In this lab, you will add loop modeling to your mover to help you close your loops after you introduce a random perturbation into your backbone.

**1) Create a new branch**

Checkout your branch from last night and create a new branch from there.

> git checkout yourusername/bootcamp\_lab6

> git personal-tracked-branch bootcamp\_lab8

**2) Refactor your functions from lab 4**

In this lab, you will be using the CCDLoopClosureMover class after making your random perturbation to a particular residue’s phi/psi dihedrals. Your new algorithm will be something like:

iterate N times

pick a random residue

perturb phi/psi for the random residue

close the break using CCD

repack

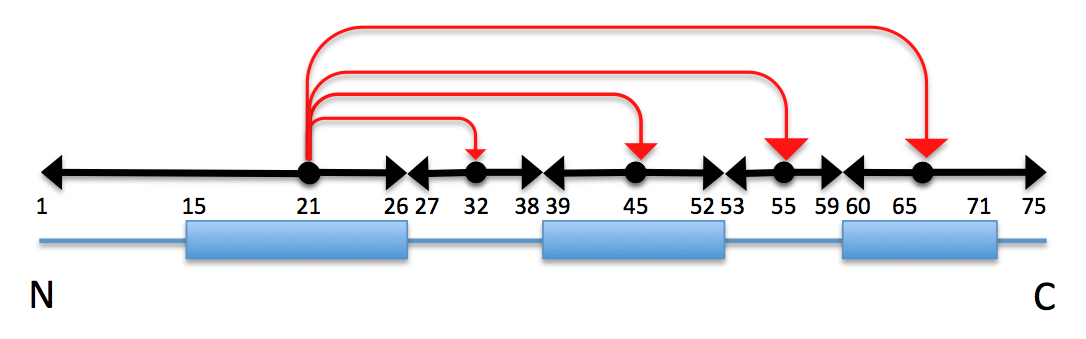
minimize

MonteCarlo accept/reject

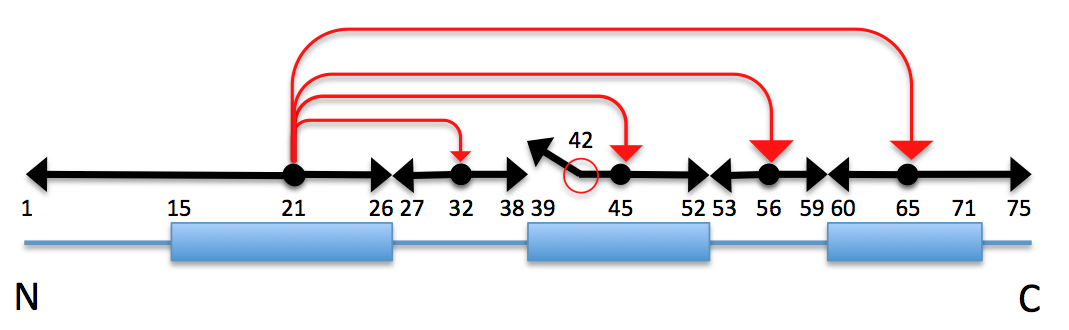
so that you have a closed loop to work with before you enter the repacking and minimization steps. This will mean that the minimizer has less of a gap to close when it begins. When minimization begins and there is a gap, the minimizer is going to try and close that gap first before really considering what the optimal packing for the protein with its loop closed would look like. The horrible chainbreak energy of a broken loop is “louder” than any other energies of the structure, so the minimizer gives all priority to minimizing that. As a result, you will likely end up with a closed gap, but with other parts of the structure trapped at a high energy. The goal here is to help the protocol to respond to the random perturbation in an order that will produce the lowest energy.

In order to use the CCDLoopClosureMover, you will need to give it a Loop object and a MoveMapOP in its constructor. Now, for CCD, “loop” just means a stretch of residues that has a break between the C atom of one residue and the N atom of the next residue. It does not refer to the secondary structure of that stretch, though quite often CCD is applied to stretches of residues that are loops. You can have a loop in the middle of a helix, and CCD will not mind. For the purposes of this lab, you should think of a loop as somewhat arbitrarily chosen stretch of residues with a certain kinetic requirement: the residues before the cutpoint all fold N → C, and the residues after the cutpoint all fold C → N. A Loop class is a convenient way of storing the three pieces of data that define a loop: the first residue in the loop, the last residue in the loop, and the cutpoint residue. The gap is between the cutpoint residue and the next residue, cutpoint+1, if you will. CCD operates by iteratively adjusting the phi and psi angles to minimize the size of the gap between the cutpoint and cutpoint+1 residues, starting at the residues furthest from the cutpoint, and working inwards. It will adjust the phi/psi dihedrals for all residues between the beginning and end positions. So you will need to create and store a list of the loops that might need to be closed as a result of introducing a random perturbation.

The other bit of complexity in introducing this new functionality is that you will need to know which Loop to try and close as a result of picking a particular residue *i* at which you’ll be making your perturbation. You want to try to close the gap that residue i has opened. Well, this is knowable from your fold tree: when you are constructing your fold tree, you have the opportunity to write down for each residue what gap in your protein you would open up if you were to make a perturbation at that residue. Recall the structure of your fold tree that you constructed in Lab 4.



In this fold tree, you have a jump to the middle of each secondary-structure element and to the middle of each loop between two secondary-structure elements. Then, from each middle residue, you have two PEPTIDE edges growing to the boundary of the nearest secondary structure. Consider what would happen, then, if you were to introduce a perturbation to the phi and psi dihedrals at residue 42. Residue 42 is along a peptide edge between residue 45 and residue 39; this edge folds along C → N direction. A perturbation along here would travel down this edge toward the cutpoint between residues 38 and 39, and would stop at residue 39. A gap between residues 38 and 39 would form.



This is the gap you would want CCD to close after your perturbation to residue 42, by making compensatory phi/psi dihedral angles to the residues surrounding this gap, maybe in the range of residues from 36 to 41.

What you need, then, is information about the set of residues surrounding each cutpoint and an assignment “this residue will open up this cutpoint.” With that, you would be able to look up which loop needs closing following every perturbation; in fact, you would then be able to recognize that some residues don’t require a loop closure. For example, a perturbation to residue 70 in the diagram above would move the C-terminus, but would not open up a gap. A change to residue 70 would not need to be followed with a call to the CCD subroutine.

You might want to generate this information at the same time you create your FoldTree, since it might be a little difficult to reconstruct this information when you have just the returned FoldTree created by the the function you wrote in Lab 4.

You are welcome to construct the data you need however you see fit and if you’d prefer to rework your function without the help of the remainder of this section, you are welcome to do so. Do remember to commit your code to your branch regularly.

-----------------------------

Here is a sketch of the data that you might need:

* a vector1 of protocols::loops::Loop objects defining the start, stop, and cutpoint residues, and
* a vector with pose.total\_residue() entries giving for position *i* the index into the loop-definition array above -- and giving an index of “0” in the case that no loop closure is needed

All together, your function needs to create a fair bit of data: the two vectors above, and the FoldTree, which is had created before. There are a few options open:

a) you could return a fairly complicated object using STL pairs, e.g.

std::pair< core::kinematics::FoldTree, std::pair< utility::vector1< protocols::loops::Loop >, utility::vector1< core::Size > > >

b) you could create a small struct that holds the returned object

struct FoldTreeFromSSData {

core::kinematics::FoldTree ft\_;

utility::vector1< protocols::loops::Loop > loop\_vector\_;

utility::vector1< core::Size > loop\_for\_residue\_;

};

or

c) you could create a new class and give this class a Pose in its constructor, so that the class constructs its data as it is initialized, and then allows the user to read this data.

class FoldTreeFromSS {

public:

FoldTreeFromSS( std::string const & ssstring );

core::kinematics::FoldTree const &

fold\_tree() const;

protocols::loops::Loop const &

loop( core::Size index ) const;

core::Size

loop\_for\_residue( core::Size seqpos ) const;

private:

core::kinematics::FoldTree ft\_;

utility::vector1< protocols::loops::Loop > loop\_vector\_;

utility::vector1< core::Size > loop\_for\_residue\_;

};

All three options would work, but the last one is the most “future proof.” If you should learn that you require more data or more functionality out of this code, then it will be easier for you to modify the code. Let’s pursue scenario c) for the sake of practice writing classes and dealing with const-correctness.

So now you should rename your previous file src/protocols/bootcamp/fold\_tree\_from\_ss.hh to src/protocols/bootcamp/FoldTreeFromSS.hh and your .cc files accordingly. This is readily accomplished with this git command (if executed in the Rosetta/main/source directory):

> git mv src/protocols/bootcamp/fold\_tree\_from\_ss.hh src/protocols/bootcamp/FoldTreeFromSS.hh

After you have moved these two files, commit your changes. Go ahead and copy and paste the new class interface in scenario c) above into your FoldTreeFromSS.hh file. You can remove the declaration to your fold\_tree\_from\_ss function -- that will simply become the constructor to the FoldTreeFromSS class. Go ahead and keep the identify\_secondary\_structure\_spans function. In the .cc file, you will need to change the name of the fold\_tree\_from\_ss function to be the constructor to the FoldTreeFromSS class (i.e. FoldTreeFromSS::FoldTreeFromSS( std::string const & ssstring )). Note that the constructor does not provide a return type. (Also note that “ssstring” is arguably an awful variable name.)

The rest of this code is up to you. Write your function to create the Loop objects for each cutpoint (let’s say that the two residues before the cutpoint residue and the three residues after it should define the span of each Loop (e.g. if residue 28 is the cutpoint residue, then the loop start should be residue 26, and the loop end should be residue 31; this is a 6 residue loop) except, of course, if this span would go passed the beginning of the peptide edges. For example, if there are only 4 residues between two secondary structure elements, so that there is a jump to residue 15, a C → N edge from residue 15 to residue 14, and an N → C edge from residue 15 to 17, then the Loop that spans the cutpoint at residue 13 should range from 11 to 14 (changes to the phi and psi dihedrals at residue 15 would propagate down the N → C edge). Write a set of unit tests to make sure your code appropriately creates the Loop objects and correctly assigns each residue to the appropriate loop. You should also update your existing unit tests to use your new class, instead of calling your old function, and make sure that you have not accidentally broken your function’s old functionality. Commit your code regularly as you work.

3) Integrate your new code into your BootCamp mover

Update the old calls to your previous fold\_tree\_from\_ss function so that you can again call your function from your BootCamp mover. Create and use a CCDLoopClosureMover using the mix of psuedocode below:

if ( ftfss.loop\_for\_residue( randres ) != 0 ) {

protocols::loops::Loop ranloop = ftfss.loop( ftfss.loop\_for\_residue( randres ));

std::cout << "Closing loop: " << ranloop.start() << " " << ranloop.stop()

<< " " << ranloop.cut() << std::endl;

protocols::loops::loop\_closure::ccd::CCDLoopClosureMover ccd(

ranloop, mm );

ccd.apply( pose );

}

where you will first check to make sure that the perturbation you’ve just committed at randres needs to be followed by a loop closure function call by looking up the loop index that it corresponds to (an index of 0 is used to signify “no loop”), and then you will create a CCDLoopClosureMover with arguments like the ones given above and then call its apply function passing it your pose. Note that the CCDLoopClosureMover takes a MoveMapOP and not a MoveMap. In lab 2, you may have created a MoveMap; at this point, you will have to replace it with a MoveMapOP to get it to work with the CCDLoopClosureMover. (CCDLoopClosureMover is smart enough to generate a MoveMap from the Loop object, but since you went to all that trouble of making a MoveMap, use it!)

Recompile and run your bootcamp.cc application and watch it as it closes loops following phi/psi perturbations. What happens to the acceptance and rejection rates? What happens to the energies? Commit your code and push your branch to github, then relax (the human action), because you’ve just finished writing a fairly capable relax (the algorithm) protocol!