

Workshop #5: PyRosetta Refinement

Suggested Reading

1. P. Bradley, K. M. S. Misura & D. Baker, "Toward high-resolution de novo structure prediction for small proteins," *Science* **309**, 1868-1871 (2005), including Supplementary Material.
2. Z. Li & H. A. Scheraga, "Monte Carlo-minimization approach to the multiple-minima problem in protein folding," *Proc. Natl. Acad. Sci. USA* **84**, 6611-6615 (1987).

One of the most basic operations in protein structure and design algorithms is manipulation of the protein conformation. In Rosetta, these manipulations are organized into Movers. A Mover object simply changes the conformation of a given pose. It can be simple like a single ϕ or ψ angle change, or complex like an entire refinement protocol.

In the last workshop, you encountered the ClassicFragmentMover, which inserts a short sequence of backbone torsion angles, and the SwitchResidueTypeSetMover, which doesn't actually change the conformation of the pose but instead swaps out the residue types used.

In this workshop, we will introduce a variety of other movers, particularly those used in high-resolution refinement (e.g., in Bradley's 2005 paper).

Before you start, load a test protein and make a copy of the pose so we can compare:

```
start = Pose("test_in.pdb")
test = Pose()
test.assign(start)
```

Small and shear moves

The simplest move types are small moves, which perturb ϕ or ψ of a random residue by a random small angle, and shear moves, which perturb ϕ of a random residue by a small angle and ψ of the same residue by the same small angle of opposite sign.

For convenience, the small and shear movers can do multiple rounds of perturbation. They also check that the new ϕ/ψ combinations are within an allowable region of the Ramachandran plot by using a Metropolis acceptance criterion based on the *rama* score change. (The *rama* score is a statistical score from Simons *et al.* 1999, parametrized by bins of ϕ/ψ space.) Finally, like most movers, these require a MoveMap to specify which degrees of freedom are fixed and which are free to change. Thus, we can create our movers like this:

```
kT = 1.0
n_moves = 1
movemap = MoveMap()
movemap.set_bb(True)
smallmover = SmallMover(movemap, kT, n_moves)
shearmover = ShearMover(movemap, kT, n_moves)
```

We can also adjust the maximum magnitude of the perturbations as follows:

```
smallmover.angle_max('H',25)
smallmover.angle_max('E',25)
smallmover.angle_max('L',25)
```

Here, H, E, and L refer to helical, sheet, and loop residues, and the magnitude is in degrees. Set all the maximum angles to 25° to make the changes easy to visualize.

1. Test your mover by applying it to your pose:

```
smallmover.apply(test)
```

Confirm that the change has occurred by one of two methods: (1) use your old program for printing the ϕ/ψ angles of the *start* and *test* poses to find the change or (2) use `dump_pdb` to output the poses to files, and compare the structures in PyMol. Alternately, you could write a quick program to compare the poses.

Which torsion angles changed? By how much?

2. *Comparing small and shear movers.* Reset the test pose by re-assigning it a conformation from the start position, and create a second test pose in the same manner. Reset the existing MoveMap object to *only* allows backbone angles of residue 50 to move (Hint: set all residues to False, then set just residues 50 and 51 to True using `movemap.set_bb(50,True)` and `movemap.set_bb(51,True)`). Note that the SmallMover is connected to your MoveMap and it will automatically know you have made these changes and use the modified MoveMap in future moves.

Make one small move on one of your test poses, and one shear move on the other test pose. Output all the poses to files and view them in PyMol (show only backbone atoms and view as lines or sticks). Identify the torsion angle changes that occurred. What was the magnitude of the change in sheared pose? How does the displacement of residue 60 compare between the small- and shear-perturbed poses?

Minimization moves

The MinMover carries out a gradient-based minimization to find the nearest local minimum in the energy function, such as that used in one step of the Monte-Carlo-plus-Minimization algorithm of Li & Scheraga.

```
minmover = MinMover()
```

The minmover needs at least a MoveMap and a ScoreFunction. You can also specify different minimization algorithms and a tolerance (see the command reference appendix). For now, set up a new movemap that is flexible from residues 40 to 60, inclusive, using:

```
mm4060 = MoveMap()  
mm4060.set_bb_true_range(40,60)
```

Create a standard, full-atom scorefunction, and then attach these objects to the MinMover:

```
minmover.movemap(mm4060)  
minmover.score_function(scorefxn)
```

3. Apply the MinMover to your sheared pose. Dump the coordinates and compare them in PyMol. How much motion do you see, relative to the original shear move? How far does the C_α atom of residue 60 move?

Monte Carlo object

PyRosetta has several objects for convenience for building more complex algorithms. One example is a MonteCarlo object. This object performs all the bookkeeping you need for creating a Monte Carlo search. That is, it can decide whether to accept or reject a trial conformation, and it keeps track of the lowest-energy conformation and other statistics about the search. Having the Monte Carlo operations packaged together is convenient, especially if we want multiple Monte Carlo loops to nest within each other or to operate on different parts of the protein. To create the object, you need an initial pose, a score function, and a temperature:

```
mc = MonteCarlo(pose, scorefxn, kT)
```

After modifying the protein, the MonteCarlo will automatically accept or reject the protein and update a set of internal counters:

```
mc.boltzmann(pose)
```

4. Test out a MonteCarlo object. Before doing so, you may need to adjust your small and shear moves to smaller maximum angles (3-5°) so they are more likely to be accepted. Apply several small or shear moves, output the score using `print scorefxn(test)` then call `mc.boltzmann(pose)`. A response of 'True' indicates the move is accepted, and 'False' indicates that the move is rejected. If the move is rejected, the pose is reverted to its last-accepted state. Manually iterate a few times between moves and calls to `mc.boltzmann`. Do enough cycles to observe at least two True and two False outputs. Do the acceptances match what you expect given the scores you obtain? After doing a few cycles, use `mc.show_scores()` to find the score of the last-accepted state and the lowest energy state. What energies do you find? Is the last-accepted energy equal to the lowest-energy?

5. See what information is stored in the Monte Carlo object using:

```
mc.show_scores()  
mc.show_counters()  
mc.show_state()
```

What information do you get from each of these?

Trial Mover

A TrialMover combines a mover with a Monte Carlo object. Each time a TrialMover called, it performs a trial move *and* tests that move's acceptance with the MonteCarlo object. You can create a TrialMover from any other type of mover. You might imagine as we start nesting these together, we can build some complex algorithms!

```
trialmover = TrialMover(smallmover, mc)  
trialmover.apply(pose)
```

6. Apply the TrialMover above ten times. Using `trialmover.num_accepts()` and `trialmover.acceptance_rate()`, what do you find?
7. The TrialMover also communicates information to the MonteCarlo object about the type of moves being tried. Create a second TrialMover using a ShearMover and the same MonteCarlo object, and apply this second TrialMover ten times. Look at the MonteCarlo object state (`mc.show_state()`). What are the acceptance rates of each mover? Which mover is accepted most often, and which has the largest energy drop per trial? What are the average energy drops?

Sequence and Repeat Movers

A SequenceMover applies several movers in succession and is useful for building up complex routines.

```
seqmover = SequenceMover()  
seqmover.add_mover(smallmover)  
seqmover.add_mover(shearmover)  
seqmover.add_mover(minmover)
```

This mover will apply first the small, then the shear mover, and finally the minmover.

8. Create a TrialMover using the sequence mover above, and apply it five times to the pose. How is the sequence mover shown by `mc.show_state()`?

A RepeatMover will apply its input mover multiple times each time it is applied:

```
repeatmover = RepeatMover(trialmover,10)
```

9. Use these tools to build up your own ab-initio protocol. Create TrialMovers for 9-mer and 3-mer fragment insertion. Create RepeatMovers for each, and then create TrialMovers for each using the same MonteCarlo object. Create a SequenceMover to do the 9-mer trials and then the 3-mer trials, and iterate the sequence 10 times. Write out a flowchart of your algorithm here:

10. *Hierarchical search.* Write a TrialMover which tries to insert a 9-mer fragment, and then refines the protein with 100 alternating small and shear trials before the next 9-mer fragment trial. The interesting part is this: you will use one MonteCarlo object for the small and shear trials, inside the whole 9-mer combination mover. But use a separate MonteCarlo object for the 9-mer trials. In this way, if a 9-mer fragment insertion is evaluated after the optimization by small and shear moves, and if it is rejected, the pose goes all the way back to before the 9-mer fragment insertion.

Refinement Protocol

The standard Rosetta refinement protocol, similar to that presented in Bradley, Misura & Baker 2005 is available as a mover. Note that the protocol can require ~40 minutes for a 100-residue protein.

```
relax = ClassicRelax()  
relax.apply(pose)
```

Programming exercises

1. Use the mover constructs to create a complex folding algorithm. Create a simple program to do the following:
 - a. Five small moves
 - b. Minimize
 - c. Five shear moves
 - d. Minimize
 - e. Monte Carlo Boltzmann step
 - f. Repeat a-e 100 times
 - g. Repeat a-f five times, each time decreasing the magnitude of the small and shear moves from 25° to 5° in 5° increments.

Sketch a flowchart, and submit both the flowchart and your code.

2. *Ab initio* folding algorithm. Using the Monte Carlo energy optimization algorithm from Workshop 4, write a complete program that will fold a protein. A suggested algorithm involves preliminary low-resolution modifications by fragment insertion (first 9-mers, then 3-mers), followed by high-resolution refinement using small, shear, and minimization movers, as well as side-chain packing.

Test your code by attempting to fold a zinc finger. How do your results compare with the crystal structure? If your lowest-energy conformation is different than the native structure, explain why this is so in terms of the limitations of the computational approach.

Bonus: Dump the pose coordinates as you go and use PyMol to create an animation.

3. *AraC N-terminal arm.* The AraC transcription factor is believed to be activated by the conformational change which occurs in the N-terminus when arabinose binds. Let's test whether PyRosetta can capture this change. Specifically, we will start with the arabinose-bound form, and see if PyRosetta can refold it to the apo form.

Download the arabinose-bound form of the AraC transcription factor. Edit the PDB file so it contains only the arabinose-binding domain, and also remove any non-protein atoms (especially the arabinose). Set up a move map to include only 15 N-terminal residues. Perform an ab-initio search to find the lowest conformation state. How does it compare to the apo crystal form?

Thought questions

1. With $kT = 1$, what is the change in propensity of the rama score that has a 50% chance of being accepted as a small move?
2. How would you test whether an algorithm is effective? That is, what kind of measures can you use? What can you vary within an algorithm to make it more effective?