Workshop #4: Folding

In this workshop you will write your own Monte Carlo protein folding algorithm from scratch, and we will explore a couple of the tricks used by Simons *et al.* (1997, 1999) to speed up the folding search.

Suggested Readings

1. K. T. Simons *et al.*, "Assembly of Protein Structures from Fragments," *J. Mol. Biol.* **268**, 209-225 (1997).
2. K. T. Simons *et al.*, "Improved recognition of protein structures," *Proteins* **34**, 82-95 (1999).
3. Chapter 4 (Monte Carlo methods) of M. P. Allen & D. J. Tildesley, *Computer Simulation of Liquids*, Oxford University Press, 1989.

A Simple *de Novo* Folding Algorithm

1. First, we would like to create a simple folding algorithm. Begin with a new pose, and then create an all-atom starting structure with 10 alanines using:

   ```
   pose = pose_from_sequence('A'*10)
   ```

   Use the `PyMOLMover` to echo this structure to PyMOL:

   ```
   from pyrosetta import PyMOLMover
   pmm = PyMOLMover()
   pmm.apply(pose)
   ```

   You should see ideal bond lengths and angles, although the set of ϕ and ψ angles will not be useful.

   Now, write a program that implements a Monte Carlo algorithm to optimize the protein conformation. In the main program, create a loop with 100 iterations. Each iteration should call a subroutine to make a random trial move, score the protein, and then accept or reject the new conformation based on the Metropolis criterion, for which the probability of accepting a move is $P = e^{-\Delta E/kT}$, when $\Delta E \geq 0$, and $P = 1$, when $\Delta E < 0$. Use $kT = 1$ Rosetta Energy Unit (REU).

   For the random trial move, write a *subroutine* to choose one residue at random using `random.randint()` and then randomly perturb either the φ or ψ angles by a random number chosen from a Gaussian distribution. Use the Python built-in function `random.gauss()` from the `random` library with a mean of the current angle and a

standard deviation of 25°. After changing the torsion angle, use `pmm.apply(pose)` to update the structure in PyMOL.

For the energy function, use the standard full-atom scoring approach with *only* the van der Waals and hydrogen bonding terms. With this scoring function, what kind of structures do you expect to be most stable?

At each iteration of the search, output the current pose energy and the lowest energy ever observed. The final output of this program should be the lowest energy conformation that is achieved at any point during the simulation. Be sure to use `low_pose.assign(pose)` rather than `low_pose = pose`, since the latter will only copy a pointer to the original pose.

Output the last pose and the lowest-scoring pose observed and view them in PyMOL. Plot the energy and lowest-energy observed vs. cycle number. What are the energies of the initial, last, and lowest-scoring pose? Is your program working? Has it converged to a good solution?

2. Using the program you wrote for Workshop #2, force the A_{10} sequence into an α-helix. Does this structure have a lower score than that produced by your algorithm? What does this mean about your sampling or discrimination?

3. Since your program is a stochastic search algorithm, it may not produce an ideal structure consistently, so try running the simulation multiple times or with a different number of cycles (if necessary). Using a kT of 1, your program may need to make up to 500,000 iterations.

Low-Resolution (Centroid) Scoring

Following the treatment of Simons *et al.* (1999), Rosetta can score a protein conformation using a low-resolution representation. This will make the energy calculation faster.

4. Load a protein with which you are familiar (*e.g.*, Ras or cetuximab). Calculate the full-atom energy and note the coordinates of residue 5 using `print pose.residue(5)`.

   E: _____ (_____, _____, _____)

5. Convert the pose to the centroid form by using a `SwitchResidueTypeSetMover` object and the `apply` method:

   ```
   switch = SwitchResidueTypeSetMover("centroid")
   switch.apply(pose)
   print pose.residue(5)
   ```

   How many atoms are now in residue 5? _____ How is this different than before?

6. Score the new, centroid-based pose using the standard score function `"score3"`. What is the new total score? _____ What scoring terms are included in `"score3"`? Do these match Simons?

7. Convert the pose back to all-atom form by using another switch mover (`SwitchResidueTypeSetMover("fa_standard")`). Confirm that you have all the atoms back. Are the atoms in the same position as before? _____

8. Adjust your folding algorithm to use centroid residue types. Use only `vdw` and `hbond_sr_bb` energy score components. How much faster does your program run?

Not counting the `PyMOLMover`, which is a special case, `SwitchResidueTypeSetMover` is the first example we have seen of a `Mover` class in PyRosetta. Every `Mover` object in PyRosetta has been designed to apply specific and complex changes (or "moves") to a pose. Every mover must be "constructed" and have any options set before being applied to a pose with the `apply()` method. `SwitchResidueTypeSetMover` has a relatively simple construction with only the single option `"centroid"`. (Some movers, as we shall see, require no options and are programmed to operate with default values.)

Protein Fragments

9. Create a 3-mer fragment file from the Robetta server (http://robetta.bakerlab.org/fragmentsubmit.jsp) for a given test sequence of at least 26 residues of your choice. Before submitting your job, select Exclude homologs. When your job is complete, download the file `aatestA03_05.200_v1_3`. This file contains

3-mer fragments for the test sequence we are trying to fold. You should see sets of three-lines describing each fragment. For the first fragment, which PDB file does it come from? _____ Is this fragment helical, sheet, in a loop, or a combination? _____ What are the ϕ, ψ, and ω angles of the middle residue of the first fragment window?

φ: _____ ψ: _____ ω _____

10. How many 3-residue windows are there in your 20-residue peptide? ____ How many fragments does the data file have per window? _____

11. Create a new subroutine in your folding code for an alternate random move based upon a "fragment insertion". A fragment insertion is the replacement of the torsion angles for a set of consecutive residues with new torsion angles pulled at random from a fragment library file. Prior to calling the subroutine, load the set of fragments from the fragment file:

```
from rosetta.core.fragment import *
fragset = ConstantLengthFragSet(3)
fragset.read_fragment_file("aatestA03_05.200_v1_3")
```

Next, we will construct another `Mover` object — this time a `FragmentMover` — using the above fragment set and a `MoveMap` object as options. A move map specifies which degrees of freedom are allowed to change in the pose when the mover is applied (in this case, all backbone torsion angles):

```
from rosetta.protocols.simple_moves import *
movemap = MoveMap()
movemap.set_bb(True)
mover_3mer = ClassicFragmentMover(fragset, movemap)
```

(Note that when a `MoveMap` is constructed, all degrees of freedom are set to `False` initially. If you still have a PyMOLMover instantiated, you can quickly visualize which degrees of freedom will be allowed by sending your move map to PyMOL with `pmm.send_movemap(pose, movemap)`.)

Each time this mover is applied, it will select a random 3-mer window and insert only the backbone torsion angles from a random matching fragment in the fragment set:

```
mover_3mer.apply(pose)
```

When you change your random move to a fragment insertion, how much faster is your folding code? Does it converge to a protein-like conformation more quickly?

Programming Exercises

1. Fold a 10-mer poly-alanine using 100 independent trajectories, using any variant of the folding algorithm that you like. (A trajectory is a path through the conformation space traveled during the calculation. The end result of each independent trajectory is called a "decoy". Given enough sampling, the lowest energy decoy may correspond to the global minimum.) Create a Ramachandran plot using the lowest-scoring conformations (decoys) from all 100 independent trajectories. Repeat this for a 10-mer poly-glycine. How do the plots differ? Compare with the plots in Richardson's article.

2. Test your folding program's ability to predict a real fold from scratch. Choose a small protein to keep the computation time down, such as Hox-B1 homeobox protein (1B72) or RecA (2REB). How many iterations and how many independent trajectories do you need to run to find a good structure?

3. Modify your folding program to include a simulated annealing temperature schedule, decaying exponentially from $kT = 100$ to $kT = 0.1$ over the course of the search. Again, fold a test protein. Does this approach work better?

4. Modify your folding program to remove the Metropolis criterion and instead accept trial moves *only* when the energy decreases. Plot energy vs. iteration and examine the final output structures from multiple runs. How is the convergence and performance affected? Why?

Thought Questions

1. [Introductory] What are the limitations of these types of folding algorithms?

2. [Advanced] How might you design an intermediate-resolution representation of side chains that has more detail than the centroid approach yet is faster than the full-atom approach? Which types of residues would most benefit from this type of representation?