In this lab, you will be writing code to compute a new FoldTree for a given structure, breaking it into short segments connected by jumps. Your segments will be defined by the Pose’s secondary structure elements, helices and sheets, and the loops connecting these elements will be where you will introduce your cutpoints. You’re going to write your code in one of the protocols libraries, and write unit tests for your code to make sure it works correctly. Once you have written your new functions, you will call it from within the bootcamp.cc application you began developing yesterday.

**1) Create a new branch**

Begin by checking out your branch from yesterday so that you can build from where you finished. Then create a new branch from there.

> git checkout yourusername/bootcamp\_lab2

> git personal-tracked-branch bootcamp\_lab4

**2) Add code to measure the acceptance / rejection rate to your executable in boocamp.cc**

One drawback of an internal-geometry kinematics system that it can cause small structural changes made at one end of the system to amplify as they move distant sections. The default N→C fold tree, which your bootcamp application from yesterday uses, causes changes that occur close to the N-terminus to be propagated all the way through the system. If the structure makes too big of a movement, then it will likely induce a collision, and collisions can be hard to minimize out of. As a result, a very high fraction of the attempted random perturbations will be rejected. Rejected moves happen -- that’s just part of creating a Monte Carlo search strategy -- but the higher the rejection rate, the less efficient its search becomes, and the more computationally demanding it is. Generally, efforts to increase the acceptance rate (besides simply increasing the temperature) often result in more efficient algorithms. Today you will be writing code to change the kinematics of the system you are perturbing with the goal that it will become more efficient.

Start by looking at the MonteCarlo class. It offers routines to tell you whether the last move that it considered was rejected or accepted. Hunt down this routine by reading through the MonteCarlo::boltzmann method defined in src/protocols/moves/MonteCarlo.cc. This code will modify class member data to record what decision it has made (member variables are named with a trailing underscore, e.g. class Foo{ private: int bar\_; };. Find the data member(s?) that get modified inside the boltzmann method. Then find the functions that will grant you access to this data.

Once you have determined which functions you will need to call, add code to compute and print out the acceptance rate every 100 iterations through your loop. You should also compute and print out the average score of your Pose. (HINT: the Pose’s Energies object contains the last computed score for the Pose. Read the names of the member variables and the member functions ti see if what you are looking for jumps out at you.). Run your code a few times to get a sense of the acceptance rates you most commonly see. You will want to compare this acceptance rate with the ones you generate after you have created a new kinematics system for your Pose. Note that version control will let you revert to this version of your code later, to recalculate these acceptance rates, should you forget to write them down...

**3) Write a function to recognize the beginning and ending of secondary structure elements**

(**NOTE**: read through this full section, which remains in its original form, because, you know, it rocks. But you’re not going to have to perform the work for this section. This lab has been abbreviated. At the end of this section, I have provided you with the code that this section had requested you to write.)

You’re going to write a function that takes a std::string representing the secondary structure determination of each residue in a structure and that returns data describing how many elements were found, and who the first and last residue of each element is. This is the first step in the process of initializing the kinematics of a protein to propagate structural changes only through a single secondary-structure element. More on this in part 4 below.

Rosetta includes an implementation of the DSSP secondary-structure determination algorithm, which reports the secondary structure type for every residue in a Pose. It describes helix residues with the letter ‘H’ and strand residues with the letter ‘E’. It recognizes several more subtypes but for this exercise, we’re only going to focus on helix and sheet. So, our code will recognize ONLY letter ‘H’ and ‘E’, any IGNORE other letters. Your function will take a string with DSSP designations and will return a utility::vector1< std::pair< core::Size, core::Size > > where the length of the vector will represent the number of secondary-structure elements, and each pair will represent the first and last residue indices of each SS element.

Your function should be declared with the following signature:

utility::vector1< std::pair< core::Size, core::Size > >

identify\_secondary\_structure\_spans( std::string const & secstruct\_codes );

You’ll put the function, eventually, into a new directory src/protocols/bootcamp, but you will write it and test it within the test/protocols/bootcamp directory. Why? SCons can be a bit pokey and so it’s nice to develop in a context where you can recompile quickly. I often develop functionality within the unit test system before moving it to the rest of the code base. It also makes it much easier to develop the unit tests if you’re writing the code and the unit tests at the same time in the same file, and every time you compile your code, you’ve also compiled your unit tests.

Create a new directory test/protocols/bootcamp. Copy the unit test file test/protocols/match/ProteinSCSampler.cxxtest.hh to the test/protocols/boocamp directory and give the new file a name like “FoldTreeFromSS.cxxtest.hh”; this unit test file will be used both for writing this function and for the next function. Rename the class that’s declared within this file to FoldTreeFromSSTests and remove all of its internals except its setUp() function. Also remove all of the #includes to code in protocols/match; you won’t need any of them. Add a single unit test function

void test\_hello\_world() {

TS\_ASSERT( true );

}

It turns out that a unit-test suite won’t compile unless you have at least one unit test within it. Add your file to the test/protocols.test.settings file and compile it with SCons.

> cd /path/to/Rosetta/main/source

> ./scons.py cat=test > lab4\_build\_commands.txt

saving the output that SCons generates so that you can reissue your scons build command repeatedly like you did in lab 2.

Once you’ve compiled your unit test, open a new terminal window and run your unit test from within the Rosetta/main/source directory with the following command line

> python test/run.py -1 FoldTreeFromSSTests

if you’re using clang to compile, you will need to add the flag -c clang to the command line above. You should see that your as-of-yet empty unit test suite runs and exits with the statement “All tests pass!”. Add your new file to git and stage your modifications to the .test.settings file (git add test/protocols/bootcamp/FoldTreeFromSS.cxxtest.hh test/protocols.test.settings), commit your modifications (run “git status” and “git add” first), and push your branch to github.

Now proceed to write your function. Beware that std::strings are 0-indexed, and that the sequence positions that Rosetta uses are 1-indexed. It might be useful for you to create a utility::vector1< char > and to move the contents of the string into this vector so that you’re only using 1-based indexing throughout your code, but this is your choice.

(It turns out that this kind of “iterate across an array and keep track of the beginning and ending positions” algorithm is really common, and one you’ll find yourself writing many times over the course of your programming career. However, if this is the first time you’ve written something like this, then it might take you quite some time to work out what you need to write. As an example, the following code can look at a sequence of 0’s and 1’s and print out index pairs for each stretch of 1s. E.g. 0011100 should print out: “start: 3 stop: 5”

void find\_ones\_block( utility::vector1< int > const & bitstring ) {

int start = 0;

for ( uint ii = 1; ii <= bitstring.size(); ++ii ) {

// std::cout << "ii: " << ii << " " << bitstring[ ii ] << std::endl;

if ( start != 0 ) {

if ( bitstring[ ii ] != 1 ) {

std::cout << "start: " << start << " stop: " << ii-1 << std::endl;

start = 0;

}

} else {

if ( bitstring[ ii ] == 1 ) {

start = ii;

}

}

}

}

Feel free to model your code on the above function. Your code, though, will need to write down each range -- addin data to a list, perhaps -- instead of outputting it to the screen.)

Write unit tests that call your function and that make sure that it returns the right answers. Here are some example input strings that you should feed your unit test as well as the right answer for those strings.

" EEEEE HHHHHHHH EEEEE IGNOR EEEEEE HHHHHHHHHHH EEEEE HHHH "

with 7 secondary structure elements, spanning residues 4 to 8, 12 to 19, 22 to 26, 36 to 41, 45 to 55, 58 to 62, and 65 to 68 (notice there are some non H/E elements that you should ignore.

"HHHHHHH HHHHHHHHHHHH HHHHHHHHHHHHEEEEEEEEEEHHHHHHH EEEEHHH "

with 7 secondary structure elements, spanning residues 1 to 7, 11 to 22, 29 to 40, 41 to 50, 51 to 57, 59 to 62, and 63 to 65.

and

"EEEEEEEEE EEEEEEEE EEEEEEEEE H EEEEE H H H EEEEEEEE"

with 9 secondary structure elements, spanning residues 1 to 9, 11 to 18, 20 to 28, 30 to 30, 32 to 36, 38 to 38, 40 to 40, 42 to 42, and 44 to 51.

You should write unit tests for each of these three string inputs and make sure that the outputs from your function are correct. Commit your code to your branch (run “git status” and “git add” first) and push your branch to github.

**NOTE**: Part 3 ended here previously.

The new instructures for part 3 are:

Use this function below to determine from a string the start and end points of every secondary-structure element. Put this function into a unit test file as if you had been following the instructions above and had composed it yourself. Write the unit tests for this function from the examples given above. (Be careful as you copy and paste this code because the google quote marks go screwy.)

utility::vector1< std::pair< core::Size, core::Size > >

identify\_secondary\_structure\_spans( std::string const & ss\_string )

{

utility::vector1< std::pair< core::Size, core::Size > > ss\_boundaries;

core::Size strand\_start = -1;

for ( core::Size ii = 0; ii < ss\_string.size() ++ii ) {

if ( ss\_string[ ii ] == 'E' || ss\_string[ ii ] == 'H' ) {

if ( strand\_start == -1 ) {

strand\_start = ii;

} else if ( ss\_string[ii] != ss\_string[strand\_start] ) {

ss\_boundaries.push\_back( std::make\_pair( strand\_start+1, ii ) );

strand\_start = ii;

}

} else {

if ( strand\_start != -1 ) {

ss\_boundaries.push\_back( std::make\_pair( strand\_start+1, ii ) );

strand\_start = -1;

}

}

}

if ( strand\_start != -1 ) {

// last residue was part of a ss-eleemnt

ss\_boundaries.push\_back( std::make\_pair( strand\_start+1, ss\_string.size() ));

}

for ( core::Size ii = 1; ii <= ss\_boundaries.size(); ++ii ) {

std::cout << "SS Element " << ii << " from residue "

<< ss\_boundaries[ ii ].first << " to "

<< ss\_boundaries[ ii ].second << std::endl;

}

return ss\_boundaries;

}

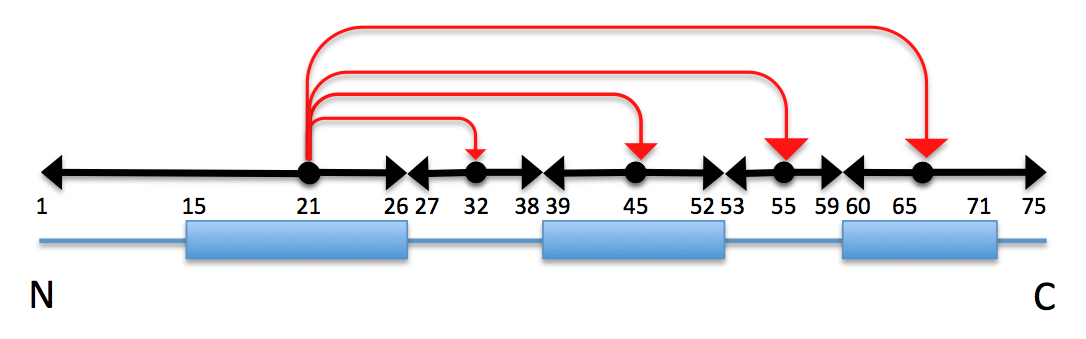
**4) Write a function that creates a new FoldTree for an input secondary structure string**

The next stage is to produce a new function, fold\_tree\_from\_ss, that takes a Pose and returns a FoldTree. You’ll actually write two functions: one that takes a Pose and returns a FoldTree, and a second that takes a string and returns a FoldTree. Your first function will call your second function. fold\_tree\_from\_dssp\_string will take the string returned by the DSSP calculation code. The first function will ask DSSP for a string, and then hand that string to the second function.

The fold tree your second function will create should have 4N-2 peptide edges\* and 2N-2 jump edges if there are N secondary structure elements in your Pose. It should have a jump between the middle residue of the first secondary-structure element to the middle residue of every other secondary-structure element. It should also have a jump between the middle residue of the first secondary-structure element to the middle residue of every inter-secondary-structure gap (each “loop”). It should have a peptide edge starting at the middle residue of each secondary structure element, and going to the end of that secondary structure element. It should also have a peptide edge starting at the middle residue of each secondary structure element and going backwards toward the beginning of that secondary structure element. It should have a similar pair of edges for each loop.

(\*Actually, there are reasons why your fold tree might have fewer peptide edges; I will detail them below).

Take, for example, this protein whose secondary structure elements are shown as blue boxes.



The correct fold tree for this structure should have four jumps (4\*3-2, red arrows) and ten peptide edges (2\*3-2, black arrows). Note that the peptide edge between residue 21 and 26 does not overlap with the peptide edge starting at residue 32 and going back toward 27. Note also that the first edge starts at residue 21 and goes backwards toward residue 1. Residue 21 is the root of this fold tree.

To create this fold tree, you should use the default constructor to create an empty FoldTree and then use the FoldTree::add\_edge function. This function lets you add edges between nodes without first saying that the nodes need to exist. You will also specify the direction in which conformational changes should propagate; the index of the first residue should be the upstream residue and the second should be the downstream residue. Use the constant core::kinematics::Edge::PEPTIDE as the third argument when adding a peptide edge, and a (positive) index as the third argument when adding a jump edge.

E.g., though you wouldn’t write your code this way (you will write for loops), you could begin inserting edges into a fold tree to get the one above by using this code:

FoldTree ft;

ft.add\_edge( 21, 1, core::kinematics::Edge::PEPTIDE );

ft.add\_edge( 21, 26, core::kinematics::Edge::PEPTIDE );

ft.add\_edge( 21, 32, 1 ); // first jump edge

ft.add\_edge( 32, 27, core::kinematics::Edge::PEPTIDE );

ft.add\_edge( 32, 38, core::kinematics::Edge::PEPTIDE );

ft.add\_edge( 21, 45, 2 ); // second jump edge

…etc.

There are three things you need to know about adding edges (at the very minimum). First, what you can consider happening is that the first edge you add declares two nodes in this tree: nodes “21” and “1”. Then for every subsequent edge you add to the graph, *you must have already added the first node to the graph*, (and you must not have yet added the second node). Note that the jump from 21 to 32 has to be added to the fold tree (and thus node 32 has been added to the graph), before the edge from 32 to 27 or the edge from 32 to 38. It would not work to add the edge from 32 to 27 before adding the jump from 21 to 32. Second (and this will come up when you go to test your code), you cannot add an edge from a node to itself; e.g.,

ft.add\_edge( 32, 32, core::kinematics::Edge::PEPTIDE );

Would eventually crash the code. For this reason, it’s possible the number of peptide edges in your fold tree will be fewer than 4N-2. Thirdly, you should never add an edge twice. If you had

ft.add\_edge( 32, 27, core::kinematics::Edge::PEPTIDE );

ft.add\_edge( 32, 27, core::kinematics::Edge::PEPTIDE );

this would lead to the code crashing. It should probably not crash, but it does.

Write unit tests for your fold\_tree\_from\_ss function using the following secondary structure string.

" EEEEEEE EEEEEEE EEEEEEEEE EEEEEEEEEE HHHHHH EEEEEEEEE EEEEE "

The fold tree should have 38 edges:

|  |  |  |  |
| --- | --- | --- | --- |
| 7 → 1 | 7 → 10 | **7 → 12** | 12 → 11 |
| 12 → 14 | **7 → 18** | 18 → 15 | 18 → 21 |
| **7 → 26** | 26 → 22 | 26 → 30 | **7 → 35** |
| 35 → 31 | 35 → 39 | **7→ 41** | 41 → 40 |
| 41 → 43 | **7 → 48** | 48 → 44 | 48 → 53 |
| **7 → 55** | 55 → 54 | 55 → 56 | **7 → 59** |
| 59 → 57 | 59 → 62 | **7 → 67** | 67 → 63 |
| 67 → 71 | **7 → 76** | 76 → 72 | 76 → 80 |
| **7 → 85** | 85 → 81 | 85 → 89 | **7 → 92** |
| 92 → 90 | 92 → 99 |  |  |

where the edges in bold represent jump edges. You should make sure the correct number of edges are present and that each individual edge has the correct start and stop residue index. You can write your unit test however you would like, but you might find it useful to use the FoldTree::get\_residue\_edge function to find the edge for a particular residue, and then get the “start” and “stop” indices from the Edge that’s returned (see src/core/kinematics/Edge.hh). You can also print a fold tree to std::cout to help you debug your function while you’re writing it. When your unit test is working properly, commit your code (run “git status” and “git add” first).

Write a final unit test where you compute the secondary structure from “test\_in.pdb” (a Pose from test\_in.pdb can be constructed by calling the function that lives in test/util/pose\_funcs.hh named create\_test\_in\_pdb\_pose) using the DSSP code that lives in src/core/scoring/dssp. With that secondary-structure string, call your function and set the returned foldtree to be the fold tree for the test\_in pose. If when you set the fold tree for your pose, your unit test crashes, that is not good. Try calling “check\_fold\_tree” on your fold tree to make sure that your fold tree is valid. Commit your code (run “git status” and “git add” first) and push your branch to github.”

**5) Move your functions to the protocols library**

Now that your functions are working, you will move them to the protocols library. Create a new folder src/protocols/bootcamp and within that folder create two files fold\_tree\_from\_ss.hh and fold\_tree\_from\_ss.cc. Put the declaration to your functions in the .hh file and the implementations in the .cc file. Make sure you include the copyright header at the top of the file and that you put the right namespace declarations around your functions

namespace protocols {

namespace bootcamp {

…

}

}

Add fold\_tree\_from\_ss to the protocols.7.src.settings file and compile with SCons. Then go back to your unit test file, add a new #include to <protocols/bootcamp/foold\_tree\_from\_ss.hh> and appropriately namespace scope your calls to your two functions (e.g. calls to identify\_secondary\_structure\_spans should become calls to protocols::bootcamp::identify\_secondary\_structure\_spans). Make sure your unit tests still compile and run correctly. Add your new files to git, commit your modifications, and push your branch to github.

**6) Use your new FoldTree**

Add calls to DSSP and to your new function to create a FoldTree for the input Pose in your bootcamp.cc application. This should be done outside of your application’s main loop. Before you can fully use this new fold tree, you will need to add some code to ensure that the geometry at the cutpoints does not degrade as you perturb your structure. Adding cutpoints means that the ideal bond-length and bond-angle geometry is no longer being enforced by the kinematics of the system. To make up for this, you must make two adjustments: one to Pose, and one to the ScoreFunction.

You will need to enable a new score term in your ScoreFunction, the linear\_chainbreak term, which will penalize the introduction of bad geometries across the cutpoints. Set the weight on this term to 1. Look at ScoreFunction.hh to see if there’s some way to *set* a *weight*. (Make sure you set the weight in your ScoreFunction \*before\* you create your MonteCarlo object: remember, the MonteCarlo owns its ScoreFunction -- it creates a deep copy -- so that it can maintain the integrity of its data: namely that the energy of the lowest-score Pose is less than or equal to the last-accepted Pose. If MonteCarlo object made a shallow copy of the ScoreFunction it was given -- that is, if it kept a pointer to the input ScoreFunction -- then someone could go and change the weight on the ScoreFunction it held so that the last-accepted Pose was lower in energy than the lowest-seen according to the new set of weights. To paraphrase the vice president Joe Biden: data integrity is a big f’n deal.) The linear-chainbreak term requires special ResidueTypes to be present at the cutpoints; there are upper and lower-cutpoint variants of each basic amino acid type. These variants introduce extra *virtual atoms* to the system -- atoms that do not contribute to the score, but that are useful in describing the kinematics of the system. These virtual atoms are used by the linear chainbreak term to make sure that the bond angles and the bond dihedrals stay within their acceptable ranges.

To add the cutpoint variants to your Pose, you can use the utility function correctly\_add\_cutpoint\_variants() that is declared in core/pose/util.hh. (This function violates one of our coding conventions in a pretty significant way: it takes as input a raw pointer instead of a reference or an owning pointer. It ought to be fixed. You shouldn’t look at this function as a guide for how to develop code in Rosetta.)

Once you have finished adding the cutpoint variants to your Pose, and you have added the chainbreak term, re-run your program and watch the trajectory using the PyMol viewer. You can see that the perturbations are now much more local -- they do not propagate all the way through the chain. How do your acceptance rates now compare to before you made this change? How do your energies compare?