**Lab 1: Your first Rosetta protocol**

In this lab, you will be creating a new application for optimizing the side chains and backbone of a structure, and you will watch as this application modifies a structure in real time using PyMol.

This is the first of four evening labs where the code you develop in this lab will be your starting point for the next evening lab. You will have the rest of the afternoon to work on this lab, and then a session after dinner, from 8 until 10. It’s entirely likely that this will not be enough time. Don’t worry! You will keep working on this lab when we get to

**1. Create a new branch off master in git.**

You will be developing your application in the Rosetta source directory alongside the existing code. You will want to be able to save your progress and to push your progress to Github to make sure it isn’t lost, just as you will when you develop code after boot camp ends, and you will want to do this without effecting the rest of the Rosetta community’s ability to develop their own code. The way to handle this is to create your own branch off of master, and to develop your code in that branch. You will be able to commit your code to this branch, and you will be able to push your branch to Github. All of this activity will be completely roped off from master; you don’t have to worry about your changes effecting anyone else. For this reason, you may commit freely to your branch, without worrying about whether your code compiles, or runs properly, or adheres to the coding conventions.

You are also developing in a way that ensures you will be able in the future to bring your code back into master in the future -- that is, to merge. Git has been designed to make merging as easy as possible, which is a major advantage of git over other version control systems, such as CVS or SVN. The code you develop in these evening labs is mostly pedagogical, so it is unlikely that you will be merging it back into master, but by working in a branch, that possibility remains open.

To create a branch, you must first choose an appropriate branchpoint. You should check out the “master” branch, which you should have already compiled before coming to boot camp.

> cd /path/to/Rosetta/main

> git checkout master

(The lines in the fixed-with font, as seen above, are either code or commands that you will use to type into the command line. If the line begins with a greater-than symbol (>) then that symbol is meant to represent the command prompt. Do not type that symbol into the command line. ALSO: google docs replaces double quotes with funky “smart” quotes. If you paste smart quotes into a source file and send them through the compiler, the compiler won’t recognize them and will give you an error. So if you see an error message about a stray symbol in your program, then you should take a very close look at the quote symbols.)

The git checkout command will checkout the master branch if you are not already in it. To check which branch you are already in, you can use the git branch command. This will list a set of all the local branches you have, and indicate your currently checked-out branch with an asterix.

> git branch

aleaverfay/aleaverfay/fa\_elec\_in\_etable

aleaverfay/cenrot\_speedup

aleaverfay/dom\_assemb\_jd2

aleaverfay/h2o\_packer

\* aleaverfay/mt\_msd

aleaverfay/profile\_sho

aleaverfay/resource\_manager\_doc\_commit

aleaverfay/resource\_manager\_documentation

aleaverfay/vectorize\_lj

aleaverfay/znhash\_w\_backups

centroid\_rotamer

glemmon/residue\_refactor

loops\_refactor\_team/infrastructure\_new

luki-mbi/ast\_transform

luki-mbi/rosetta\_scripts

master

mcclurej/mt\_msd

momeara/talaris2013

sacombs/backtrack

In this case, I have currently checked out the aleaverfay/mt\_msd branch, and so I will need to run git checkout master before I create a new branch. If the asterix were next to master instead, then I could skip the git checkout master execution.

With the master branch checked out, now you will use the Rosetta-community provided command personal-tracked-branch. This is an alias for a two-part command to create a branch with your github username out front (e.g. my github username is “aleaverfay,” so all of my branches are named with aleaverfay/ out front) while making sure that your branch will be remotely tracked by github, and then to check out that branch. You will create a branch named “bootcamp\_lab2.”

> git personal-tracked-branch bootcamp\_lab2

which will create the branch yourusername/bootcamp\_lab2. Now when you execute the git branch command, you will see the asterix next to this branch instead of next to master. Tomorrow, in lab 4, you will create a branch off of this branch instead of from master.

**2. Creating a new application**

New applications are added to the Rosetta source code in subdirectories of the source/src/apps/pilot directory. cd into this directory and create a new subdirectory with your name (at some point, we started naming pilot-app directories after the developer who created them, and the convention has stuck).

> cd source/src/apps/pilot

> mkdir yourname

> cd yourname

Within this directory, you will create a new “hello world” application and name it “bootcamp.cc”. I prefer emacs. Some people prefer vi. Some people enjoy XCode, Eclipse, or NetBeans.

// -\*- mode:c++;tab-width:2;indent-tabs-mode:t;show-trailing-whitespace:t;rm-trailing-spaces:t -\*-  
// vi: set ts=2 noet:  
//  
// (c) Copyright Rosetta Commons Member Institutions.  
// (c) This file is part of the Rosetta software suite and is made available under license.  
// (c) The Rosetta software is developed by the contributing members of the Rosetta Commons.  
// (c) For more information, see http://www.rosettacommons.org. Questions about this can be  
// (c) addressed to University of Washington UW TechTransfer, email: license@u.washington.edu.

#include <iostream>

int main() {

std::cout << "Hello World!" << std::endl;

return 0;

}

Note the Rosetta copyright header at the top. This copyright header needs to be placed at the top of all the files you write in Rosetta. It can be copy and pasted from the coding conventions page on the Wiki (Note in most IDEs you can set it up to add this header automatically when you create new hh and cc files):

https://wiki.rosettacommons.org/index.php/Coding\_Convention\_and\_Examples#Copyright\_Header

You can also take another application (say, apps/public/design/fixbb.cc), copy it to your desired filename, then just delete out all of its code, leaving the header shell and a main() function empty except for devel::init and the exception handler.

Your application doesn’t do much yet, but it will. You will next add this new application to the list of executables to compile and link against the Rosetta libraries. Compilation in Rosetta is managed by the SCons build management system, which is written in Python. The internal DocWiki has more info on the build system if you want to look at that later. You will need to add your executable to the existing list of executables that lives in the Rosetta/main/source/src/pilot\_apps.src.settings.all file. Open this file up. You will see that it contains a list of directories, sorted alphabetically, and within each directory a list of file names (without the .cc endings). Each one of these files will be compiled and linked into an executable by SCons.

Find the appropriate place within this file and add your directory and your new executable, following the syntax that the other entries in this file use. Beware: Python is sensitive to indentation, so make sure you align your entry using spaces with the one above it. (For those familiar with python, this whole file represents the definition of a dictionary named “sources”, and each directory represents one key in the dictionary, where the colon separates the key from the value; the value is a list.)

Now compile your program using SCons.

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

> ./scons.py bootcamp

to produce the bootcamp.default.<os><compiler>debug application (<os> will either be macos or linux; the compiler will be either “gcc” or “clang”; from here forward, I’ll write macosgccdebug, but your application may have a different name). In this command line, you’re telling SCons that the “target” that it should compile is your bootcamp.cc executable. Now run your application:

> bin/bootcamp.default.macosgccdebug

You should see that it prints “Hello World!” to the screen.

Your SCons command, if it executed without error, should have produced two commands that it echoed to the screen. The first of them compiled your application, the second of them linked it. Copy these two commands into a new text file, build\_bootcamp\_app\_debug.sh, for safe keeping. You can reissue these two command by running

> source build\_bootcamp\_app\_debug.sh

and compilation will go much faster. Why? Well, SCons has to make sure that you haven’t modified any of the libraries upon which your application depends. If you had modified src/core/scoring/ScoreTypes.hh for example, SCons would have needed to recompile a lot of source files and re-link a lot of libraries. So to make sure that everything is up to date, SCons has to go through and check every source file to make sure it hasn’t changed. This takes a lot of time! If you know you haven’t modified any source files in the Rosetta libraries, then you can save a lot of time by just executing the compilation and linking commands.

Throughout this week, you will create many of these shell scripts; if you find yourself running “scons” a second time in a row, you should stop and create one of the build-command scripts so that the third time you need to run scons, you can run the script instead.

You should also edit your build\_bootcamp\_app\_debug.sh file so that the two commands are on a single line and separated by an “&&”, e.g. if the file had been

command1

command2

then you should change it to be

command1 && command2

The “&&” says to your shell “only execute the second command if the first command exited without error” and will save you the confusion of a linker-error message if the compilation should fail. If you want to spruce up your command a bit, then you could have it output progress messages to the screen (beware of smart quotes below):

echo "compiling" && command1 && echo "linking" && command2

Add your application to your git repository:

> git add src/apps/pilot/yourname/bootcamp.cc

This says to git that it should begin tracking changes to this file. Next tell git that you want it to save the changes you made to src/pilot\_apps.src.settings.all; this also requires using the git add command:

> git add src/pilot\_apps.src.settings.all

Now git has “staged” two files to be committed. Staging commits is a great way to logically bundle a set of changes into a single commit. It’s possible to commit every file you’ve changed, but its a better practice to tell git exactly which files that have changed ought to be commited. (Below, when the instructions say “commit your code”, make sure you’ve staged code to be committed first! Always run a “git status” before you run “git commit.”) Now you should make your commit.

> git commit

which will pull up your text editor. Describe what you have done in a sentence or two. This “log file” message will be read by hundreds of other developers, so you’ll want to accurately represent what you’ve done in this revision. For this commit, you’ll want to say something along the lines of “Creating a new, as of yet empty pilot\_app, yourname/bootcamp, and adding it to the pilot\_apps.src.settings.all file.”

Inspect your commit using the git log command

> git log --graph

Does your commit log message say what you want it to say? If not you may edit the commit log message using this command

> git commit --amend

This will pop your text editor back up and let you edit the commit log without having to re-specify which files you want to commit. You will want to get in the habit of checking your commit messages before you push your commits to github or to anywhere else. It’s only possible to edit a commit message before you push your commit somewhere else.

Now push your commit to github

> git push origin yourusername/bootcamp\_lab2

Congrats! You’re getting familiar with the safety net that version control offers. Version control is an incredibly important part of developing software. If you’re not using version control, you are wasting your own time. I can guarantee you’ll encounter a scenario where you’ve made too many modifications and want to go back to some previous state -- that’s just part of developing code. Everyone, myself included, makes mistakes. If you’re not using version control, then getting back to your previous state is always hard and is sometimes impossible. With git, it’s as easy as using the git checkout <revisionid> command.

**3. Edit your bootcamp executable to load in a Pose from a PDB file**

Next, you will use the options system to provide you with a file name that’s provided on the command line for a PDB that you should load in, and load in that Pose using the import\_pose function. There are several steps here.

A) Add #includes to the global option system object.

The option system “option” global variable is declared in src/basic/options/option.hh. #include this file at the top of your executable using angle brackets (< and >) around the file name -- but excluding the src/ portion of the file path. You will also need to #include an “Option Keys” file that contains the option you’ll be using. In particular, you will need

to #include <basic/options/keys/in.OptionKeys.gen.hh>. Within this file is the “s” option, which is the option usually used to provide Rosetta with the name of the PDB file to use.

Edit the declaration of int main() so that it takes two arguments

int

main( int argc, char \*\* argv ) {...

and add the following lines to your existing bootcamp.cc file:

devel::init( argc, argv );

utility::vector1< std::string > filenames = basic::options::option[ basic::options::OptionKeys::in::file::s ]();

if ( filenames.size() > 0 ) {

std::cout << "You entered: " << filenames[ 1 ] << " as the PDB file to be read" << std::endl;

} else {

std::cout << "You didn’t provide a PDB file with the -in::file::s option" << std::endl;

return 1;

}

The call to devel::init( argc, argv ) initializes the options system -- if you try to read from the options system without having first called devel::init, then your application will exit with an error message. You will need to #include <devel/init.hh> in order to call this function.

The -s flag is a “vector” flag, so that you can pass multiple files to a Rosetta executable. This is often handy. For this lab, however, you will only be using the first file that’s passed in on the command line; the one in position “1” of the filenames vector. Recompile your application. It will fail with an error message. The code I’ve given you is ok, but something (not in the code block above) is missing. Can you figure out what?

Once your program compiles, run it. Go ahead and download the 1UBQ pdb file for ubiquitin, which you will be using in this lab.

> wget [http://www.rcsb.org/pdb/files/1UBQ.pdb.gz](https://www.rcsb.org/pdb/files/1UBQ.pdb.gz)

or

> curl -OL [http://www.rcsb.org/pdb/files/1UBQ.pdb.gz](https://www.rcsb.org/pdb/files/1UBQ.pdb.gz) -o 1UBQ.pdb.gz

so that you can run your application with the following command:

> bin/bootcamp.default.macosgccdebug -in:file:s 1UBQ.pdb.gz

You should see it print:

Hello World!

You have entered 1UBQ.pdb.gz as the PDB file to be read

Commit your changes to bootcamp.cc and push them to github.

Now you will need to construct a Pose object from the pdb file. To do so, use the function “pose\_from\_file” and pass it the first entry in the filenames vector. Use ack or ack-grep to locate the pose\_from\_pdb function -- it lives somewhere in src/core. This function has several flavors; use the one that returns a PoseOP.

You will eventually write code that looks like this:

core::pose::PoseOP mypose = core::import\_pose::pose\_from\_file( filenames[1] );

This will initialize an *owning pointer* to a Pose. PoseOP is a “typedef” to utility::pointer::shared\_ptr< core::pose::Pose >, but it used to be typedeffed to utility::pointer::owning\_ptr< core::pose::Pose > and the name “owning pointer” hasn’t gone away. You can find this typedef in src/core/pose/Pose.fwd.hh. OP typedefs are almost always declared in .fwd.hh files. Note that objects held in owning pointers should not be deleted directly; owning pointers are smart pointers that will automatically delete the objects they point at once they can tell that no other object points to them. You do not have to worry about leaking memory when you use owning pointers. You will need to #include <utility/pointer/owning\_ptr.hh> and <core/pose/Pose.hh> as well as the file that contains the pose\_from\_pdb function in order to get this code to compile.

Compile your code and run it. You will need to provide two new command line arguments: -database /path/to/Rosetta/main/database and -ignore\_unrecognized\_res. 1UBQ.pdb has water residues that Rosetta, by default, will choke on. The water being off by default is a feature, not a bug. <https://carbon.structbio.vanderbilt.edu/mantisbt/view.php?id=326> (It is possible to have Rosetta read in waters, but we’re not going to be doing that in this lab). By giving your app this flag, it will skip passed the water residues without exiting with an error message. Your application doesn’t do much more than it did last time, but you should see new messages whiz down the screen saying that files are being opened and initialized. Commit your code using git add/git commit and push it to github.

**4) Create a ScoreFunction object, score your pose, and print its score.**

Initialize a ScoreFunction object by calling get\_score\_function(). This function lives somewhere in src/core/scoring. Use ack, ack-grep, or IDE search to find it. get\_score\_function returns a ScoreFunctionOP, so save the returned value in a new ScoreFunctionOP variable named sfxn. Once you have this variable initialized, you can score your pose using the following code:

core::Real score = ( \*sfxn )( mypose );

(Oops! The code above does not work. Take a look at ScoreFunction’s declaration (where is it declared? Did you remember to #include that file?) and find the “operator ()” function -- it should be near line 226 of the header file. What kind of argument does it expect? What kind of argument does the code above give it? (Look at where mypose was declared.) What do you have to do to make the above code work?)

Add code to print your score to the screen. Recompile your code, and run it. You should see that 1UBQ has a score of 100.6. Stage your changed files, commit your code, and push your commits to github.

**5) Create a monte carlo protocol that, in a loop, perturbs the phi and psi values of a single residue, and then, using a MonteCaro object, either accepts or rejects the perturbed values based on the change to the score of the pose.**

You’ll be working with several new pieces of functionality in this part.

a) a random number generator

b) several methods of class Pose

- total\_residue()

- phi( core::Size seqpos )

- psi( core::Size seqpos )

- set\_phi( core::Size seqpos, core::Real new\_phi\_value )

- set\_psi( core::Size seqpos, core::Real new\_psi\_value )

c) the MonteCarlo class, and in particular, its “boltzmann” function.

You will write a loop and inside this loop you will perturb the phi/psi values and invoke the MonteCarlo::boltzmann method.

**a) Random number generators**

You will be perturbing phi and psi of a single residue by a random amount in each iteration through a loop. To get this random amount, you will draw two random numbers from a Gaussian distribution. The random number generation in Rosetta is centralized so that with a constant random-number “seed,” Rosetta will take the same trajectory. This is very useful for testing (to make sure you get the same answer on repeated executions of the code) and for debugging (so you can repeated execute the code that leads to the hard-to-track-down error). You can use the “gaussian” function that lives somewhere in the src/numeric directory; use ack or IDE search to track it down. The gaussian function returns a value between negative infinity and positive infinity drawn from a standard normal. You will also need to pick a random residue to perturb; you can use the “uniform” function that also lives somewhere in src/numeric to return a random number uniformly distributed in the range [0..1). If you have such a randomly distributed number stored in the variable uniform\_random\_number, and if there are N residues in your Pose, then the expression

static\_cast< core::Size > ( uniform\_random\_number \* N + 1 )

will give you an (unsigned) integer between 1 and N.

**b) Pose**

The total\_residue() method returns the total number of residues in the Pose. You will need this value to pick a residue at random to perturb.

The phi( core::Size seqpos ) method returns the currently assigned phi value for a particular residue in degrees. Note that it is up to you to make sure that the given residue is an amino acid, otherwise, phi would not make sense.

The psi( core::Size seqpos ) method returns the currently assigned psi value for a particular residue in degrees.

The set\_phi and set\_psi values will actually change the currently assigned phi and psi values to a new value given in degrees

Your code will eventually look like this:

core::Size randres = … code here to pick the index of a random residue in the Pose

core::Real pert1 = … code here to get a random number

core::Real pert2 = … code here to get another random number

core::Real orig\_phi = mypose.phi( randres );

core::Real orig\_psi = mypose.psi( randres );

mypose.set\_phi( randres, orig\_phi + pert1 );

mypose.set\_psi( randres, orig\_psi + pert2 );

**c) MonteCarlo**

The MonteCarlo class lives somewhere in src/protocols; find it. You should create a MonteCarlo object outside of your loop, and you will need to pass it a Pose and a ScoreFunction in its constructor. Properly constructed, the MonteCarlo object is ready to use. Its boltzmann function takes a Pose reference, scores the Pose (using the ScoreFunction given to it in its constructor) and compares the new score against the score of the Pose that it holds internally. If the new score is lower than the old score, then the MonteCarlo object “accepts” the new conformation of the Pose by copying the input pose into its internally-held pose. If it’s higher than the old score, then it picks a random number and accepts the new conformation with a certain probability given by the Metropolis criterion. If it rejects the new conformation, then it copies the internally-held pose into the input pose -- it effectively overwrites any change that’s been made to the input pose.

Your code will have this structure:

// 1. Initialize a pose, a score function object, and then a MonteCarloObject

// 2. Begin loop

// 3. perturb the phi/psi values for your Pose

// 3. call MonteCarlo object’s boltzmann method, passing it your Pose.

Recompile your code, and run it. Commit it and push your branch to github.

**6. Connect your application to PyMol using the PyMolObserver**.

The PyMolObserver lives in protocols/moves/PyMolMover.hh. Before entering the perturb/boltzmann loop, you will call the function

protocols::moves::PyMolObserverOP the\_observer = protocols::moves::AddPyMolObserver( \*mypose, true, 0 );

which will create a PyMolObserver and attach it to the Pose. It will then start broadcasting changes to your Pose to any listening PyMol session. (Make sure you capture the observer returned by AddPyMolObserver, since it’s this observer that’s going to broadcast coordinates to your pymol session. If you don’t save this observer in a local variable, the owning pointer will delete it as soon as the AddPyMolObserver function exits.)

Recompile your code, this time in release mode

> ./scons.py bootcamp mode=release

which will give you the executable

bin/bootcamp.default.macosgccrelease

that runs about 10x faster than the debug build does. Once it is compiled, open up a PyMol session and within your pymol session run the script that’s described in the online documentation here:

<https://www.rosettacommons.org/docs/wiki/rosetta-basics/graphics-and-guis#Pymol-Mover>

You will have to authenticate using your github account in order to access this documentation.

Once you have run this script, you’re ready to run your application. When you run it this time, you will see that your PyMol session starts recording “states” showing that the Pose is changing. If you want to rerun the application, you might want to reinitialize your pymol session.

**7. Add packing and minimization.**

Rosetta offers two modules that are useful for almost any protocol. It offers a side-chain optimization module, “the packer,” which uses discrete side-chain conformations called rotamers to explore the space of all possible side-chain conformations on a fixed backbone. It also offers a quasi-Newton minimization routine that works in internal degrees of freedom so that we can optimize structures without modifying bond lengths or angles, but only their dihedral angles. In this final part of tonight’s exercise, you will add packing and minimization calls to your application after you have perturbed the phi and psi values, but before you have called the MonteCarlo::boltzmann function.

You will spend more time learning about both the packer and the minimizer in the remainder of boot camp, so we have written the code you will need to add; you will simply have to hunt down the appropriate set of files that have to be #included to get the code to compile.

**a. Packing**

You will create a PackerTask object immediately before you use it. PackerTask objects are one-time use objects; it’s important to create the PackerTask object immediately before you use it. Once you’ve created a PackerTask, you will modify it so that it only repacks (without redesigning) your Pose. Then you will call the pack\_rotamers function, passing it your pose, your score function, and your packer task.

core::pack::task::PackerTaskOP repack\_task =

core::pack::task::TaskFactory::create\_packer\_task( \*mypose );

repack\_task->restrict\_to\_repacking();

core::pack::pack\_rotamers( \*mypose, \*sfxn, repack\_task );

\*\*NOTE: This is actually a nonstandard way to call the packer within your code. We are doing it the “hard way” once for learning purposes\*\*

**b. Minimization**

You will create several classes to get minimization to work properly. These three classes should be declared outside of your main loop and, unlike the PackerTask, can be used repeatedly inside of it.

i) a MoveMap, which describes the degrees of freedom that should be opened up during minimization. In the example code, both the backbone dihedral angle degrees of freedom, and the sidechain dihedral angle degrees of freedom have been opened up. We will talk about the MoveMap on Thursday.

core::kinematics::MoveMap mm;

mm.set\_bb( true );

mm.set\_chi( true );

ii) a MinimizerOptions object, which gives the parameters needed to describe the minimization process

core::optimization::MinimizerOptions min\_opts( "lbfgs\_armijo\_atol", 0.01, true );

iii) an AtomTreeMinimizer object, which will actually run minimization.

core::optimization::AtomTreeMinimizer atm;

You will invoke minimization after your call to pack\_rotamers, with this line

atm.run( \*mypose, mm, \*sfxn, min\_opts );

\*\*NOTE: This is actually a nonstandard way to call minimization within your code. We are doing it the “hard way” once for learning purposes\*\*

It turns out that the above code will run very slowly; it’s not that the minimizer runs slowly (though it’s not terribly fast) but that the minimizer is updating your pose and that update sends a new structure to PyMol. Sending the structure is quite slow. You can make this protocol run much more quickly by making a copy of mypose, minimizing the copy, and then copying the copy into mypose.

core::pose::Pose copy\_pose = \*mypose;

atm.run( \*copy\_pose, mm, \*sfxn, min\_opts );

\*mypose = copy\_pose;

In fact, you can declare copy\_pose outside of the main loop to give yourself an even larger speedup, since creating and destroying a Pose can be quite time consuming.

core::pose::Pose copy\_pose;

for ( ... ) {

...

copy\_pose = \*mypose;

atm.run( \*copy\_pose, mm, \*sfxn, min\_opts );

\*mypose = copy\_pose;

...

}

Compile your code in release mode and run it again. Commit it, and push your branch to github. You will begin lab 4 from this point tomorrow.