ProDy

Protein Dynamics & Sequence Analysis

ClustENMD

Release

Burak Kaynak, Pemra Doruker

December 04, 2024

CHAPTER

ONE

INTRODUCTION

ClustENMD *[KD21]* is a highly efficient, unbiased conformational search algorithm, which is suitable for generating atomistic conformers for ensemble docking purposes and for large, supramolecular assemblies like the ribosome *[KD16]*. In this hybrid method, the following steps are carried out at each generation/cycle: (1) conformer generation by random sampling along global ANM modes, (2) hierarchical clustering of generated conformers and (3) relaxation of cluster representatives by short MD simulations using OpenMM *[E17]*. The relaxed conformers are then fed back to Step 1, each being used as a starting point for a new generation of conformers. This iterative procedure (Steps 1-3) is repeated for several generations to allow for sufficiently large excursions from the initial structure. Thus, conformational sampling can be efficiently performed for highly flexible systems composed of proteins, RNA and/or DNA chains. Furthermore, the generated ensemble can be analyzed and utilized using the available tools in ProDy.

ANMD *[CM22]* is a simple conformers generation algorithm that explores motions along single ANM modes. The normal mode calculation is only performed once and a set of structures along each of the first few modes is subjected to energy minimization, producing physically reasonable conformers near the initial energy well. A key feature of ANMD is that each successive mode is sampled with a lower amplitude related to its frequency.

The first part of this tutorial demonstrates how to use ClustENMD to perform conformational sampling for the homodimeric enzyme HIV-1 protease in an open conformation without any ligand (PDB id: 1tw7). Furthermore, we will show the application of ProDy ensemble analysis tools to study the conformers and generate their population distribution.

The last part demonstrates how to use ANMD to explore the first two modes of a metabotropic glutamate receptor N-terminal domain (PDB id: 1ewk).

1.1 Required Programs

The latest versions of **ProDy_**, OpenMM[1], and PDBFixer[2] are required for ClustENMD.

1.2 Getting Started

We recommend that you will follow this tutorial by typing commands in an IPython session, e.g.:

```
$ ipython
```

or with pylab environment:

```
$ ipython --pylab
```

First, we will make necessary imports from **ProDy_**, **NumPy_**, and **Matplotlib_** packages.

[1] https://openmm.org/
[2] https://github.com/openmm/pdbfixer

```
In [1]: import numpy as np

In [2]: import matplotlib.pyplot as plt

In [3]: from prody import *

In [4]: plt.ion()
```

We have included these imports in every part of the tutorial, so that code copied from the online pages is complete. You do not need to repeat imports in the same Python session.

1.3 How to Cite

If you benefited from ClustENMD in your research, please cite the following paper:

Additionally, please also cite the following paper for OpenMM:

CHAPTER

TWO

CLUSTENMD SIMULATION AND ANALYSIS

First, we will make the following necessary imports **ProDy_**, **NumPy_**, and **Matplotlib_** if you haven't already done it:

```
In [1]: import numpy as np

In [2]: import matplotlib.pyplot as plt

In [3]: from prody import *

In [4]: plt.ion()
```

2.1 Preparing the system and ClustENMD simulation

We start our calculations by parsing the structure, from which we would like to sample conformations. For this tutorial, we will fetch the X-Ray structure of HIV-1 protease in an open conformation in the absence of any inhibitor (PDB id: 1tw7) from the PDB server.

It is important to note that if the starting structure is provided by the user, it should satisfy the PDB file standards, e.g., the chain IDs need to be set properly.

The pdb file (PDB id: 1tw7) is fetched by the method `parsePDB()`. Please check the ProDy Basics tutorials[3] for the details.

```
In [5]: pdb = parsePDB('1tw7', compressed=False)
```

```
@> PDB file is found in working directory (1tw7.pdb).
@> 1890 atoms and 1 coordinate set(s) were parsed in 0.03s.
```

ClustENMD is implemented as a ProDy class, named as `ClustENM`, so we can instantiate an object of it. You can provide a title, but it is optional.

```
In [6]: clustenm = ClustENM()
```

Before running a simulation, we need to set the atom group that we would like to use. This method uses PDBFixer to add all hydrogen atoms as well as any missing heavy atoms in any partially resolved residues. Note that even though PDBFixer can add any residues/segments that are not resolved in the PDB structure, we are not using this option of PDBFixer. Instead, we leave modeling of those parts to the user. User-provided models should include chain IDs in their PDB files.

At this step, you can also set the pH level of the solution to select the protonation states for adding hydrogens by setting the `pH` parameter (default `pH=7.0`).

[3] http://prody.csb.pitt.edu/tutorials/prody_tutorial/basics.html

```
In [7]: clustenm.setAtoms(pdb)
```

```
@> Fixing the structure ...
@> 3108 atoms and 1 coordinate set(s) were parsed in 0.03s.
@> The structure was fixed in 1.91s.
```

After setting the atoms, you can write the fixed PDB file by the method `ClustENM.writePDBFixed()`.

```
In [8]: clustenm.writePDBFixed()
```

A ClustENMD simulation is started by the `ClustENM.run()` method. This method accepts numerous parameters, and we will only cover the essential ones to perform a simulation in this tutorial. Therefore, we would like to encourage the readers to refer to the docstring of this method for the description of all parameters.

As this method is iterative, the user needs to set the number of generations (default `n_gens=5`). Depending on the system size, its flexibility, and the computational resources available, the user can increase or decrease the number of generations. In this tutorial, we are using its default value.

The parameters regarding the main steps of the method can be grouped as follows:

1. **ANM sampling:**

   `cutoff` : Cutoff distance $(\mathring{A})$ for pairwise interactions used in ANM computations (default is 15.0).

   `n_modes` : Number of global modes for sampling (default is 3).

   `n_confs` : Number of new conformers generated from each parent conformer (default is 50).

   `rmsd` : RMSD $(\mathring{A})$ of new conformers with respect to the parent (default is 1.0).

   `v1` : Full enumeration of ANM modes, which is used in the original ClustENM method (default is False; see below).

   In the current ClustENMD version, ANM sampling is done randomly by the ProDy method `sampleModes`, where the `rmsd` value corresponds to the average RMSD of the new conformers with respect to the parent conformer. As the bigger RMSD value yields larger excursions from the parent, the user should be cautious on increasing its value.

   In contrast, the original ClustENM *[KD16]* uses the full enumeration (all possible combinations) of ANM modes with fixed maximum RMSD, which can be enabled by setting `v1=True`.

   In both cases, we suggest using the first 3 to 5 global modes as they are known to facilitate the conformational transitions.

   The `rmsd` parameter can be not only set to a single value across the generations, but also provided exclusive to each generation as a tuple, e.g., `rmsd=(1.0, 1.5, 1.5)`.

2. **Clustering:**

   `maxclust` : Maximum number of clusters to be formed in each generation (default is None).

   `threshold` : RMSD threshold to apply when forming clusters (default is None).

   We are using SciPy hierarchical clustering library[4] to cluster the conformers in each generation. Either `maxclust` or `threshold` parameter must be specified by the user. As a guideline, we suggest to use the `maxclust` parameter. Furthermore, the parameters can be not only set to a single value across the generations, but also provided exclusive to each generation as a tuple, e.g., `maxclust=(20, 40, 60)`. Increasing the number of maximum clusters in subsequent generations allows for maximum excursion from the initial structure, thus should be preferred.

[4] https://docs.scipy.org/doc/scipy/reference/cluster.hierarchy.html

3. **Relaxation via MD simulations:**

   `temp` : Temperature at which the simulation is conducted (default is 303.15 K).

   `solvent` : Solvent model to be used. Default is `'imp'`, which corresponds to the implicit solvent model (`'amber99sbildn.xml'`, `'amber99_obc.xml'`). To choose the explicit solvent model (`'amber14-all.xml'`,`'amber14/tip3pfb.xml'`), `solvent` should be set to `'exp'`. The user may choose other force fields available in OpenMM, please see the description of `force_field` parameter. However, only the default force-fields named above have been tested in ClustENMD so far. In the current implementation of ClustENMD, implicit solvent model is applicable to protein chains only. If there are any DNA/RNA chains in your structure, ClustENMD automatically uses explicit solvent.

   `padding` : Padding distance to be used for solvation (default is 1.0 nm).

   `ionicStrength` : Total concentration of ions (both positive and negative) to add. This does not include ions that are added to neutralize the system. Default concentration is 0.0 molar.

   `tolerance` : Energy tolerance to be used for performing a local energy minimization on the system (default is 10.0 kJ/mole).

   `maxIterations` : Maximum number of iterations to perform during energy minimization. If this is 0 (default), minimization is continued until the results converge without regard to how many iterations it takes.

   `sim` : A short MD simulation using a time step of 2.0 fs is performed if `sim=True`. Note that there is also a *heating-up phase* until the desired temperature is reached before the short MD simulation. If `sim` is set to False, only energy minizimation is performed. If only a heating-up phase is to be performed, the parameters `t_steps_i` and `t_steps_g` should be set to 0 with `sim=True` (please see below).

   `t_steps_i` : Number of simulation steps for the starting conformer, i.e. zeroth generation, (default is 1000).

   `t_steps_g` : Number of simulation steps for all conformers except the starting conformer, (default is 7500). If desired, time steps for subsequent generations can be varied and given as a tuple, e.g., (3000, 5000, 7000).

   `platform` : Achitecture on which the OpenMM runs (default is None). It can be chosen as `'CUDA'`, `'OpenCL'`, or `'CPU'`. For efficiency, `'CUDA'` or `'OpenCL'` is highly recommended.

We suggest to use implicit solvation and GPU platform for computational efficiency. Default parameters are highly efficient on GPU platform for proteins comprising several thousand residues. For larger assemblies, the user may prefer: (*i*) to decrease the number of clusters and/or generations, (*ii*) to perform only energy minimization with/out heating-up phase, or (*iii*) to carefully shrink the padding distance in explicit solvent.

2.2 Running a ClustENMD simulation

In the following, we will perform a ClustENMD simulation of 5 generations using the first 3 global modes. Relaxation of conformers is carried out in implicit solvent via energy minimization followed by a heating-up phase. We are conducting the simulation on a GPU platform. Simulation details will be printed out during execution.

```
In [9]: clustenm.run(n_modes=3, n_gens=5,
   ...:             maxclust=tuple(range(20, 120, 20)),
   ...:             sim=True, solvent='imp',
   ...:             t_steps_i=0, t_steps_g=0,
   ...:             platform='CUDA')
   ...:
```

```
@> Kirchhoff was built in 0.02s.
@> Generation 0 ...
@> Minimization & heating-up in generation 0 ...
@> Completed in 1.94s.
@> #-------------------/*\-------------------#
@> Generation 1 ...
```

```
@> Sampling conformers in generation 1 ...
@> Hessian was built in 0.07s.
@> 3 modes were calculated in 0.04s.
@> Parameter: rmsd = 1.00 A
@> Parameter: n_confs = 50
@> Modes are scaled by 24.611726681118544.
@> Clustering in generation 1 ...
@> Centroids were generated in 0.24s.
@> Minimization & heating-up in generation 1 ...
@> Structures were sampled in 33.37s.
@> #------------------/*\------------------#
@> Generation 2 ...
@> Sampling conformers in generation 2 ...
@> Hessian was built in 0.07s.
@> 3 modes were calculated in 0.08s.
@> Parameter: rmsd = 1.00 A
@> Parameter: n_confs = 50
@> Modes are scaled by 21.96801859205728.
@> Hessian was built in 0.06s.
@> 3 modes were calculated in 0.07s.
...
@> #------------------/*\------------------#
@> Generation 5 ...
@> Sampling conformers in generation 5 ...
@> Hessian was built in 0.06s.
@> 3 modes were calculated in 0.03s.
@> Parameter: rmsd = 1.00 A
@> Parameter: n_confs = 50
@> Modes are scaled by 19.25666801776903.
...
@> Clustering in generation 5 ...
@> Centroids were generated in 14.04s.
@> Minimization & heating-up in generation 5 ...
@> Structures were sampled in 174.84s.
@> #------------------/*\------------------#
@> Creating an ensemble of conformers ...
@> Ensemble was created in 0.00s.
@> All completed in 558.38s.
```

The generated conformers are stored in a ClustENM ensemble object. For future reference, the parameters set for a simulation can be saved into a file by the method `ClustENM.writeParameters()`:

```
In [10]: clustenm.writeParameters()
```

As ClustENM ensemble is actually a ProDy ensemble[5], we can also save it by using the `saveEnsemble()` method:

```
In [11]: saveEnsemble(clustenm)
```

```
'1tw7_clustenm.ens.npz'
```

We also provide a method, called `ClustENM.writePDB()`, to write the conformers into a PDB file. The boolean parameter `single` (default is `True`) of this method controls whether the conformers are stored as models in a single PDB file, or each of them are saved as a separate PDB file.

[5] http://prody.csb.pitt.edu/manual/reference/ensemble/index.html

```
In [12]: clustenm.writePDB()
```

```
@> PDB file saved as 1tw7_clustenm.pdb
```

One can also load the previously saved ensemble using `loadEnsemble()`

```
In [13]: saved_clustenm = loadEnsemble('1tw7_clustenm.ens.npz')
```

2.3 Features of ClustENM ensembles

As we mentioned above, ClustENM class is derived from ProDy ensemble class, therefore the methods defined for the latter, such as `ClustENM.getCoordsets()`, `ClustENM.superpose()` and many more can apply to ClustENM objects as well. All conformers in generations ($i = 1, 2, 3, \ldots$) are automatically superposed onto the initial/zeroth conformer based on C^{α}-atoms during a ClustENMD simulation.

There are alternative ways of indexing the generated conformers. User can either index ClustENM object by `clustenm[3]`, which picks the 4th conformer (presumably the 2nd conformer in the 1st generation) or equivalently with the generation number and an index as `clustenm[1, 2]`. Note that indices start from 0.

Let's check we obtain the same coordinates by two alternative methods:

```
In [14]: np.allclose(clustenm[3].getCoords(), clustenm[1, 2].getCoords())
```

```
True
```

A ClustENM object supports slicing as well. For example, if we want to select the 4th conformer for every generation, then we only need to specify the index of the conformer in the second slot and select all in the first slot. If the desired conformers are not available in a particular generation, then they will be skipped.

```
In [15]: clustenm[:, 3]
```

```
<ClustENM: 1tw7_clustenm (5 conformations; 3108 atoms)>
```

We can access the coordinates of these conformers by the `ClustENM.getCoordsets()` method:

```
In [16]: clustenm[:, 3].getCoordsets()
```

```
array([[[ -3.95957387,  32.35691799,  -4.37383242],
        [ -4.94566778,  32.35594469,  -4.59228821],
        [ -3.63788137,  31.46009385,  -4.70897438],
        ...,
        [ -2.37337274,  29.5071206 ,  -3.7201629 ],
        [ -1.39627789,  29.60381804,  -3.27034612],
        [ -7.98974581,  31.21050202,  -4.31887029]],

        [[ -6.89570222,  32.89490785,  -5.27764023],
        [ -7.80893237,  32.7297113 ,  -5.67617107],
        [ -6.31021832,  32.07285054,  -5.23854147],
        ...,
        [ -5.32171232,  30.53324814,  -3.46080742],
        [ -4.58778402,  30.86851485,  -2.74293152],
        [-10.41683474,  31.15561532,  -5.46381784]],

        [[ -6.3447726 ,  34.20123262,  -5.5673921 ],
        [ -7.22727328,  34.01664711,  -6.02260974],
        [ -5.82362403,  33.34645491,  -5.43376411],
        ...,
        [ -4.07602444,  31.36764316,  -4.08790043],
```

```
        [ -3.22430149,  31.72057964,  -3.52540378],
        [-10.13066977,  31.95881599,  -6.06925207]],

        [[ -6.03426394,  33.17008188,  -5.2525952 ],
        [ -6.90546384,  32.76869162,  -5.56882538],
        [ -5.41631979,  32.40739972,  -5.01477094],
        ...,
        [ -4.18322255,  30.96462084,  -3.54549089],
        [ -3.39843848,  31.42003303,  -2.95973127],
        [-10.00982495,  30.65422159,  -6.45285668]],

        [[ -5.90545369,  33.39176383,  -5.49324755],
        [ -6.79399411,  33.26907861,  -5.95751872],
        [ -5.56441284,  32.44150355,  -5.52143941],
        ...,
        [ -2.89975089,  29.95653924,  -5.45052765],
        [ -1.8757943 ,  30.2292032 ,  -5.24180161],
        [ -9.38759977,  30.58004821,  -5.53001208]]])
```

On the other hand, we may want to select all the conformers of a specific generation. It is then enough to set the index of the generation in the first slot and select all in the second slot.

```
In [17]: clustenm[3, :]
```

```
<ClustENM: 1tw7_clustenm (60 conformations; 3108 atoms)>
```

2.4 Analysing the results

We would like to show how the computed conformers populate the conformational space as regards the essential dynamics of the structure. For this aim, we perform a principal component analysis (PCA) on the generated ensemble. Next, we will project the conformers onto the space spanned by the first two PCs, which explain the highest variance of the ensemble. This can be done using ProDy ensemble analysis[6].

We are calculating PCs based on the C^{α}-atoms. This selection can be done directly on the ClustENM object.

```
In [18]: clustenm.select('ca')
```

```
In [19]: clustenm
```

```
<ClustENM: 1tw7_clustenm (301 conformations; selected 198 of 3108 atoms)>
```

```
In [20]: pca_clustenm = PCA()

In [21]: pca_clustenm.buildCovariance(clustenm)

In [22]: pca_clustenm.calcModes()
```

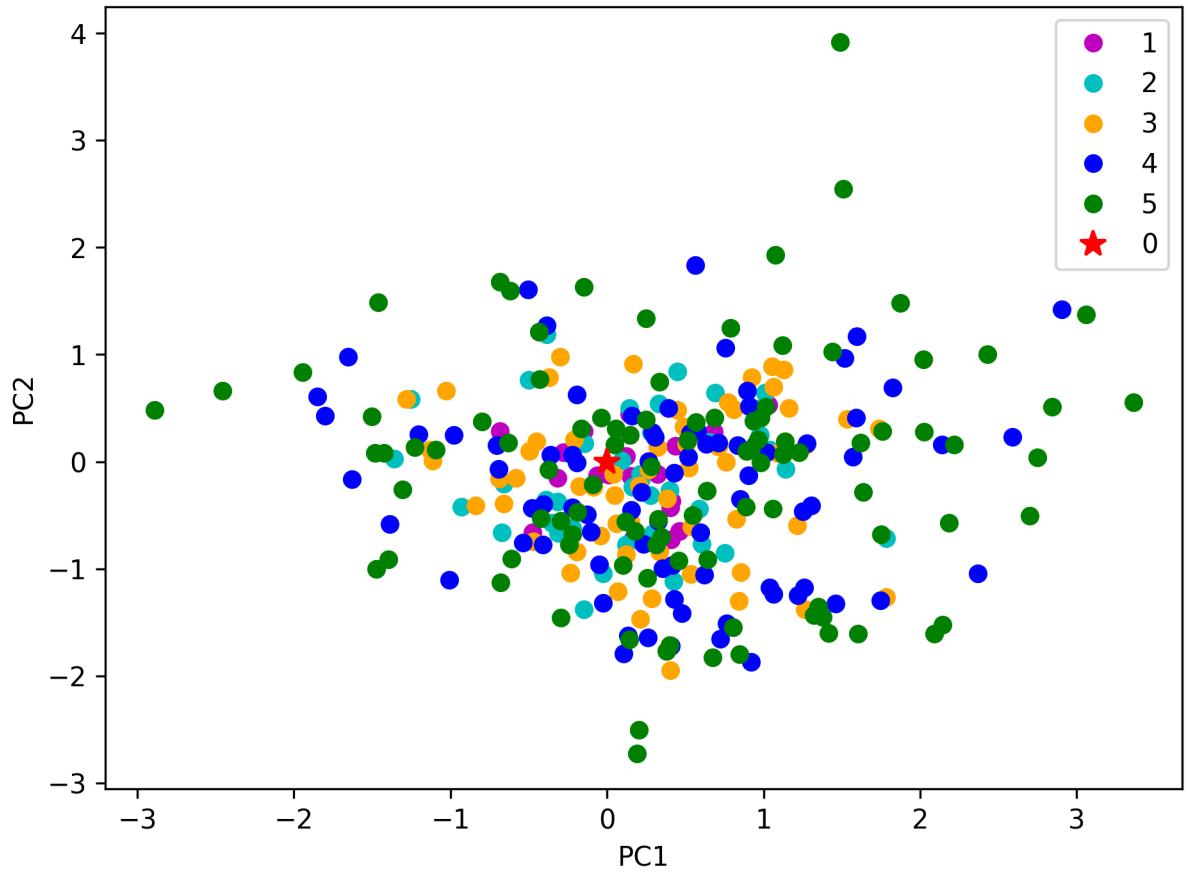
```
@> Covariance is calculated using 301 coordinate sets.
@> Covariance matrix calculated in 0.016746s.
@> 20 modes were calculated in 0.06s.
```

We can observe the progression of the conformers by coloring them in successive generations (from initial/zeroth to the last/fifth).

```
In [23]: with plt.style.context({'figure.dpi': 300,
   ....:                         'axes.labelsize': 'x-large',
```

[6] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/

```
....:                           'xtick.labelsize': 'large',
....:                           'ytick.labelsize': 'large'}):
....:     colors = ['r', 'm', 'c', 'orange', 'blue', 'green']
....:     plt.figure()
....:     for i in range(1, clustenm.numGenerations() + 1):
....:         showProjection(clustenm[i, :], pca_clustenm[:2],
....:                        c=colors[i], label='%d'%i)
....:     showProjection(clustenm[0, :], pca_clustenm[:2],
....:                    c=colors[0], label='0',
....:                    marker='*', markersize=10)
....:     plt.xlabel('PC1')
....:     plt.ylabel('PC2')
....:     plt.legend()
....:     plt.tight_layout()
....:     plt.show()
....:
```



The median and maximum RMSDs with respect to the initial conformer can be calculated for the whole ensemble by

```
In [24]: rmsds = clustenm.getRMSDs()
```

```
In [25]: np.median(rmsds), np.max(rmsds)
```

```
(1.6681441595969058, 4.407775779940453)
```

One can also check the RMSDs of the conformers in each generation with respect to the initial conformer:

```
In [26]: rmsd_gens = []

In [27]: for i in range(1, clustenm.numGenerations()+1):
   ....:     tmp = calcRMSD(clustenm.getCoords(),
   ....:                    clustenm[i, :].getCoordsets())
   ....:     rmsd_gens.append([tmp.min(), tmp.mean(), tmp.max()])
   ....:

In [28]: rmsd_gens = np.array(rmsd_gens)
```

```
In [29]: with plt.style.context({'figure.dpi': 300,
   ....:                         'axes.labelsize': 'x-large',
   ....:                         'xtick.labelsize': 'large',
   ....:                         'ytick.labelsize': 'large'}):
   ....:     plt.figure()
   ....:     plt.bar(np.arange(1, 6)-0.15, rmsd_gens[:, 0],
   ....:             width=.15, color='c', label='min')
   ....:     plt.bar(np.arange(1, 6), rmsd_gens[:, 1],
   ....:             width=.15, color='m', label='mean')
   ....:     plt.bar(np.arange(1, 6)+0.15, rmsd_gens[:, 2],
   ....:             width=.15, color='r', label='max')
   ....:     plt.xlabel('Generation')
   ....:     plt.ylabel(r'RMSD($\AA$)')
   ....:     plt.tight_layout()
   ....:     plt.show()
   ....:
```

We want to also observe if our conformers approach the closed state of HIV-1 protease. For this purpose, an NMR ensemble of 28 models (PDB ID: 1bve with closed flaps) is projected onto the same subspace.

Let's first fetch these models and superpose them onto the initial/zeroth conformer. For this step, we generate a temporary ensemble of NMR models.

```
In [30]: closed = parsePDB('1bve', subset='ca', compressed=False)
```

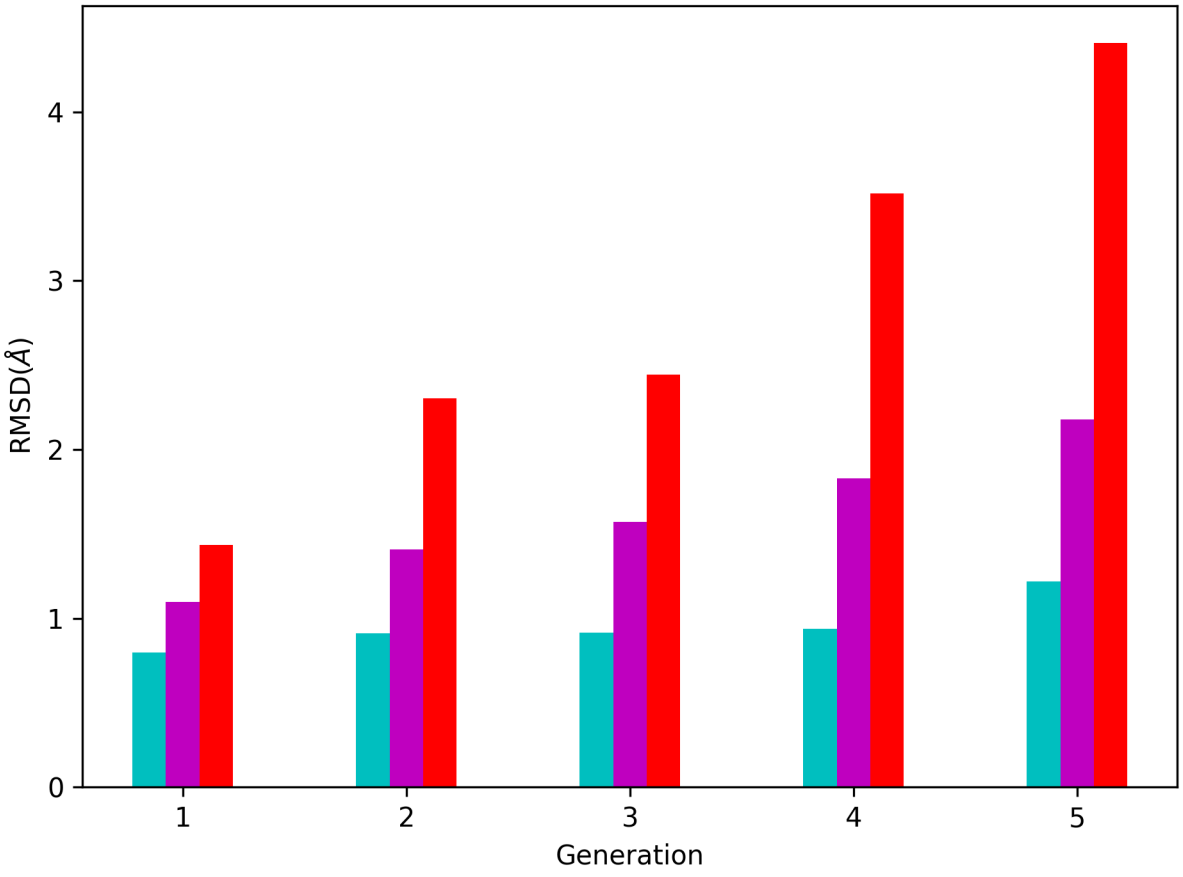
```
@> PDB file is found in working directory (1bve.pdb).
@> 198 atoms and 28 coordinate set(s) were parsed in 0.10s.
```

```
In [31]: ens_cl = Ensemble()

In [32]: ens_cl.setAtoms(closed)

In [33]: ens_cl.setCoords(clustenm.getCoords())

In [34]: ens_cl.addCoordset(closed.getCoordsets())

In [35]: ens_cl.superpose()
```

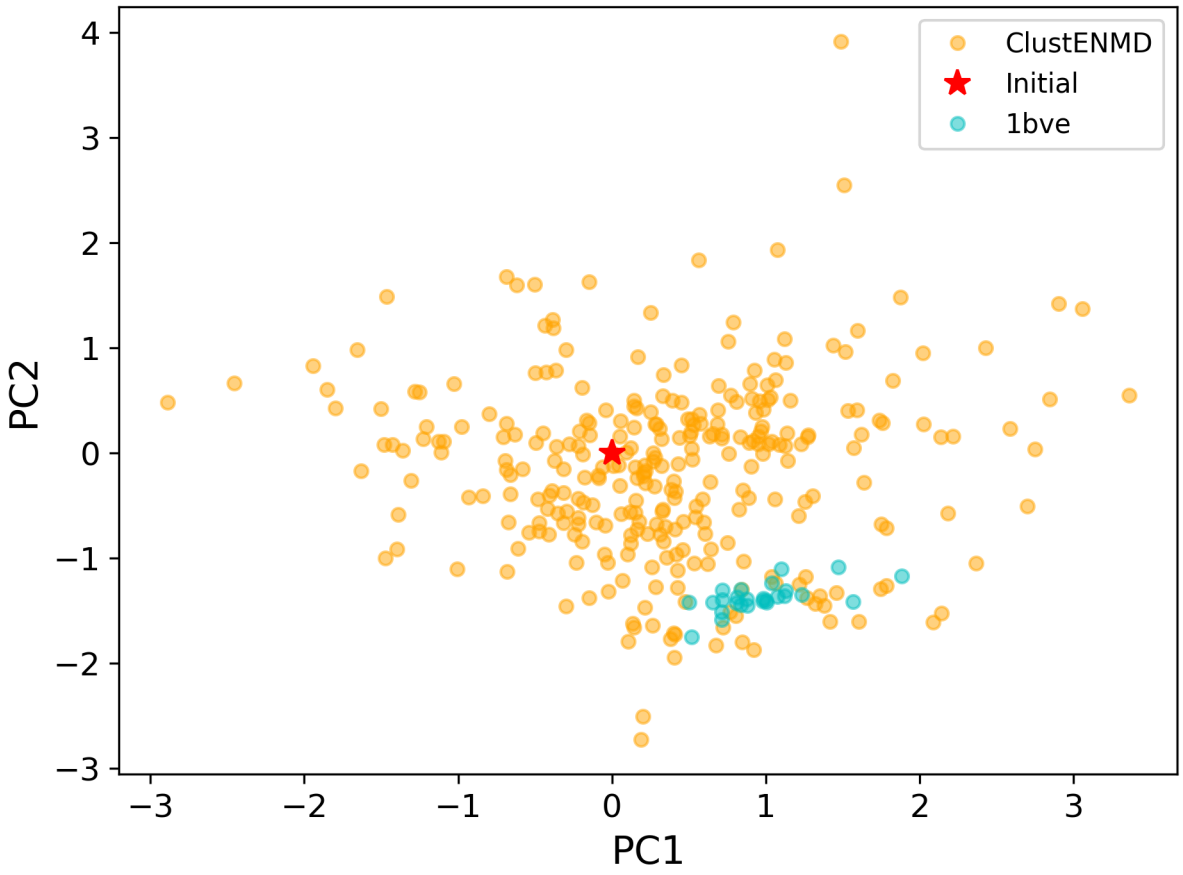
```
@> Superposition completed in 0.03 seconds.
```

At this point, we will project both ClustENMD and NMR conformers on the subspace spanned by the first two PCs of the ClustENMD ensemble.

```
In [36]: with plt.style.context({'figure.dpi': 300,
   ....:                         'axes.labelsize': 'x-large',
   ....:                         'xtick.labelsize': 'large',
   ....:                         'ytick.labelsize': 'large'}):
   ....:     plt.figure()
   ....:     showProjection(clustenmd, pca_clustenmd[:2],
```


RMSD(Å)
0
1
2
3
4
1
2
3
4
5
Generation

```
....:                         c='orange', markersize=5, alpha=.5, label='ClustENMD')
....:     showProjection(clustenmd[0], pca_clustenmd[:2],
....:                         c='r', marker='*', markersize=10, label='Initial')
....:     showProjection(ens_cl[2:], pca_clustenmd[:2],
....:                         markersize=5, c='c', label='1bve', alpha=.5)
....:     plt.xlabel('PC1')
....:     plt.ylabel('PC2')
....:     plt.legend()
....:     plt.tight_layout()
....:     plt.show()
....:
```



The figure above indicates that the unbiased conformer generation starting from the open state of HIV-1 protease (red star) can successfully encompass the NMR models representing its closed state (cyan dots). Each time you perform a ClustENMD run, you will obtain a unique ensemble due to the random sampling and MD simulations. Therefore, it is good practice to perform at least three independent runs, and combine the resulting ensembles for analysis.

Note: In this tutorial we showed the variability of our generated conformers following the procedure in our original paper *[KD16]*. An alternative approach could also be followed if there are enough experimentally resolved homologous structures representing alternative states of a flexible protein. In this approach, we can perform PCA on the ensemble of experimental structures and later project the ClustENMD conformers onto the subspace defined by PCs of experimental structures (see the examples in *[KD21]*). The movie on the ClustENMD webpage displays how the distribution, generated by a Gaussian kernel estimate plot, of HIV-1 protease conformational ensemble progresses as more generations are included. In that movie, ClustENMD conformers are projected on the experimental PC1 vs PC2. Specifically, blue surfaces/levels correspond to the progress of the runs starting from open structure.

CHAPTER

THREE

ANMD SIMULATION AND ANALYSIS

First, we will make the following necessary imports **ProDy_**, **NumPy_**, and **Matplotlib_** if you haven't already done it:

```
In [1]: import numpy as np

In [2]: import matplotlib.pyplot as plt

In [3]: from prody import *

In [4]: plt.ion()
```

3.1 mGluR1 initial alignment and assessment of parameters

We start our calculations by parsing the structure, from which we would like to sample conformations. For this tutorial, we use an initial structure of the metabotropic glutamate receptor 1 (mGluR1) N-terminal venus fly trap domain (VFTD) in the open conformation (chain B from PDB structure 1ewk) with missing loops modelled by MODELLER *[FA00]*. This file is available in the following archives can be used to follow this tutorial:

- ClustENMD and ANMD Tutorial Files (TGZ)
- ClustENMD and ANMD Tutorial Files (ZIP)

In the extracted directory, you will find the output of ClustENMD from the previous tutorial as well as two PDB files that we will parse here. The first one is the open chain model 1ewkB_modeller_protein.pdb, which we will use for ANMD.

The second is the closed chain 1ewkA_protein_trim.pdb, which we compare against. This already has the ligand L-glutamate and an extra C-terminal residue that isn't in chain B trimmed away.

```
In [5]: ag1 = parsePDB('1ewkB_modeller_protein.pdb')
```

```
@> 3775 atoms and 1 coordinate set(s) were parsed in 0.03s.
```

Before running ANMD, we first parse the closed chain in the closed state to evaluate what RMSD and which ANM modes are required for the clamshell closure transition.

```
In [6]: ag2 = parsePDB('1ewkA_protein_trim.pdb')
```

```
@> 3552 atoms and 1 coordinate set(s) were parsed in 0.03s.
```

To do this, we find the matching atoms between the chains and superpose them and calculate a deformation vector and compare it to the ANM modes from the starting model.

As we only have extra atoms in the open chain model, corresponding to the modelled top loop, we can just align that chain to the closed chain.

```
In [7]: amap = alignChains(ag1.ca, ag2.ca)[0]

In [8]: amap_alg, T = superpose(amap, ag2.ca, weights=amap.getFlags('mapped'))

In [9]: rmsd = calcRMSD(amap, ag2.ca)
```

```
@> Trying to map atoms based on residue numbers and identities:
@>   Comparing Chain A from 1ewkB_modeller_protein (len=476) with Chain A from 1ewkA_protein_trim:
@>      Mapped: 447 residues match with 100% sequence identity and 100% overlap.
@> Finding the atommaps based on their coverages...
@> Identified that there exists 1 atommap(s) potentially.
```

We find an RMSD of 4.5 $\mathring{A}$, so we will select a maximum RMSD that is higher than this: 6 $\mathring{A}$.

We next calculate the ANM modes for the full system and slice them with a selection based on this atom map to compare against the deformation vector.

```
In [10]: anm, ag1_ca = calcANM(ag1)
```

```
@> Hessian was built in 0.15s.
@> 20 modes were calculated in 0.15s.
```

```
In [11]: ag1_selection = ag1.select('index ' + ' '.join([str(i) for i in amap.getIndices()]))

In [12]: anm_slc, ag1_ca_slc = sliceModel(anm, ag1_ca, ag1_selection)
```

```
In [13]: defvec = calcDeformVector(ag1_selection, ag2.ca)

In [14]: showOverlap(defvec, anm_slc, abs=False)
```

We see that the 1st mode (index 0 in Python) has a strong negative overlap, so we will focus on this mode for ANMD. We will also use the 2nd mode (index 1 in Python) to illustrate that the method can traverse multiple modes in the same execution.

3.2 Running an ANMD simulation

ANMD is implemented as a ProDy function called **:function:'.runAMND'**. The main parameters regarding the main steps of the method are as follows:

`atoms` : a complete atomic model for the calculations. It is ok to be missing some side chain atoms and hydrogens, but not fragments, such as loops.

`num_modes` : Number of global modes for sampling (default is 2).

`num_steps` : Number of steps along each mode in each direction (default is 5).

`max_rmsd` : Maximum RMSD for the first global mode in $\mathring{A}athring A$ (default is 2). Successive modes are downscaled to lower RMSDs based on their frequency.

`skip_modes` : Number of modes to skip if the first modes are not interesting.

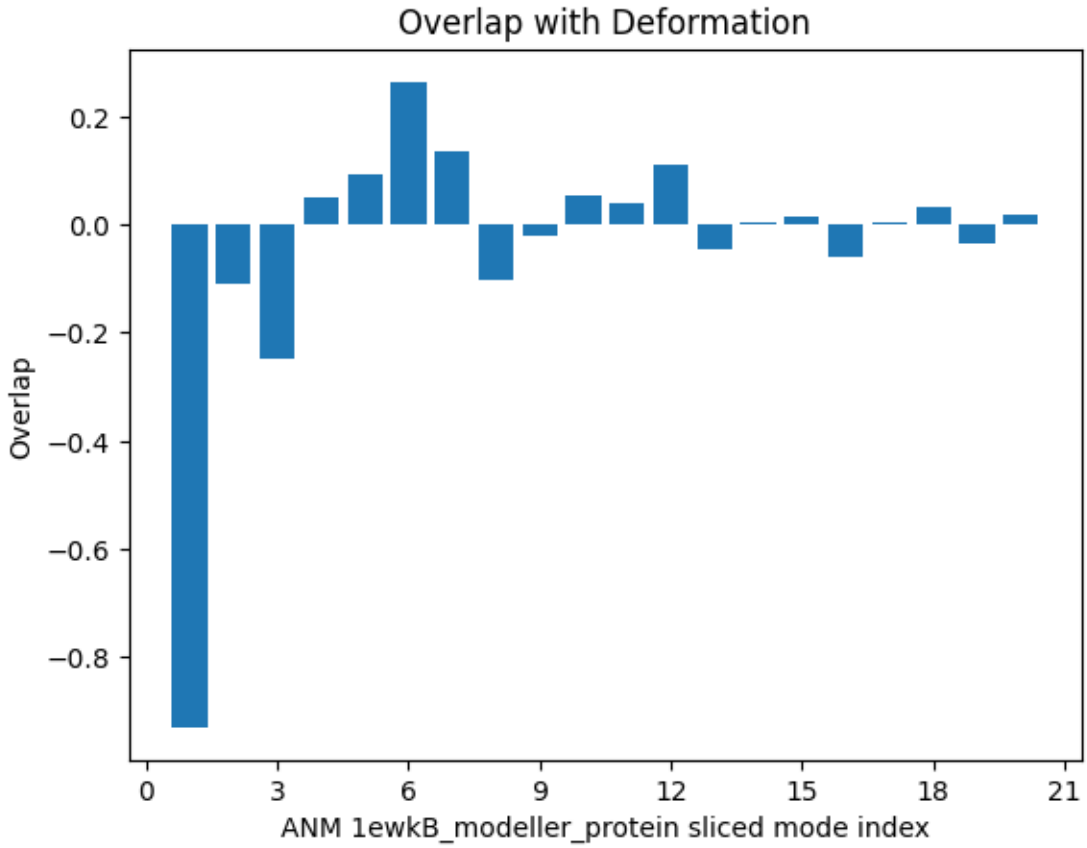
`tolerance` : Energy tolerance to be used for performing a local energy minimization on the system in kJ/mole (default is 10.0).

`anm` : Optional input of your own NMA or ModeSet object to use instead

Other keyword options are also possible for controlling traverse mode:

`pos` : whether to include steps in the positive mode direction, default is **True**

`neg` : whether to include steps in the negative mode direction, default is **True**


Overlap with Deformation
Overlap
0.2
0.0
−0.2
−0.4
−0.6
−0.8
0
3
6
9
12
15
18
21
ANM 1ewkB_modeller_protein sliced mode index

`reverse` : whether to reverse the direction default is **False**

In the following, we will perform ANMD simulations with 5 steps up to a maximum RMSD of 6 $\mathring{A}$ using the first 2 global modes. This means that the first mode has 5 steps of 1.2 $\mathring{A}$, while the second mode has slightly smaller steps.

Relaxation of conformers is carried out in implicit solvent via energy minimization only. Simulation details will be printed out during execution.

We also use keyword options of `traverseMode()` to control the direction along the modes and the output ensemble. The default pos=True, neg=True and reverse=False leads to ensembles with 5 conformations in the negative direction ordered such that the most extreme one is first and the last one is closest to the starting conformation, then the starting conformation, then 5 conformations in the positive direction, giving a total of 11 conformations.

In this case, we only want to follow the negative direction along these modes, so we set pos=False. We also set reverse=True, meaning that rather than starting at the negative extreme of the mode and ordering to approach towards the starting structure, the trajectories start at the starting structure and approach towards the negative extreme.

```
In [15]: ensembles = runANMD(ag1, max_rmsd=6, num_modes=2, num_steps=5,
   ....:                     neg=True, pos=False, reverse=True)
   ....:
```

```
Warning: importing 'simtk.openmm' is deprecated.  Import 'openmm' instead.
@>
Fixed structure found
@>
Minimised fixed structure found
@> 7479 atoms and 1 coordinate set(s) were parsed in 0.07s.
@> Hessian was built in 0.14s.
@> 2 modes were calculated in 0.28s.
@> Parameter: rmsd = 6.00 A
@> Parameter: n_steps = 5
@> Step size is 1.20 A RMSD
@> Mode is scaled by 31.21526594789081.
@>
Minimising 6 conformers for mode 0 ...
@>
Minimising structure 1 along mode 0 ...
@> The structure was minimised in 31.22s.
@>
Minimising structure 2 along mode 0 ...
@> The structure was minimised in 132.40s.
@>
Minimising structure 3 along mode 0 ...
@> The structure was minimised in 171.45s.
@>
Minimising structure 4 along mode 0 ...
@> The structure was minimised in 286.93s.
@>
Minimising structure 5 along mode 0 ...
@> The structure was minimised in 366.90s.
@>
Minimising structure 6 along mode 0 ...
@> The structure was minimised in 459.04s.
@> Parameter: rmsd = 5.45 A
@> Parameter: n_steps = 5
@> Step size is 1.09 A RMSD
@> Mode is scaled by 31.215268055351423.
@>
Minimising 6 conformers for mode 1 ...
```

```
@>
Minimising structure 1 along mode 1 ...
@> The structure was minimised in 30.34s.
@>
Minimising structure 2 along mode 1 ...
@> The structure was minimised in 126.45s.
@>
Minimising structure 3 along mode 1 ...
@> The structure was minimised in 191.12s.
@>
Minimising structure 4 along mode 1 ...
@> The structure was minimised in 1064.22s.
@>
Minimising structure 5 along mode 1 ...
@> The structure was minimised in 327.64s.
@>
Minimising structure 6 along mode 1 ...
@> The structure was minimised in 413.48s.
```

We can also save these using the `saveEnsemble()` method and also write them to PDB files:

```
In [16]: for i, ensemble in enumerate(ensembles):
   ....:     writePDB('1ewkB_mode_{0}_ensemble.pdb'.format(i), ensemble)
   ....:

In [17]: saveEnsemble(ensemble, '1ewkB_mode_{0}_ensemble.ens.npz'.format(i))
```

```
'1ewkB_mode_0_ensemble.ens.npz'
'1ewkB_mode_1_ensemble.ens.npz'
```

One can also load the previously saved ensemble using `loadEnsemble()` or `parsePDB()`.

```
In [18]: ensembles = [Ensemble(parsePDB('1ewkB_mode_{0}_ensemble.pdb'.format(i))) for i in range(2)]

In [19]: ensembles
```

```
@> 7479 atoms and 6 coordinate set(s) were parsed in 0.19s.
@> 7479 atoms and 6 coordinate set(s) were parsed in 0.16s.

[<Ensemble: AtomGroup 1ewkB_mode_0_ensemble (6 conformations; 7479 atoms)>,
 <Ensemble: AtomGroup 1ewkB_mode_1_ensemble (6 conformations; 7479 atoms)>]
```

3.3 Analysing the results

We would like to show how the computed conformers populate the conformational space as regards the essential dynamics of the structure. For this aim, we perform a principal component analysis (PCA) on the generated ensemble. Next, we will project the conformers onto the space spanned by the first two PCs, which explain the highest variance of the ensemble. This can be done using ProDy ensemble analysis[7].

We are calculating PCs based on the C^{α}-atoms and excluding the extra loop. This selection can be done directly on the Ensemble objects but we also add the two ensembles together.

```
In [20]: full_ensemble = ensembles[0] + ensembles[1]

In [21]: full_ensemble.setAtoms(ag1_ca_sel)
```

[7] http://prody.csb.pitt.edu/tutorials/ensemble_analysis/

```
In [22]: for ensemble in ensembles:
   ....:     ensemble.setAtoms(ag1_ca_sel)
   ....:
```

```
In [23]: full_ensemble
```

```
<Ensemble: AtomGroup 1ewkB_mode_0_ensemble + AtomGroup 1ewkB_mode_1_ensemble (12 conformations; selec
```

Next, prior to PCA, we perform an iterative superposition to align the full ensemble onto converged average coordinates.

```
In [24]: full_ensemble.iterpose()

In [25]: for ensemble in ensembles:
   ....:     ensemble.setCoords(full_ensemble.getCoords(selected=False))
   ....:

In [26]: ensemble.superpose()
```

```
In [27]: pca = PCA()

In [28]: pca.buildCovariance(full_ensemble)

In [29]: pca.calcModes()
```

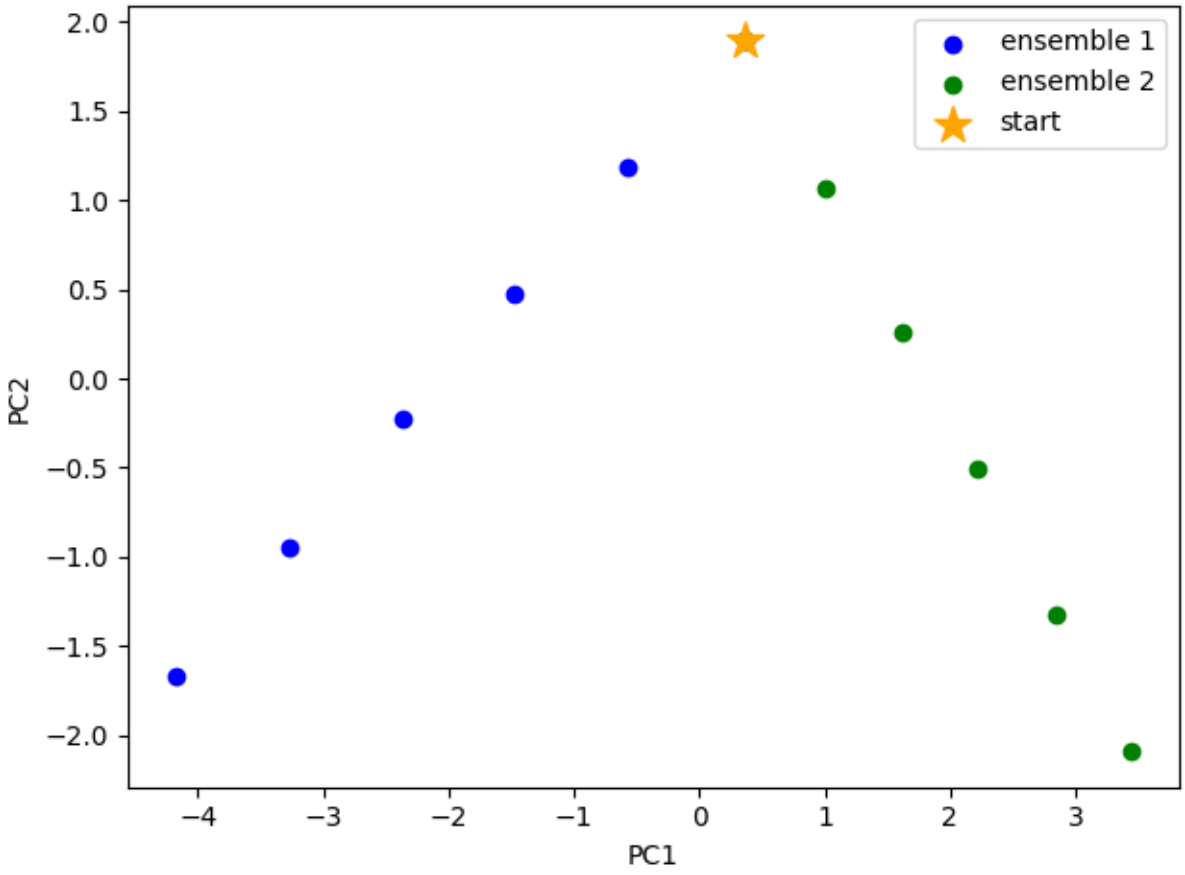
```
@> Covariance is calculated using 12 coordinate sets.
@> Covariance matrix calculated in 0.186342s.
@> 10 modes were calculated in 0.16s.
```

We can observe the progression of the conformers by coloring them by successive modes.

```
In [30]: colors = ['blue', 'green']

In [31]: plt.figure()

In [32]: for i in range(len(ensembles)):
   ....:     showProjection(ensembles[i], pca[:2],
   ....:                    c=colors[i], label='ensemble %d' %(i+1))
   ....:

In [33]: showProjection(ensembles[0][0], pca[:2], c='orange',
   ....:                label='start', marker='*', markersize=200)
   ....:

In [34]: plt.xlabel('PC1')

In [35]: plt.ylabel('PC2')

In [36]: plt.legend()

In [37]: plt.tight_layout()

In [38]: plt.show()
```

The median and maximum RMSDs with respect to the initial conformer can be calculated for the combined full ensemble as follows

```
In [39]: rmsds = full_ensemble.getRMSDs()
```


ensemble 1
ensemble 2
start
PC1
PC2
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−4
−3
−2
−1
0
1
2
3

```
In [40]: np.median(rmsds), np.max(rmsds)
```

```
(2.7045534338432105, 5.733287945495706)
```

We want to also observe if our conformers approach the closed state of the mGluR1 VFTD. As one way to check this, the closed chain (1ewkA that we loaded before) is projected onto the same subspace of 2 PCs.

We therefore need to convert the atomic object to an ensemble one with the CA atoms, and superpose it onto the average coordinates.

```
In [41]: ens2 = Ensemble(ag2.ca.copy())

In [42]: ens2.setCoords(full_ensemble.getCoords(selected=True))

In [43]: ens2.superpose()
```

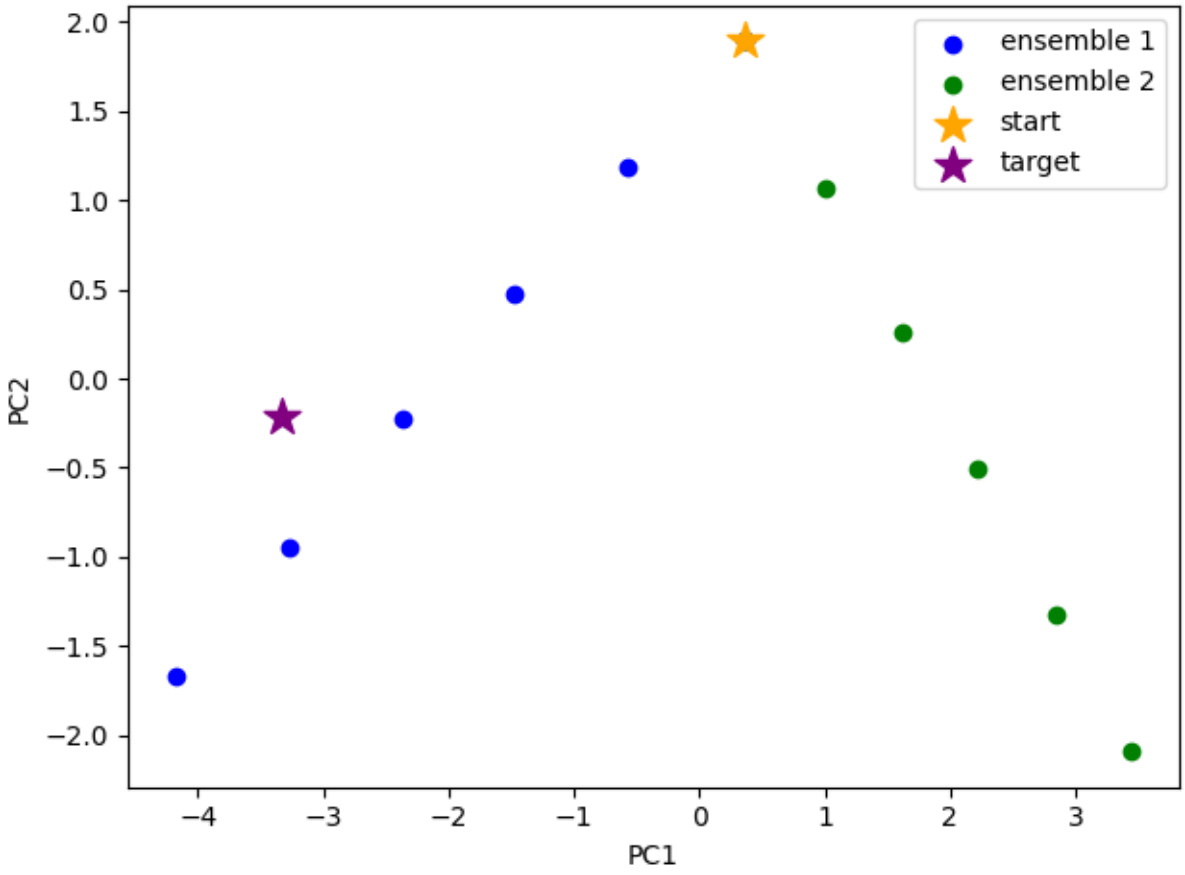
Now, we can plot as before with an extra showProjection command for the target structure as an ensemble onto the two PCs.

```
In [44]: colors = ['blue', 'green']

In [45]: plt.figure()

In [46]: for i in range(len(ensembles)):
   ....:     showProjection(ensembles[i], pca[:2],
   ....:                    c=colors[i], label='ensemble %d' %(i+1))
   ....:

In [47]: showProjection(full_ensemble[0], pca[:2], c='orange',
   ....:                label='start', marker='*', markersize=200)
   ....:

In [48]: showProjection(ens2, pca[:2], c='purple',
   ....:                label='target', marker='*', markersize=200)
   ....:

In [49]: plt.xlabel('PC1')

In [50]: plt.ylabel('PC2')

In [51]: plt.legend()

In [52]: plt.tight_layout()

In [53]: plt.show()
```

The figure above indicates that conformer generation along mode 1 starting from the open state of mGluR1 (orange star) can successfully reach conformations near the closed state (purple star).

One could also calculate RMSDs from the closed state by setting the closed coordinates as the reference coordinates for the ensembles. To do this, we need to first make a coordinates set with the right shape by adding dummy coordinates using alignChains.

```
In [54]: ag3 = ensembles[0].getAtoms(selected=False)

In [55]: amap2 = alignChains(ag2, ag3)[0]

In [56]: amap2
```

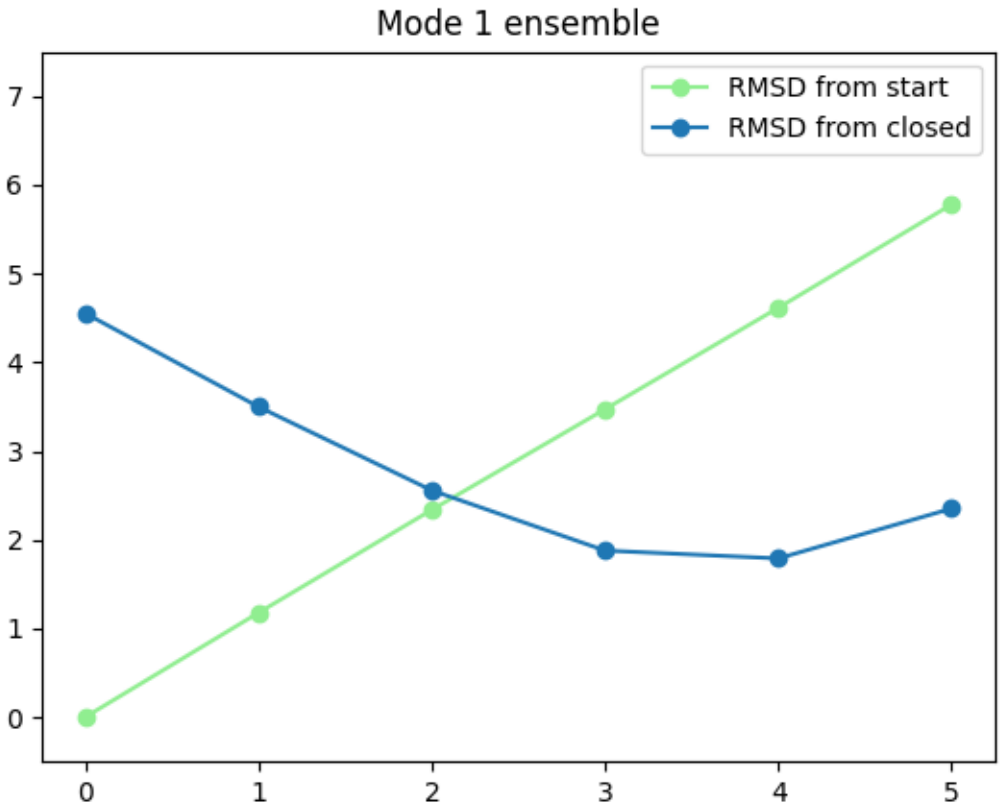

ensemble 1
ensemble 2
start
target
PC1
PC2
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−4
−3
−2
−1
0
1
2
3

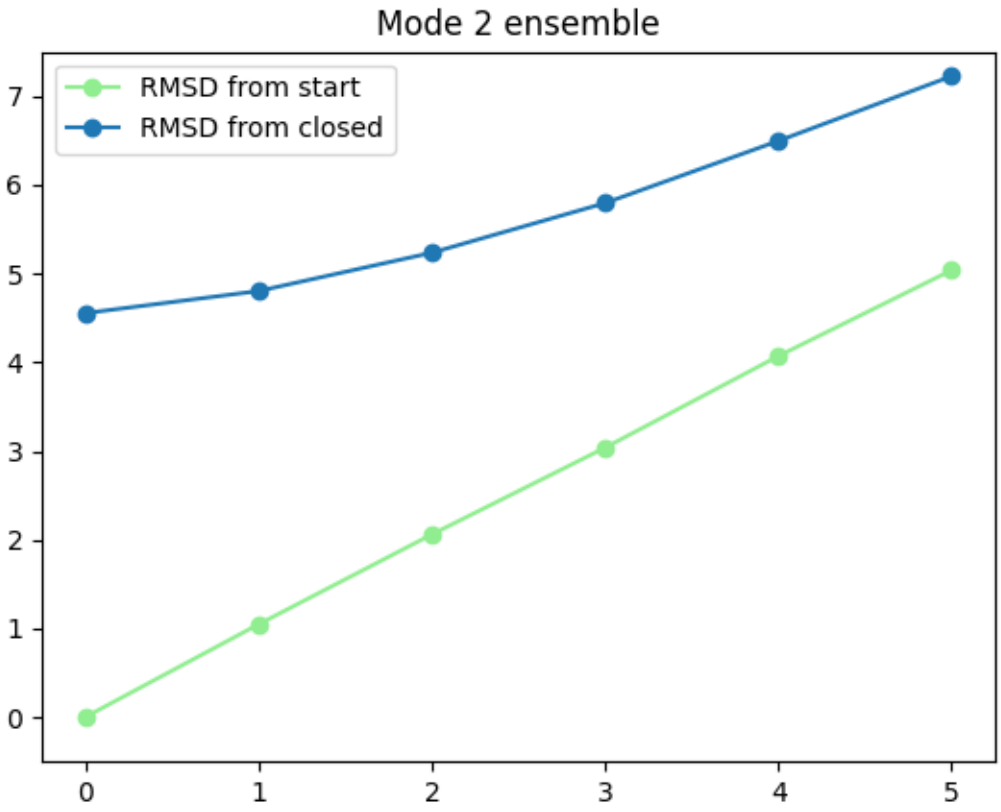
```
<AtomMap: (Chain A from 1ewkA_protein_trim -> Chain A from 1ewkB_mode_0_ensemble) from 1ewkA_protein_
```

Now, we can use this get the RMSDs from the starting and target states.

```
In [57]: rmsds_from_start = np.zeros((2,6))

In [58]: rmsds_from_closed = np.zeros((2,6))

In [59]: for i, ensemble in enumerate(ensembles):
   ....:     ensemble.setCoords(ensemble.getCoordsets(selected=False)[0])
   ....:

In [60]: rmsds_from_start[i] = ensemble.getRMSDs()

In [61]: ensemble.setCoords(amap2.getCoords())

In [62]: rmsds_from_closed[i] = ensemble.getRMSDs()

In [63]: plt.figure()

In [64]: plt.title('Mode {0} ensemble'.format(i+1))

In [65]: plt.plot(rmsds_from_start[i], 'o-', label='RMSD from start', color='lightgreen')

In [66]: plt.plot(rmsds_from_closed[i], 'o-', label='RMSD from closed')

In [67]: plt.ylim([-0.5, 7.5])

In [68]: plt.legend()
```

../template/acknowledgments.rst


Mode 1 ensemble
RMSD from start
RMSD from closed
0
1
2
3
4
5
6
7
0
1
2
3
4
5


Mode 2 ensemble
RMSD from start
RMSD from closed
0
1
2
3
4
5
6
7
0
1
2
3
4
5

[KD16] Kurkcuoglu Z., Bahar I., and Doruker P., ClustENM: ENM-Based Sampling of Essential Conformational Space at Full Atomic Resolution, *J Chem Theory Comput* **2016** 12: 4549.

[KD21] Kaynak B.T., Zhang S., Bahar I., and Doruker P., ClustENMD: Efficient sampling of biomolecular conformational space at atomic resolution, *Bioinformatics* **2021** 37(21): 3956–3958.

[CM22] Mary Hongying Cheng, James M Krieger, Anupam Banerjee, Yufei Xiang, Burak Kaynak, Yi Shi, Moshe Arditi, Ivet Bahar. Impact of new variants on SARS-CoV-2 infectivity and neutralization: A molecular assessment of the alterations in the spike-host protein interactions, *iScience* **2022** 25(3):103939.

[E17] Eastman P., et al. OpenMM 7: Rapid development of high performance algorithms for molecular dynamics, *PLoS Comput Biol* **2017** 13:e1005659.

[FA00] Fiser A, Do RKG, Sali A. Modeling of loops in protein structures. *Protein science* **2000** 9:1753-73