to compute and evolve forces on CVs
                >save final coordinates, velocities and box size
                >save final values of the CVs
        done
        >reparametrize new string
done
```

In the example folder we provide there are a few other files that might need modifications to run OTF canculations. These are:

1. *reparam.f90*. This file contains fortran code for the reparametrization and the default version if set to use e collective variables (e.g., the coordinates of the center of mass). The source code is provided with comments on which parts to change if we need to reparametrize to a larger number of points in the string, or to use a different number of collective variables. Clearly, the code needs to be recompiled after every change (into *reparametrization.x* in our folder for instance).

2. **.namd* files. These are namd configuration files that should work on most systems provided that the input has been properly minimized, heated, and equilibrated. Periodic boundary conditions should be change to match thouse of your system and finaly we should also consider that this file calls (a) a file to use the CV module (*CV_config.in*), and (b) a file to use the force calculation module (*namd_OTF.tcl*).

3. *CV_config.in*. This file is provided to use the CV module on VMD, specifically this file keeps the ligand from moving aroung or rotating, therefore the user should change the indices of the atoms that will be influenced by the CV module and the forces they want to apply to those atoms. *CV_config.in* uses for the reference coordinates the file *fixed.pdb* that should be a desired frame of the system under study.

4. *namd_OTF.tcl*. Script to use the tcl forces module of namd. The main input for this file is the atom numbers on which we will apply the forces on the OTF method. By default, the atoms in this file should match the group on which we want to calculate the center of mass.

II.c OTF calculations: convergence and reparametrization.

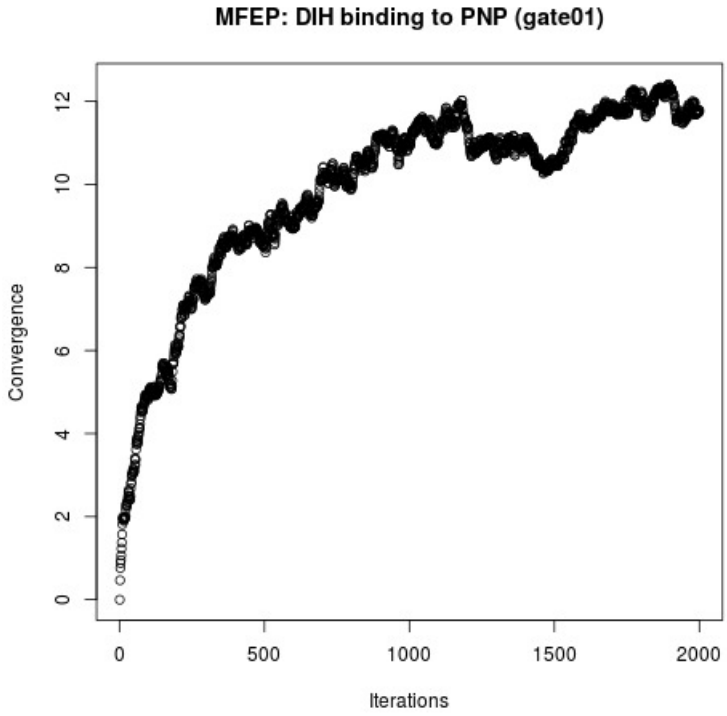
Once we launch an OTF string method calculation we need to monitor it for convergence (i.e., finding the MFEP, at least locally). We provide a couple of tools to do this, the first one is *convergence.py*:

```
./convergence.py -h
```

```
usage: convergence.py [-h] Niter IN_file

Tool to estimate the convergence of the string method for the calculation of
the minimum free energy path. The calculation is based of the succesive
differences between strings in the space of the normalized CVs. In the input
files: (1) each column represents a CV, (2) each row represents a point along
the path (string), and (3) the columns should be separated by a space
character.

positional arguments:
  Niter       number of iterations files updating the strings
  IN_file     Prefix of the files containing the string iterations. Example:
              if you have the files file1 file2, etc., then IN_file='file'

optional arguments:
  -h, --help  show this help message and exit

Thanks for using convergence.py!!
```

This tool uses the the notion of normalized Frechet distance Frechet distance to estimate the differences between succesive strings during the iterations. Besides that it also provides an estimation of the convergence by every point (or image) of the strings during the iterations; in that way the user can see which are the specific points or regions along the string that are delaying the convergence. However, we also provide here simplified R functions to estimate the convergence, the plot we show next explores the convergence of the string along path01 (gate) that the ligand DIH follows while binding to the trimeric protein PNP. There are in this case 2000 input files (OTF iterations using 2 centers of mass, 2COM). Input files were named string1.dat, string2.dat ... string2000.dat

```
$ R
> source("myFunctions.R")
>allData<-pasteDF2COM(2000)
>png(filename="conv.png")
>plot(convDF(allData),xlab="Iterations",ylab="Convergence", main="MFEP:
DIH binding to PNP (gate01)")
>dev.off()
>q()
```



The immediate goal of finding a converged string is to calculate the free energy along the path (i.e., the MFEP). However, we might want to increase the resolution of this path before launching MD simulations to estimate the mean forces. For example, the convergence plot we just presented was calculated using a string with 21 points. To increase this resolution we need to do the following additional steps:

1. reparametrized the curve from 21 points to a higher number, for instance 40.
2. using the closest coordinates from the previous strings to obtain 40 new systems.
3. run again the OTF string method on the new string of 40 points or images.

We provide a tool named *findNewStart.py* to help in this task:

```
$ findNewStart.py -h
```

```
usage: findNewStart.py [-h] ligand prefix Nimages Nit1 Nit2 Mout

This uses coordinate data produced by a OTF string method calculation to find
the mean string from the strings produced between iterations Nit1 and Nit2,
then reparametrizes the found mean string to Mout points and finally finds the
coordinate files that are closest to each point in the Mout reparametrized
string.

positional arguments:
  ligand      name of the ligand residue in the pdb file.
  prefix      name of the prefix from the input pdb files.
  Nimages     Number of images in the string.
  Nit1        Fist iteration to be analyzed.
  Nit2        Last iteration to be analized.
  Mout        Number ot points of the final reparametrized string.

optional arguments:
  -h, --help  show this help message and exit

Thanks for using findNewStart.py!!
```

For instance, a command line that will reparametrize 21 images into 40, by exploring the ligand (DIH) coordinates produced between iterations 1900 to 2000 is:

```
$ ./findNewStart.py DIH namdout 21 1900 2000 40
```

This command line produces an output file named *pdb_indices.dat* that has a header with the information about the input parameters given in the command line and the indices of every file used to build the reparametrized 40 point curve. Additionally, the script will also produce the pdb files of the reparametrized points of the final string (40 points in this case).

PART III - OTF STRING METHOD: ANALISIS TOOLS

III.a OTF calculations: convergence and reparametrization.