Workshop #7: Docking

Protein–protein docking is the prediction of a complex structure starting from its monomer components. The search space can be extremely large, so large amounts of computational resources are typically required. In this workshop, we will explore several of the techniques briefly; keep in mind that for real applications, many more decoys will need to be tested.

Suggested Readings

1. J. J. Gray *et al.*, "Protein-protein docking with simultaneous optimization of rigid-body displacement and side-chain conformations," *J. Mol. Biol.* **331**, 281-299 (2003).
2. S. Chaudhury & J. J. Gray, "Conformer selection and induced fit in flexible backbone protein-protein docking using computational and NMR ensembles," *J. Mol. Biol.* **381**, 1068-1087 (2008).

Fast Fourier Transform Based Docking via ZDOCK

There are several servers available based on fast Fourier transforms (FFTs). These servers are able to quickly carry out a global, grid-based matching searches.

1. Go to the ZDOCK server (http://zdock.bu.edu) and upload trypsin (2PTN) and its inhibitor (1BA7 chain B) for docking. If completing this workshop for a class, do this in groups in order to not overload the server. When the jobs have finished (typically under an hour), download the output file. You will have to also download a script for creating complexes from the output file. Use the script to generate the top five models. Are these models similar or diverse? _____ How so?

2. Are any of the models similar to the crystal structure of the bound complex (1AVW)? _____

(Other servers include SmoothDock (http://structure.pitt.edu/servers/smoothdock), ClusPro (http://cluspro.bu.edu), Haddock (http://haddock.chem.uu.nl), and GRAMM-X (http://vakser.bioinformatics.ku.edu/resources/gramm/grammx). Any of these provide global docking services to create models that might be useful for refinement by RosettaDock.)

Docking Moves in Rosetta

For the following exercises, download and clean the complex of colicin D and ImmD (1V74). Store three poses — a full-atom starting pose and centroid and full-atom "working" poses.

The fundamental docking move is a rigid-body transformation consisting of a translation and rotation. Any rigid body move also needs to know which part moves and which part is fixed. In Rosetta, this division is known as a "jump" and the set of protein segments and jumps are stored in an object attached to a pose called a "fold tree."

```
print pose.fold_tree()
```

In the fold tree printout, each three number sequence following the word `EDGE` is the beginning and ending residue number, then a code. The codes are `-1` for stretches of protein and any positive integer for a jump, which represents the jump number.

3. View the fold tree of your full-atom pose. How many jumps are there in your pose? ___

By default, there is a jump between the N-terminus of chain A and the N-terminus of chain B, but we can change this using the exposed method `setup_foldtree()`.

```
from pyrosetta.teaching import *
setup_foldtree(pose, "A_B", Vector1([1]))
print pose.fold_tree()
```

The argument `"A_B"` tells Rosetta to make chain A the "rigid" chain and allow chain B to move. If there were more chains in the pdb structure, supplying `"AB_C"` would hold chains A and B rigid together as a single unit and allow chain C to move. (The third argument `Vector1([1])` is required. The function `Vector1()` creates a Rosetta vector object — indexed from 1 — from a Python list. In this case, we pass a list with one element that identifies the first jump in the fold tree for docking use.)

4. Set up a new fold tree for docking using the command above and output the new fold tree. What has changed?

You can see the type of information in the jump by printing it from the pose:

```
jump_num = 1
print pose.jump(jump_num).get_rotation()
print pose.jump(jump_num).get_translation()
```

5. Write out the rotation matrix and the translation vector defined by the jump.

$$\begin{bmatrix} \_\_\_\_\_ & \_\_\_\_\_ & \_\_\_\_\_ \\ \_\_\_\_\_ & \_\_\_\_\_ & \_\_\_\_\_ \\ \_\_\_\_\_ & \_\_\_\_\_ & \_\_\_\_\_ \end{bmatrix}$$

(_____, _____, _____)

The two basic manipulations are translations and rotations. For translation, the change in *x*, *y*, and *z* coordinates are needed as well as the jump number. A rotation requires a center and an axis about which to rotate. The rigid-body displacement can be altered directly with the `RigidBodyTransMover` for translations or the `RigidBodySpinMover` for rotations.

However, for structure prediction calculations, we have a mover that is preconfigured to make random movements varying around set magnitudes (in this case, a mean of 8° rotation and 3 Å translation) located in the `rosetta.protocols.rigid` namespace, (which we will rename with an alias `rigid_moves` for our convenience) :

```
import rosetta.protocols.rigid as rigid_moves
pert_mover = rigid_moves.RigidBodyPerturbMover(jump_num,
                        8, 3)
```

6. Apply the `RigidBodyPerturbMover` to a pose and use a `PyMOLMover` to confirm that the motions are what you expect. What are the new rotation matrix and translation vector in the jump? How many ångströms did the downstream protein move?

Global perturbations are useful for making completely randomized starting structures. The following mover will rotate a protein about its geometric center. The final orientation is equally distributed over the "globe".

```
randomize1 = rigid_moves.RigidBodyRandomizeMover(pose,
                                jump_num,
                                rigid_moves.partner_upstream)
randomize2 = rigid_moves.RigidBodyRandomizeMover(pose,
                                jump_num,
                                rigid_moves.partner_downstream)
```

(`partner_upstream` and `partner_downstream` are predefined terms within the `rosetta.protocols.rigid` namespace, which in our case refer to chains A and B, respectively.)

7. Apply both movers to the starting structure, and view the structure in PyMOL. (You might view it along with the original pose.) Does the new conformation look like a candidate docked structure yet? _____

Since proteins are not spherical, sometimes the random orientation creates severe clashes between the docking partners, and other times it places the partners so they are no longer

touching. The `FaDockingSlideIntoContact` mover will translate the downstream protein along the line of protein centers until the proteins are in contact.

```
from rosetta.protocols.docking import *
slide = DockingSlideIntoContact(jump_num)  # for centroid
                        mode
slide = FaDockingSlideIntoContact(jump_num)  # for full-
                        atom mode
slide.apply(pose)
```

The `MinMover`, which we have previously used to change torsion angles to find the nearest minimum in the score function, can also operate on the jump translation and rotation. It suffices to set the jump variable as moveable in the `MoveMap`:

```
movemap = MoveMap()
movemap.set_jump(jump_num, True)

min_mover = MinMover()
min_mover.movemap(movemap)
min_mover.score_function(scorefxn)  # use any scorefxn
scorefxn(pose)
min_mover.apply(pose)
```

8. Apply the above `MinMover`. How much does the score change? _____ What are the new rotation matrix and translation vector in the jump? How many Ångstroms did the downstream protein move?

Low-Resolution Docking via RosettaDock

RosettaDock can also perform global docking runs, but it can require significant time. Typically, 10^5 to 10^6 decoys are needed in a global run. For this workshop, we will create a much smaller number and learn the tools needed to handle large runs.

Docking is available as a mover that completely encompasses the protocol. To use the mover, you will need a starting pose with both chains and a jump defined. The structure must be in low-resolution (centroid) mode, and you will need the low-resolution docking score function:

```
scorefxn_low = create_score_function("interchain_cen")
```

Randomize your centroid version of the complex. Then, create low-resolution docking structures as follows:

```
dock_lowres = DockingLowRes(scorefxn_low, jump_num)
dock_lowres.apply(pose)
```

9. You can compare structures by calculating the root-mean-squared deviation of all the C_α atoms, using the function `CA_rmsd(pose1, pose2)`. In docking, a more useful measure is the ligand RMSD, which is the deviation of the backbone C_α atoms of the ligand after superposition of the receptor protein backbones. You can calculate ligand RMSD with `calc_Lrmsd(pose1, pose2, Vector1([1]))`. Using both measures, how far did your pose move from the low-resolution search?

10. Examine the created decoy in PyMOL. Does it look like a reasonable structure for a protein-protein complex? _____ Explain.

Job Distributor

For exhaustive searches with Rosetta (docking, refinement, or folding), it is necessary to create a large number of candidate structures, termed "decoys". This is often accomplished by spreading out the work over a large number of computers. Additionally, each decoy created needs to be individually labeled. The object that is responsible for managing the output is called a `JobDistributor`. Here, we will use a simple job distributor to create multiple structures. The following constructor sets the job distributor to create 10 decoys, with filenames like `output_1.pdb`, `output_2.pdb`, *etc*. The pdb files will also include scores according to the `ScoreFunction` provided.

```
from pyrosetta.teaching import *
jd = PyJobDistributor("output", 10, scorefxn_low)
```

It is also useful to compare each decoy to the native structure (if it is known; otherwise any reference structure can be used). The job distributor will do the RMSD calculation and final scoring upon output. To set the native pose:

```
native_pose = pose_from_pdb("your_favorite_protein.pdb")
jd.native_pose = native_pose
```

11. Create a randomized starting pose, working pose, fold tree, score function, job distributor, and low-resolution docking mover. Now, run the low-resolution docking protocol to create a structure, and output a decoy:

```
pose.assign(starting_pose)
dock_lowres.apply(pose)
jd.output_decoy(pose)
```

    Do this twice and confirm that you have two output files.

Whenever the `output_decoy()` method is called, the `current_num` variable of the `JobDistributor` is incremented, and it also outputs an updated score file: `output.fasc`. We can finish the set of 10 decoys by using the `JobDistributor` to set up a loop:

```
while not jd.job_complete:
     pose.assign(starting_pose)
     dock_lowres.apply(pose)
     jd.output_decoy(pose)
```

Note the `jd.job_complete` Boolean variable that indicates whether all 10 decoys have been created.

12. Run the loop to create 10 structures. The score file, `output.fasc` summarizes the energies and RMSDs of all structures created. Examine that file. What is the lowest score? _____ What is the lowest energy? _____

13. Reset the `JobDistributor` to create 100 decoys (or more or less, as the speed of your processor allows) by reconstructing it. Rerun the loop above to make 100 decoys. Use your score file to plot score versus RMSD. (Two easy ways to do this are to import the score file into Excel or to use the Linux command gnuplot.) Do you see an energy funnel? _____

Error! Reference source not found. includes more helpful information about using the `PyJobDistributor` to distribute jobs over multiple computers.

High-Resolution Docking

The high-resolution stage of RosettaDock is also available as a mover. This mover encompasses random rigid-body moves, side-chain packing, and gradient-based minimization in the rigid-body coordinates. High-resolution docking needs an all-atom score function. The optimized docking weights are available as a patch to the standard all-atom energy function.

```
scorefxn_high = create_score_function_ws_patch(
                      "talaris2013", "docking")
dock_hires = DockMCMProtocol()
dock_hires.set_scorefxn(scorefxn_high)
dock_hires.set_partners("A_B")
```

Note that unlike for `DockingLowRes`, we must supply the docking partners with `"A_B"` instead of `jump_num`.

A high-resolution decoy needs side chains. One way to place the side chains is to call the `PackMover`, which will generate a conformation from rotamers. A second way is to copy the side chains from the original monomer structures. This is often helpful for docking calculations since the monomer crystal structures have good side chain positions.

```
recover_sidechains = ReturnSidechainMover(starting_pose)
recover_sidechains.apply(pose)
```

14. Load one of your low-resolution decoys, add the side chains from the starting pose, and refine the decoy using high-resolution docking. How far did the structure move during refinement? _____ How much did the score improve? _____

15. Starting from your lowest-scoring low-resolution decoy, create three high-resolution decoys. (You might use the `JobDistributor`.) Do the same starting from the native structure.

    a. How do the refined-native scores compare to the refined-decoy scores?

    b. What is the RMSD of the refined native? _____ Why is it not zero?

    c. How much variation do you see in the refined native scores? In the refined decoy scores? Is the difference between the refined natives and the refined decoys significant?

Docking Funnel

Using a job distributor and `DockMCMProtocol`, create 10 decoys starting with a `RigidBodyRandomizeMover` perturbation of `partner_downstream`, 10 decoys starting from different local random perturbations (8°, 3 Å), 10 decoys starting from low-resolution decoys, and 10 starting from the native structure. Plot all of these points on a funnel plot. How is the sampling from each method? Does the scoring function discriminate good complexes?

Programming Exercises

1. Output a structure with a 10 Å translation and another with a 30° rotation (both starting from the same starting structure), and load them into PyMOL to confirm the motions are what you expect.

2. *Diffusion.* Make a series of random rigid body perturbations and record the RMSD after each. Plot RMSD versus the number of moves. Does this process emulate diffusion? If it did, how would you know? (Hint: there is a way to plot these data to make them linear.)

3. Starting from a low-resolution docking decoy, refine the structure in three separate ways:

    a. side-chain packing
    b. gradient-based minimization in the rigid-body coordinates
    c. gradient-based minimization in the torsional coordinates
    d. the docking high-resolution protocol

    For each, note the change in RMSD and the change in score. Which operations move the protein the most? Which make the most difference in the score?

4. Using the `MonteCarlo` object, the `RigidBodyMover`, `PackRotamers`, and the `MinMover`, create your own high-resolution docking protocol. Bonus: Can you tune it to beat the standard protocol? "Beating" the standard protocol could mean achieving lower energies, running in faster time, and/or being able to better predict complexes.