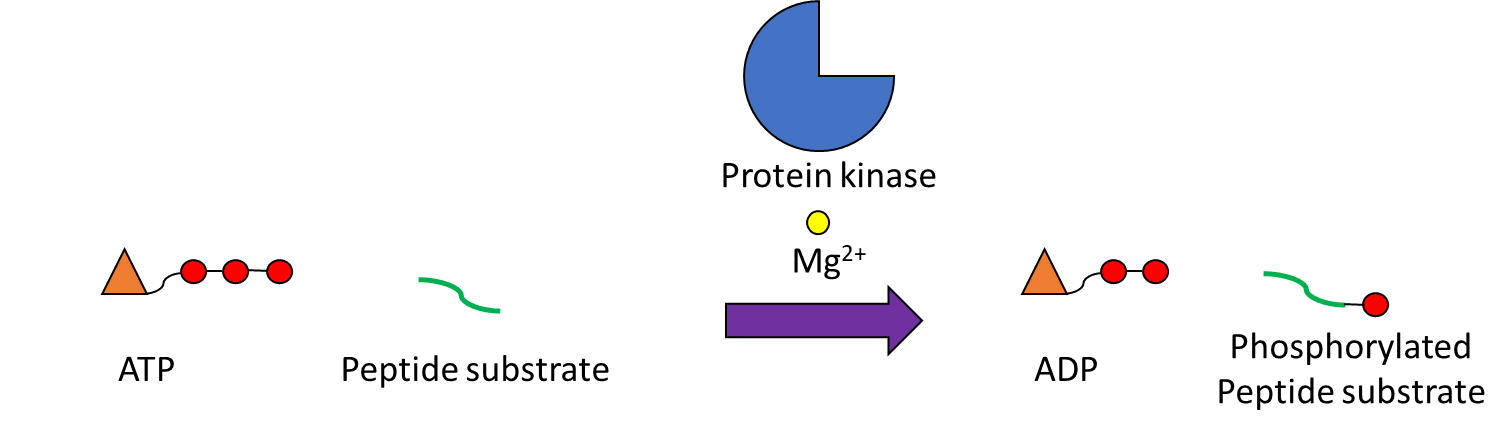
**Tutorial #5: Alchemical structure-based design simulation of type I and II kinase inhibitors**

Background

Protein kinases are signaling enzymes that catalyze the transfer of phosphate from ATP to protein/peptide substrate. A typical reaction is of the scheme ATP + Peptide -> ADP + Peptide-P:



The phosphorylated substrate is frequently an “activated” form of the downstream protein that propagates the signal that activated the kinase. Kinases may be activated by extracellular binding of growth factors to promote cell growth and division. Thus, oncogenic kinases promote carcinogenesis through aberrant signaling activity.

A common strategy to treat cancers caused by overactive kinases is to prevent binding of ATP and subsequent substrate phosphorylation with ATP-competitive small molecule inhibitors. In this tutorial, we will use a new multifaceted Rosetta mover called BCLFragmentMutateMover to perform structure-based design of small molecule inhibitors of Abl kinase. BCLFragmentMutateMover is an implementation of the FragmentMutateInterface framework in the BCL and at the time of writing represents the primary functional connection at the BCL-Rosetta interface.

An important note about the intended functionality of the BCLFragmentMutateMover is that it was built to try and retain information about the binding pose of the ligand as the ligand is chemically perturbed. The idea here is that we want to avoid a global redocking of the ligand if we can because docking is expensive and not without error. Therefore, the BCL mutates attempt to preserve coordinate information as the design process is ongoing, and Rosetta refines the protein-ligand interaction space as needed with help from other algorithms (e.g., FastRelax). If you *do* want to redock after each round of design, this is perfectly fine and it can easily be done using standard RosettaLigand after chemical structure perturbation.

Part 1: Preparing a small molecule scaffold for design

In contrast to the reaction-based design approach in Tutorial #4, most\* applications of the BCLFragmentMutateMover require a starting scaffold. One way in which a pipeline may proceed is to perform *de novo* reaction-based design as in Tutorial #4 and a second round of focused design on the most promising hits using the BCLFragmentMutateMover. For the purposes of this tutorial, we will start with a kinase inhibitor scaffold designed by (CITATION). The scaffold consists of a amino-substituted pyridopyrimidine heterocycle connected to a benzene ring.

This scaffold is interesting because it can support derivatization into either type I or type II inhibitors. Note that it contains a canonical hinge-binding motif, which is a set of 1-2 hydrogen bonds that mimic the hydrogen bond made by ATP in the binding pocket.

<hinge figure>

The first step in preparing our scaffold is to generate a Rosetta params file from a valid MDL SD file. Recall from first tutorial that we can check to make sure our scaffold meets some basic criteria before continuing with it.

<molecule:Filter -input\_filenames scaffold.sdf -defined\_atom\_types -3d -add\_h>

This has already been done for you, so we will start with the prepared input file LIG.sdf in Tutorial\_5/inputs/ligands/

To generate the params file, run the following command:

<molfile\_to\_params>

You should see the following message:

<output + warning>

A few notes:

1. This is the only time during the drug design process that we will need to make a params file. Throughout the remainder of this exercise, all newly designed molecules will be parameterized automatically internally. We need this params file strictly to load the small molecule into Rosetta.

2. For small molecule docking, it may be beneficial to explicitly declare aromatic bonds instead of the kukulized (alternating single/double bonds) structure. Recall that this can be achieved in most BCL applications by simply passing the -explicit\_aromaticity flag. This will cause aromatic substructures in the output SDF to have bond orders of 4 (indicating aromatic).

3. In our case here, it is important that you **use the kekulized structure.** This is (currently) necessary in order for Rosetta to communicate bond orders reliably to the BCL data structures that will handle design. Do not worry – the BCL will recompute aromaticity every time the molecule is modified and assign the appropriate atom types and bond orders to the molecules when it is converted into a Rosetta object again.

The second step is not strictly mandatory; however, it is important if you want to be able to specify the parts of the scaffold that can be modified during design at the atom level (i.e., specify mutable atoms explicitly). The second step is to convert our params file into a molecule file (e.g., SDF or PDB).

<restype converter>

Why did we do this, you ask? **Because Rosetta does not preserve input atom indices. The atom indices of the initial input file, e.g., LIG.sdf, differ from the atom indices of the molecule after it is read from a params file**.

If you visualize the two molecules in PyMOL you can see the difference in atom indexing (from 0) by clicking <label → atoms → rank>.

<figure showing atom indices differ>

If you were to tell Rosetta “I want derivatize my scaffold at index 8” and you did not know that Rosetta renumbered the atoms of the original molecule, you would be very confused about why your <blank> was changing instead of your <blank>.

Now we are ready to start using the BCL-Rosetta small molecule design mover!

Part 2: Intro to BCLFragmentMutateMover

During this section it may be helpful to open the slides from the talk so that you can reference the standalone BCL commands in molecule:Mutate.

Open the SimpleTest.intro.sh script with vim or gedit or any other editor of your choice. This is a wrapper shell script to call the rosetta\_scripts.bcl.<os>release application. The arguments are, in order, the RosettaScripts XML file, receptor PDB filename, ligand PDB filename ligand params filename, and an output prefix.

The first XML file contains the following block to initialize the BCLFragmentMutateMover:

<BCLFragmentMutateMover name="help"

ligand\_chain="X"

object\_data\_label="Help"

/>

Run this script with the following command-line

<command>bash ../scripts/SimpleTest.intro.sh ../scripts/SimpleTest.intro\_1.xml ../inputs/receptor/Abl\_Inactive\_A.pdb ../inputs/ligands/LIG\_0001.fa.pdb ../inputs/ligands/LIG.fa.params HELP\_0\_

You will notice that the program crashes. This is expected because the object\_data\_label “Help” is invalid.

===> Failed to read bcl::util::Implementation<bcl::chemistry::FragmentMutateInterface>, error was: Given implementation "Help" is unknown, here are the allowed values {AddBond, AddMedChem, Alchemy, Connect, Cyclize, ExtendWithLinker, Fluorinate, Halogenate, React, RemoveAtom, RemoveBond, RingSwap}

^^^ In serializer: Help

Failed at pre-read hook, exiting

Notice you obtain a similar error message if you try to pass “Help” as a mutate option in the standalone BCL app molecule:Mutate

bcl.exe molecule:Mutate -implementation Help

These messages are telling us that the allowed mutate implementations are

<list>

This is helpful if you are not familiar with the available options.

But now what? What if you think that you want e.g., Alchemy, but you are not positive what it does or how to use it?

The next intro XML script contains the block:

<BCLFragmentMutateMover name="help"

ligand\_chain="X"

object\_data\_label="Alchemy(help)"

/>

This will also stop early. It will display a description of the mutate:

===> Requested help: Transforms atom/element types inside of a molecule

Underneath that message, it will also display all of the options available to Alchemy (too long to list here).

This information can also be obtained with BCL standalone

bcl.exe molecule:Mutate -implementation “Alchemy(help)”

Part 3: Type I inhibitor design

Type I inhibitors sit directly in the ATP binding pocket. Our scaffold can still mimic additional ATP-protein interactions to increase binding affinity.

<superimpose our scaffold and the CITATION type I TKI with ATP>

In addition to the hinge binding hydrogen bonds, type I inhibitors often form hydrophobic-aromatic stacking interactions with the nucleotide-binding glycine-rich loop. Therefore, one rational design decision is to grow another aromatic ring system from our amino group. There are a lot of aromatic rings in the world and it’s not obvious what the optimal choice will be to improve target engagement. So, let’s sample a bunch and see which ones have the best interaction energies in Rosetta.

The first implementation of FragmentMutateInterface that we will initialize our BCLFragmentMutateMover mover with is ExtendWithLinker.

<BCLFragmentMutateMover name="extend\_with\_linker"

ligand\_chain="X"

object\_data\_label="ExtendWithLinker(

ov\_reverse=True,ov\_shuffle\_h=False, ring\_library=%%rotamer\_library%%/ring\_libraries/drug\_ring\_database.simple.aro.small.sdf.gz,

extend\_within\_prob=0.0,direct\_link\_prob=100000,

druglikeness\_type=None,

mutable\_atoms=18)"

/>

We define our mover with the name “extend\_with\_linker” (I frequently use “ewl” for short). The ligand is identified as chain X. Now we need to initialize the BCL object within Rosetta via the object\_data\_label option. For reference, feel free to look at the Tutorial 5 intro talk PowerPoint slides.

ExtendWithLinker grows molecules using an assortment of linkers either internally or off of a terminal heavy atom. Like all of the FragmentMutateInterface-derived objects, if left unspecified ExtendWithLinker is given free reign to make random changes to your molecule (i.e., it acts on any atom in the molecule, it randomly selects rings to either append or use as bridges, it does not filter for druglikeness, and the relative probabilities of extending with rings, single elements, or chains are approximately equal).

Here, we restrain the behavior of ExtendWithLinker to model our rational design decision. We set the mutable\_atoms index to 18, corresponding to the amino nitrogen atom. The mutable and fixed atoms/elements/fragments options are common for all of these mutates. The ring library is taken from one of the default libraries provided with the BCL in the Rosetta/main/source/external/bcl/rotamer\_library/ring\_libraries/ directory. The path to the rotamer library is set with a variable defined in the wrapper shell script. We set the relative probability of a direct link arbitrarily high (if you read the help menu, it will indicate that the probabilities for the different linking strategies are relative to one another) to ensure that our chosen ring is directly bonded to the amino nitrogen.

There are several other options here that are common to other mutates. The “ov\_reverse” and “ov\_shuffle\_h” flags are options related to “opening valences”. We cannot form a new covalent bond between the amino nitrogen and an atom in the aromatic ring randomly selected from the user-specified pool of rings if we do not remove an existing bond. If we just force a bond willy-nilly, then we are frequently going to encounter invalid atom types. To avoid this issue, we remove hydrogen atoms to open a valence prior to making a bond.

But there are two hydrogen atoms. Which one do we remove? By default, we randomly select one to remove. Chemically, it does not matter which one we remove because they are symmetric. However, because BCLFragmentMutateMover attempts to preserve binding mode information, the hydrogen atom we remove actually has a substantial impact on the pose.

<figure showing a simple benzene extended from either of two hydrogen atoms using the BCL standalone>

To target the correct hydrogen atom for removal, first we turn off the random hydrogen atom selection by passing “ov\_shuffle\_h=0”. Then we note that the index that we want to remove is the higher index. The default behavior is remove the lowest index hydrogen atom first. Therefore, we set “ov\_reverse=1” to instead start removal from the highest index.

To recreate the structures in the image shown above, run ExtendWithLinker quickly using the BCL standalone app molecule:Mutate.

<commandlines>

And that’s that! By adding this mover to our protocol, we will sample aromatic rings linked to the amino nitrogen, and we will preserve the hinge hydrogen bond donor from our ligand (unless Rosetta perturbs the pose substantially during FastRelax).

Let’s do a bit more rational design for our type I inhibitor simulations. The benzene ring of our scaffold is surrounded by several hydrocarbon sidechains. We can consider adding one or two substituents to form favorable Van der Waals interactions. To do this, we will use the Alchemy and Halogenate mutates.

<BCLFragmentMutateMover name="alchemy\_core"

ligand\_chain="X"

object\_data\_label="Alchemy(

druglikeness\_type=None,

mutable\_atoms=11 12 13 14 15,

allowed\_elements=C,

restrict\_to\_bonded\_h=1)"

/>

This defines an Alchemy mutate that will choose an atom from the benzene ring and change its bonded hydrogen atom into a carbon. If restrict\_to\_bonded\_h was set to 0, Alchemy would modify the chosen atom directly.

<BCLFragmentMutateMover name="halogenate\_core"

ligand\_chain="X"

object\_data\_label="Halogenate(

druglikeness\_type=None,

mutable\_atoms=11 12 13 14 15,

reversible=1,

allowed\_halogens=F Cl)"

/>

This defines a Halogenate mutate that will choose an atom from the benzene ring, open a valence by removing a hydrogen atom, and then add either a fluorine or chlorine. It is reversible, which means that this mutate has a 50% chance of removing a halogen from one of the mutable atoms.

You may ask “Can’t we just use Alchemy to add halogens?” Sure, no problem. Alchemy has the ability to mutate element types in halogens or vice versa if desired. However, usually in Alchemy we have to worry about making sure that we have an atom type available with the desired formal charge if we are swapping heteroatom identities or some such thing. Halogens are actually a bit easier because the atom typing is far more restricted, so in some sense the logic in Alchemy is overkill for halogens. Moreover, halogens are special in medicinal chemistry and deserve an opportunity for special logic that is ill-placed in Alchemy. For example, we can restrict the addition of individual halogens to aromatic rings, and we are experimenting with optional biased probabilities based on stability of the halogenated substructure. Nevertheless, the choice is yours.

In any event, that is more of a code design issue. Here, we will create a RandomMover with equal probability of choosing Alchemy or Halogenate.

<RandomMover name="decorate\_core" movers="halogenate\_core,alchemy\_core" weights="0.5,0.5" repeats="2"/>

This will give us up to two substituents on our benzene ring. The substituents will be either a methyl, fluorine, or chlorine.

Now put it all together! The final protocol in the XML script will perform an initial minimization of the starting receptor with the scaffold, followed by ExtendWithLinker to add an aromatic ring to our amino nitrogen atom, followed by two repeats of Alchemy and/or Halogenate to add substituents to our benzene ring. Finally, we do a constrained FastRelax on the protein-ligand interface and score the complex.

bash ../scripts/SimpleTest.design\_2.sh ../scripts/SimpleTest.design\_2.xml ../inputs/receptor/Abl\_Active\_A.pdb ../inputs/ligands/LIG\_0001.fa.pdb ../inputs/ligands/LIG.fa.params DEMO.design\_2.tk1\_

How do the inhibitors you designed look compared to the CITATION inhibitor? Are there similarities? Differences? Try additional mutates to see what else you can do.

One final note before we move on to Part 4: Have you noticed that you are able to perform sequential mutates on your molecule by referencing atom indices in your original scaffold? This is generally possible because new atoms are added to the end of the atom vector, so they do not change the numbering of our original indices. In our Alchemy mutate you may have wondered “Why is there this restrict\_to\_bonded\_h” option? Can’t we just specify an atom index for a hydrogen atom in the mutable\_atoms option?” Well, you can, but it is probably a bad idea. **During the fragment cleaning, retyping, and 3D conformer generation phase after the mutation is made, hydrogen atoms are actually removed and re-added.** This changes the indexing of hydrogen atoms. So, stick with referencing heavy atom indices and you should be good.

But what do you do if the molecule undergoes a large substructure change, or greatly expand the initial molecule and want to make changes to the new region? Find out how we handle that in Part 4!

Part 4: Type II inhibitor design

Type II inhibitors also compete with ATP binding, but unlike Type I inhibitors they extend into a back hydrophobic pocket of the kinase domain.

<superimpose our scaffold and the CITATION type II TKI with ATP>

Our rational design strategy will therefore start by extending the scaffold into the back hydrophobic pocket. Let’s start with our trusty friend ExtendWithLinker again.

<BCLFragmentMutateMover name="extend\_with\_linker"

ligand\_chain="X"

object\_data\_label="ExtendWithLinker(

ring\_library=%%rotamer\_library%%/ring\_libraries/individual\_rings/000.sdf.gz,

extend\_within\_prob=0.0,amide\_link\_prob=100000,amide\_n\_attach\_prob=1.0,

druglikeness\_type=None,

mutable\_atoms=14)"

/>

This time we will set an arbitrarily high probability for the amide linker. Amide linkers can be grown from two directions. The amide\_n\_attach\_prob specifies the probability that the nitrogen atom of the amide will be covalently bonded to the chosen atom in our starting fragment. The remainder of the probability is the likelihood that the carbon atom in the amide will be covalently bonded to the chosen atom. We set a single mutable atom. We also set a single ring to use from our library – a benzene ring (for simplicity)! Feel free to peruse the ring library or use your own rings.

So let’s assume you were to run just that mutate. You would now have your original scaffold plus an amide linker connecting it from atom index 14 to a new benzene ring. How do we go about modifying the new substructure in subsequent design steps? We certainly do not necessarily know what the atom indices will be.

Well, you may have noticed from the help menu options that we can specify mutable atoms by more than just their indices. There are four options:

1. Do not specify anything and all atoms default to mutable

2. Mutable atoms – Specify mutable atoms by their index in the molecule (0-indexed)

3. Mutable elements – All atoms of the specified element types are set as mutable.

4. Mutable fragments – This option is very useful. You can pass SD files of fragments for substructure comparison to your current molecule. You can pass as many or few fragments as you want. Atoms in matching substructures will be set to mutable. **These are cumulative.** There are two other options, mutable\_atom\_comparison and mutable\_bond\_comparison that allows users to specify the resolution of the substructure comparison. **Part of why this option is so useful is because there is also an option to invert the matched atoms.** If users set complement\_mutable\_fragments to true, then the mutable atoms are the ones that *did not* match in any of the substructure comparisons.

And if that is not good enough, **there are four identical options for setting fixed (non-mutable) atoms.** Importantly, fixed atoms are always set *after* the mutable atoms. This means that the fixed atom selections modify the mutable atoms and not vice versa. This is consistent with the default of setting all atoms to be mutable unless otherwise specified.

I am planning on adding a few more options so that users can selectively turn on or off more generic topological units, such as rings, chains, rigid fragments, etc. Stay tuned!

So with this new knowledge, let’s add a trifluoromethyl group to our newly created benzene ring.

<BCLFragmentMutateMover name="add\_medchem"

ligand\_chain="X"

object\_data\_label="AddMedChem(

medchem\_library=%%medchem\_fragments%%/trifluoromethyl.sdf.gz,

druglikeness\_type=None,

mutable\_fragments=%%mutfrag%%,

complement\_mutable\_fragments=1,

restrict\_additions\_to\_aromatic\_rings=1,

fixed\_elements=N O)"

/>

The mutfrag variable will be defined in our wrapper shell script. In this case, it will just be the SDF corresponding to our scaffold. We want to invert the substructure match, so we set complement\_mutable\_fragments to true. We then say that AddMedChem cannot try to attach the trifluoromethyl group to nitrogen or oxygen atoms, which will prevent it from randomly selecting one of the amide heteroatoms. It could in principle select the carbon atom from the amide to try and bond to; however, there are no valences that can be opened, which would cause the move to fail and try again up to n\_max\_mutates. Alternatively, we can save it the trouble and specify restrict\_additions\_to\_aromatic\_rings as true. The only aromatic ring atoms not in our starting scaffold are the benzene ring carbon atoms.

Thus, this mover will add a trifluoromethyl group to one of only five available atom indices on the amide-linked new benzene ring (two of which are symmetric, leaving us with three unique positions). At this point, if you want to even further restrict the space available for the AddMedChem mutate, the best advice I have is to start anew with the structure obtained after ExtendWithLinker so that you can see the atom indices. Beyond the obvious observation that we can only be so specific about how to add atoms to other atoms that do not exist in the starting fragment, it is also important to note that this software was written to help with drug design, not drug drawing. For the latter, stay tuned as we see how to integrate these tools into FoldIt Drug Design.

Finally, let’s add a small decoration to the core benzene ring using the Alchemy mutate.

<BCLFragmentMutateMover name="alchemy\_core"

ligand\_chain="X"

object\_data\_label="Alchemy(

druglikeness\_type=None,

mutable\_atoms=11 12,

allowed\_elements=C,

restrict\_to\_bonded\_h=1)"

/>

Put it all together and run the protocol.

bash ../scripts/SimpleTest.design\_0.sh ../scripts/SimpleTest.design\_0.xml ../inputs/receptor/Abl\_Inactive\_A.pdb ../inputs/ligands/LIG\_0001.fa.pdb ../inputs/ligands/LIG.fa.params DEMO.design\_0.tk2\_

This script provides everything you need to recreate the type II inhibitor in CITATION; however, there is some stochoasticity in the current design, and as we just discussed we have finite ability to create the exact molecule we want from scratch. But, more importantly, how do the output molecules look? Do they appear to be type II inhibitors? Do they have similar features to type II inhibitor in CITATION (or other TKIs you may have seen)?

Let’s expand our design further. Check out the SimpleTest.design\_1.xml script. How does it differ from the script we just ran? Do you expect more diversity or less in our designs? Run this protocol.

bash ../scripts/SimpleTest.design\_1.sh ../scripts/SimpleTest.design\_1.xml ../inputs/receptor/Abl\_Inactive\_A.pdb ../inputs/ligands/LIG\_0001.fa.pdb ../inputs/ligands/LIG.fa.params DEMO.design\_1.tk2\_

By this point, you have designed both type I and type II inhibitors. In Part 5, we will learn more about type I and II inhibitors and how we can begin to use our design protocols with receptor conformational sampling techniques.

Part 4 Extension (Optional): Adding the Fluorinate mover to our toolbox

In this tutorial we just used benzene as the base for our interactions with the back hydrophobic pocket. In principle, however, it does not need to be benzene. Here are a couple of extra things to try if you are interested:

1. Sample alternative rings. You can set your ring library to be an SDF with multiple rings, in which case one will be randomly selected each round, or you can sequentially iterate over SD files containing individual rings to enumerate.

2. Forget the rings. Let’s be creative. I will introduce you to the special purpose Fluorinate mutate. You may ask, “Why do we have a mutate specifically for fluorine when we can already manipulate fluorine atoms with Halogenate? If you really needed to use fluorine more dynamically, can’t you just enable it in Alchemy?” Good questions. These are things we considered. The answer boils down to “fluorine is special.” It has unique chemistry, and while it is often used to decorate scaffolds like chlorine or bromine or methyl or methoxy, it is also often used to saturate individual carbon atoms on aliphatic chains or non-aromatic rings. If you are new to small molecule drug design, you may have noticed that this effect can also be obtained with trifluoromethyl and trifluoromethoxy fragment additions in the AddMedChem mutate!

So, you will notice that the Fluorinate mutate contains several specific options to manage the minimum and maximum number of fluorine atoms that are added to a single carbon atom in a single move. Let’s combine the AddMedChem and Fluorinate mutates to add an ethylamide and saturate the terminal carbon with fluorine atoms.

bcl.exe molecule:Mutate -input\_filenames LIG.clean.sdf -output LIG.amide\_cf3.sdf -implementation "AddMedChem(mutable\_atoms=13,medchem\_library=/home/ben/workspace/bcl/rotamer\_library/medchem\_fragments/ethylamide.sdf.gz)" "Fluorinate(mutable\_fragments=LIG.clean.sdf,complement\_mutable\_fragments=1,fixed\_elements=N O, n\_min\_f=3,n\_min\_h\_sub=3)"

And let’s take it a step further. Let’s add a single carbon atom to our benzene ring first and *then* add the fluorinated ethylamide. This will give us a slightly longer chain.

bcl.exe molecule:Mutate -input\_filenames LIG.clean.sdf -output LIG.amide\_cf3.sdf -implementation "Alchemy(mutable\_atoms=13,allowed\_elements=C,restrict\_to\_bonded\_h=1)" "AddMedChem(mutable\_fragments=LIG.clean.sdf,complement\_mutable\_fragments=1,medchem\_library=/home/ben/workspace/bcl/rotamer\_library/medchem\_fragments/ethylamide.sdf.gz)" "Fluorinate(mutable\_fragments=LIG.clean.sdf,complement\_mutable\_fragments=1,fixed\_elements=N O, n\_min\_f=3,n\_min\_h\_sub=3)”

Visualize your molecules in PyMOL. Try making a protocol in RosettaScripts to do the same design but in the context of a receptor binding pocket.

Part 5: Induced-fit inhibitor design

You may have noticed that in the two previous design examples that we used two different PDB files for the Abl kinase receptor structure. For the Type I inhibitor we used the active state (activation loop extended outward; DFG-in) and for the Type II inhibitor we used the inactive state (activation loop folded inward; DFG-out). In actuality, kinases can adopt many conformations that are more or less active or inactive. The spectrum of available states differs between kinases, as well as the propensity for a kinase to be in one state or another. They are fascinatingly dynamic proteins!

Type II inhibitors require the kinase domain to be in either an inactive or intermediate state in order to bind. This is because the active state sterically occludes the back hydrophobic pocket that type II inhibitors target. In contrast, type I inhibitors have been observed in both active and inactive states (CITATION). Consistent with the observation that different kinases have different conformational preferences, it has also been observed that not all inhibitors bind to the same conformation across kinases. For example, dasatinib <contrast something with DDR1>.

<dasatinib example>

Therefore, it may be of interest to sample multiple conformational states of a protein during drug design. This can allow the user to filter out molecules that are predicted to preferentially bind an undesired conformation, or to identify features that stabilize a desired conformation.

Rosetta has a number of powerful tools for sampling protein conformational changes: fragment insertion, generalized kinematic closure, thermodynamic movers for canonical sampling, etc. There are also tools for sampling the interfaces of protein-ligand complexes explicitly, such as the low-resolution Transform mover and HighResDocker.

<transform mover xml>

<highresdocker xml>

A former graduate student in the Meiler Lab, Darwin Fu, wrote an enhancement to the Transform mover to also incorporate protein flexibility through pre-generated conformational ensembles. Users can pass an ensemble of proteins that are rapidly scored with the Transform mover scoring grids. The protein with the best score is carried forward to subsequent stages of the docking protocol.

For this example, we will use a variant of this protocol to score our inhibitor designs on multiple pre-generated conformations of the same protein prior to high-resolution refinement. The benefit to this approach is that it can be done faster than on-the-fly sampling of large conformational changes, and it ensures consistency in the sampled conformational space between molecule designs.

We will use 16 unique conformations of Abl kinase rigorously generated by Meng et al. 2018 using extensive molecular dynamics (MD) simulations coupled with Markov state models (MSM). The approach taken by Meng et al. blindly recovered known publicly available Abl conformations as well as previously unreleased inhibitor-stabilized conformers from the private Lilly database (CITATION).

Let’s begin. The Abl conformers have already been obtained from the literature and stashed in the inputs/receptors/ directory. Make a list containing the full paths of each of the conformers.

<ls blah>

Get the full path to the list

<readlink -e blah.list>

The script that we will use to run the RosettaScripts protocol is called scripts/InducedFitDrugDesign\_v1.sh. In that script we define a variable CONFS\_LIST. Define CONFS\_LIST with the full path to blah.list on your workstation. As we did previously, also make sure the rotamer library and fragments variables are set to the correct directory.

Let’s take a look at the Transform mover in this example.

<Transform name="transform" chain="X" box\_size="6.0" move\_distance="0.2" angle="2.0" cycles="500" repeats="1" temperature="%%temp%%" initial\_perturb="0.0" initial\_angle\_perturb="0.0" ensemble\_proteins="%%confs\_list%%" use\_main\_model="false" />

It differs from how we usually set it up for ligand docking because here we are not using it to sample large ligand pose changes. We just want to do a small bit of optimization of the designed ligand in each receptor pocket. We have small distance and angle perturbations, 0.2 Angstroms and 2.0 degrees, respectively, and we disable initial perturbations. The “ensemble\_proteins” option will take a variable corresponding to the PDB list we just made. The option “use\_main\_model” is set to false, indicating that the best scoring model is the one we will carry forward.

We have defined a number of BCLFragmentMutateMover movers that are very similar to the ones we used in Parts 3 and 4 of this tutorial. Take some time to go through them and make sure you fully understand what each is doing. One thing you will notice is that we are now using “druglikeness\_type=IsConstitutionDruglike” instead of “druglikeness\_type=None”. This acts as an internal filter to help prevent the design of grotesque molecules. To see what this filter does, run the following:

bcl.exe molecule:Properties -add "IsConstitutionDruglike(help)"

Finally, a note on the configurational sampling in this protocol. The initial minimization may bias the simulation toward the input conformer. Therefore, for production simulations it is advised that multiple independent trials are run starting from each conformer, or alternatively that the minimization step is removed and each conformer is minimized with the scaffold outside of this protocol prior to starting.

Let’s run this protocol!

First, for type I inhibitors:

i=1; bash ../scripts/InducedFitDrugDesign\_v1.sh ../scripts/InducedFitDrugDesign\_v1.xml ../inputs/receptor/conformers/ct7b01170\_si\_004\_0001\_A.pdb ../inputs/ligands/LIG\_0001.fa.pdb ../inputs/ligands/LIG 1.2 $i TEST\_t${i}\_ &

Second, for type II inhibitors:

i=2; bash ../scripts/InducedFitDrugDesign\_v1.sh ../scripts/InducedFitDrugDesign\_v1.xml ../inputs/receptor/conformers/ct7b01170\_si\_004\_0001\_A.pdb ../inputs/ligands/LIG\_0001.fa.pdb ../inputs/ligands/LIG 1.2 $i TEST\_t${i}\_ &

Finally, there is a RandomMover defined that will randomly select either a type I or type II design pattern if the “TYPE” variable in the wrapper shell script is set to “1-2”.

i=1-2; bash ../scripts/InducedFitDrugDesign\_v1.sh ../scripts/InducedFitDrugDesign\_v1.xml ../inputs/receptor/conformers/ct7b01170\_si\_004\_0001\_A.pdb ../inputs/ligands/LIG\_0001.fa.pdb ../inputs/ligands/LIG 1.2 $i TEST\_t${i}\_ &

Now visualize your output structures. How did they turn out? Are the type II inhibitors more frequently in an inactive state than an active state? Do the type I inhibitors have a conformational state preference? Note that we initialized these simulations with a receptor in the active conformation. If you start with a different protein conformer, does the output follow a similar trend?

In a preliminary benchmark using a related (but different) set of mutate options to create type I and II inhibitors, we found the following distribution of states:

<poster figure>

Congratulations! You have completed Tutorial 5. Please feel free to continue tinkering with the options. If you have any questions, do not hesitate to ask.