Workshop #5: PyRosetta Refinement

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.

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).

Introduction

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 later:

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

For convenient viewing in PyMOL, set the names of both poses:

```
start.pdb_info().name("start")
test.pdb_info().name("test")

pmm = PyMOL_Mover()
pmm.apply(start)
pmm.apply(test)
```

We also want to activate the `keep_history` setting so that PyMOL will keep separate frames for each conformation as we modify it (more on this shortly):

```
pmm.keep_history(True)
```

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 component change. (The `rama` score is a statistical score from Simons *et al.* 1999, parametrized by bins of φ/ψ space.) Because they use the Metropolis criterion, we must also supply *kT*. Finally, like most movers, these require a `MoveMap` object 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)
small_mover = SmallMover(movemap, kT, n_moves)
shear_mover = ShearMover(movemap, kT, n_moves)
```

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

```
small_mover.angle_max("H", 25)
small_mover.angle_max("E", 25)
small_mover.angle_max("L", 25)
```

Here, `"H"`, `"E"`, and `"L"` refer to helical, sheet, and loop residues — as they did in the fragment library file — and the magnitude is in degrees. We will set all the maximum angles to 25° to make the changes easy to visualize. (The default values in Rosetta are 0°, 5°, and 6°, respectively.)

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

   ```
   small_mover.apply(test)
   ```

   Confirm that the change has occurred by comparing the `start` and `test` poses in PyMOL. Second, try the PyMOL animation controls on the bottom right corner of the Viewer window. There should be a play button (►) as well as frame-forward, rewind, *etc*. Play the movie to watch PyMOL shuffle your pose move back and forth.

   Can you identify which torsion angles changed? By how much? If it is hard to view on the screen, it may help to use your old programs to compare torsion angles or coordinates.

2. *Comparing small and shear movers*. Reset the test pose by re-assigning it the conformation from `start`, and create a second test pose (`test2`) in the same manner. Reset the existing `MoveMap` object to *only* allow the 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` contains a pointer to your `MoveMap`, and so 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 both poses to PyMOL using the `PyMOL_Mover`. Be sure to set the name of each pose so they are distinguishable 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 the 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.

```
min_mover = MinMover()
```

3. The minimization mover needs at least a `MoveMap` and a `ScoreFunction`. You can also specify different minimization algorithms and a tolerance. (See Appendix A). 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 default `MinMover` with the following methods:

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

   Finally, attach an "observer". The observer is configured to execute a `PyMOL_Mover.apply()` every time a change is observed in the pose coordinates. The `True` is a flag to ensure that PyMOL keeps a history of the moves.

   ```
   observer = PyMOL_Observer(test2, True)
   ```

4. Apply the `MinMover` to your sheared pose. Observe the output in PyMOL. (This may take a couple minutes — the Observer can slow down the minimization significantly.) How much motion do you see, relative to the original shear move? How many coordinate updates does the `MinMover` try? How does the magnitude of the motion change as the minimization continues? At the end, how far has the C_α atom of residue 60 moved?

Monte Carlo Object

PyRosetta has several object classes for convenience for building more complex algorithms. One example is the `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 factor:

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

After the pose is modified by a mover, we tell the `MonteCarlo` object to automatically accept or reject the new conformation and update a set of internal counters by calling:

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

5. 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(test)`. 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 automatically reverted for you 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?

6. 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 `MonteCarlo` object. Each time a `TrialMover` is 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 that, as we start nesting these together, we can build some complex algorithms!

```
trial_mover = TrialMover(small_mover, mc)
trial_mover.apply(pose)
```

7. Apply the `TrialMover` above ten times. Using `trial_mover.num_accepts()` and `trial_mover.acceptance_rate()`, what do you find?

8. 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` is another combination `Mover` and applies several movers in succession. It is useful for building up complex routines and is constructed as follows.

```
seq_mover = SequenceMover()
seq_mover.add_mover(small_mover)
seq_mover.add_mover(shear_mover)
seq_mover.add_mover(min_mover)
```

The above example mover will apply first the small, then the shear, and finally the minimization movers.

9. 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` `n` times each time it is applied:

```
repeat_mover = RepeatMover(trial_mover, n)
```

10. Use these tools to build up your own *ab initio* protocol. Create trial movers for 9-mer and 3-mer fragment insertions. First, create repeat movers for each and then create the trial movers using the same `MonteCarlo` object for each. 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:

11. *Hierarchical search*. Construct a `TrialMover` that 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 is rejected, the pose goes all the way back to before the 9-mer fragment insertion.

Refinement Protocol

The entire 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.set_scorefxn(scorefxn)
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 Metropolis criterion
    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*. Based on 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. Output both your low-resolution intermediate structure and the final refined, high-resolution decoy.

    Test your code by attempting to fold domain 2 of the RecA protein (the last 60 amino acid residues of PDB ID 2REB). How do your results compare with the crystal structure? (Consider both your low-resolution and high-resolution results.) 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: After using the `PyMOL_Mover` or `PyMOL_Observer` to record the trajectory, export the frames and tie them together to create an animation. Search the Internet for "PyMOL animation" for additional tools and tips. Animated GIF files are probably the best quality; MPEG and QuickTime formats are also popular and widely compatible and uploadable to YouTube.

3. *AraC N-terminal arm*. The AraC transcription factor is believed to be activated by the conformational change that 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 that 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 the 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 component 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?