Non-canonical amino acid (NCAA) parameterization using Rosetta

When designing peptides, one can find the vocabulary of the twenty canonical amino acids to be restrictive in designing tight, specific binders to the receptor. Therefore, a framework needs to exist for modeling amino acids which can have any sidechain chemistry, i.e., non-canonical amino acids (NCAAs). In order to include these NCAAs into Rosetta design simulations, you must first parameterize them, which involves listing atoms and bonds and their respective types, recording the initial geometry, assigning rotamers, etc. Thankfully, most of this process can be handled automatically by the `molfile_to_params_polymer.py` script, but rotamer assignment still remains a challenge for which Rosetta proposes multiple solutions. This tutorial will detail how to use each of these tools in combination with `molfile_to_params_polymer.py` to go from a molfile (.sdf) of a NCAA to a parameter file (.params) which is useable in Rosetta design simulations.

1. Using existing canonical parameters for non-canonicals

The simplest way of assigning rotamers to a NCAA is to use a set of rotamers which already exist and simply attach them to your NCAA. However, this requires that your NCAA be quite similar to an existing canonical AA. For this step, we will be parameterizing a 3,4,5-trifluorophenylalanine (abbreviated in this tutorial as TFF), which highly resembles a phenylalanine. First, make a directory for your params files and `cd` into that directory.

```
mkdir NCAA_params
cd NCAA_params
```

The molfile for TFF is located at `../input_files/TFF.sdf`. There are a few things to note about this input which are required for proper parameterization by Rosetta. First of all, you should notice that the backbone is in a dipeptide form where each end of the amino acid is extended to a methyl group representing the adjacent C-alpha atoms of neighboring amino acids. This is necessary for Rosetta to understand how to connect this AA to adjacent AA's. In the file itself, there is also a set of lines which inform Rosetta which atoms correspond to the various atoms of the backbone, which atoms connect to the upper and lower AA's in the sequence, and properties such as charge, aromaticity, and chirality. Because of these instructions, to parameterize this NCAA, all we have to run is:

```
python <RosettaDir>/main/source/scripts/python/public/molfile_to_params_polymer.py \
    --clobber --polymer --no-pdb --name TFF --use-parent-rotamers PHE \
    -i ../input_files/TFF.sdf
```

Note the usage of `--use_parent_rotamers` in this command, as this is what establishes which canonical AA rotamers you want to use. If you look at the generated parameter file with `cat TFF.params`, you should see a line which says `ROTAMER_AA PHE`, indicating phenylalanine's rotamers are being used for this AA.

2. Rigorous rotamer calculation using MakeRotLib

In 2012, Renfrew et. al. developed MakeRotLib, for generating NCAA rotamers through minimization of iterated initial conformational states using a hybrid Rosetta/CHARMM energy function. This protocol remains the most rigorous calculation of NCAA rotamers that exists in Rosetta, but due to its rigor, its runtime is not suitable for large libraries of NCAAs. In addition, the runtime scales exponentially with the number of chi angles and caps at 4 chis, so this protocol is also not suitable for highly flexible sidechains. Finally, MakeRotLib is not capable of handling anything other than monosubstituted alpha amino acids, so if your amino acid structure is exotic, it won't be able to be processed by MakeRotLib. To demonstrate the functionality of MakeRotLib, you will parameterize an amino acid with an ethyl group as a sidechain (abbreviated as EAA in this tutorial). Similar to the previous case, `EAA.sdf` is in dipeptide form and has the necessary instructions for Rosetta to parameterize the molecule. Run `molfile_to_params_polymer.py` to get the parameter file:

```
python <RosettaDir>/main/source/scripts/python/public/molfile_to_params_polymer.py \
    --clobber --polymer --no-pdb --name EAA -i ../input_files/EAA.sdf
```

However, since we excluded `--use_parent_rotamers`, this parameter file is not ready for use in Rosetta yet. In order to run MakeRotLib and generate the rotamers, you will need an options file, supplied at `../input_files/EAA_makerotlib_options.in`. This file specifies the angle ranges over which MakeRotLib should iterate, the number of chi angles in the sidechain, and initial guesses as to how many chi angle bins there are and where they lie. For this tutorial, the options file will consider all possible phi and psi angle values at increments of 10 degrees, every value of the single chi angle at 30 degree increments, and assume there are 3 chi rotamer bins each spaced 120 degrees apart (which is reasonable since the canonicals generally show this pattern as well). Now to run MakeRotLib:

```
<RosettaDir>/main/source/bin/MakeRotLib.default.linuxgccrelease \
    -extra_res_fa ./EAA.params -score:weights mm_std \
    -options_file ../input_files/EAA_makerotlib_options.in
```

This calculation should take a few minutes, after which you should have a ton of `EAA_*` files in your current directory. This directory contains logs from running MakeRotLib as well as a `.rotlib` file for each pair of phi/psi angle values. The final objective is to consolidate all of these files into a single `.rotlib`, and add a reference to this rotlib into the parameter file:

```
for i in `seq -170 10 180`; do
    for j in `seq -170 10 180`; do
        cat EAA_180_${i}_${j}_180.rotlib >> EAA.rotlib
    done
done
echo "NCAA_ROTLIB_PATH $PWD/EAA.rotlib"  >> EAA.params
echo "NCAA_ROTLIB_NUM_ROTAMER_BINS 1 3" >> EAA.params
```

Open the EAA.rotlib and EAA.params files with a text editor, and check to make sure there's no obvious errors. If not, you can remove the temporary files

```
rm -rf EAA_*
```

Now that the file `EAA.params` has been assigned rotamers, it is now ready to use in Rosetta. Note that the `NCAA_ROTLIB_PATH` is hardcoded, so if you use the example .params file in `output_files`, you will need to change this path to match your environment.

3. Small molecule conformers as NCAA rotamer libraries (FakeRotLib)

While MakeRotLib is the most accurate method for rotamer construction in Rosetta, it does not apply in many contexts, as was previously discussed. To address some of these shortcomings, we consider the NCAA as a small molecule and define the rotamers of the NCAA as low energy conformers of the "small molecule". The implementation of this idea is the `fake_rotlib.py` script, which uses RDKit to generate conformations of the NCAA, score the conformations using the UFF forcefield, and utilize the N lowest energy conformations in the parameter file as "PDB rotamers". The distinction between this implementation of the rotamer library and the previous methods is that the previous methods define the *distribution* of rotamers and then score a given conformation according to its position in that distribution, whereas PDB rotamers store a set of acceptable conformations and randomly draws from these conformations when modeling the residue. Since PDB rotamers don't have to fit into the distribution parameters accepted by Rosetta, pretty much any NCAA can be accommodated by PDB rotamers. On the other hand, PDB rotamers inherently discretize the conformational space, are not compatible with some movers, and generally require more compute time and memory in modeling. To allow both types of rotamer libraries to be built, `fake_rotlib.py` also has functionality to generate a rotamer distribution file (`.rotlib`) from the PDB rotamers (as long as the NCAA has four or less chi angles). In addition to rotamer modeling, `fake_rotlib.py` automates a few other steps of the process, including dipeptide capping, writing the params instructions, and running `molfile_to_params_polymer.py`. For more information on FakeRotLib, see it's relevant publication.

As a demonstration of `fake_rotlib.py`, we parameterize another phenylalanine derivative (with an attached Bis(2-chloroethyl)amine group), abbreviated as MFF in this tutorial. Since this molecule has far more chi angles than any other we've parameterized before, we will be using PDB rotamers in lieu of a `.rotlib` file. `fake_rotlib.py` depends on RDKit to generate conformers, so we need a python environment with it installed. Since RDKit is already installed in our base python interpreter, simply run:

```
python <RosettaDir>/main/source/scripts/python/public/fake_rotlib.py \
    --input ../input_files/MFF.sdf --dip -n 100
mv ../input_files/MFF* ./
```

Note that the `--dip` flag is used here because the input `MFF.sdf` is already in dipeptide form and has parameterization instructions pre-generated. If instructions need to be generated, run without this flag and ensure that the input is NOT in dipeptide form (either neutral or zwitterionic backbone is acceptable). This causes two important files to be generated: the `MFF.params` which possesses a reference to the PDB rotamers file `MFF_rotamer.pdb`. `MFF_rotamer.sdf` is also here, but this is an intermediate file used input to `molfile_to_params_polymer.py`. Regardless, as long as `MFF_rotamer.pdb` remains in the same directory as `MFF.params`, the file is ready to be used in Rosetta simulations.

Macrocyclic peptide design in Rosetta

This tutorial will guide you through the basics of macrocyclic peptide design based on the protocol published in Mulligan et. al. Computationally designed peptide macrocycle inhibitors of New Delhi metallo-beta-lactamase 1. Proc Natl Acad Sci U S A. 2021 Mar 23;118(12):e2012800118. doi: 10.1073/pnas.2012800118. PMID: 33723038; PMCID: PMC8000195.

1. Background

Peptides are interesting molecules because they lie in size between small molecules and have properties of both including high-affinity and incorporation of non-canonical amino acids. Peptides are attractive as protein-protein interaction and enzyme catalysis inhibitors. Mulligan et. al developed a protocol that designs peptide macrocycle inhibitors of New Delhi metallo-beta-lactamase 1, an enzyme that degrades beta-lactam antibiotics. They start from the structure of L-captopril, a small molecule with weak inhibition of New Delhi metallo-beta-lactamase 1 and develop a macrocycle with 50 times greater potency.

The PDB structure of L-captopril bound to New Delhi metallo-beta-lactamase 1 is 4EXS. L-captopril looks a a D-cysteine, L-proline dipeptide and is easily convered into a D-cysteine L-proline dipeptide stub. This stub will serve as the anchor for peptide extension.

Note that example output files for the macrocyle protocol and be found in the `macrocycle/demo` directory.

2. Anchor Extension

If you are not already in the macrocycle directory, cd to macrocycle - assuming you are in peptide_ncaa_macrocycle_design:

```
cd macrocycle
```

The prepared inputs for extension can be found in `extend/inputs` and include the dipeptide stub, the Rosetta flags, and a manual foldtree.

Anchor extension requires generation of a foldtree, so that pertubation of anchor stub torsion angles will not disrupt the desired anchor interaction geometry. This can be found in `inputs/foldtree1.txt`. (See Appendix for link on foldtrees.)

From the macrocycle directory:

```
cd extend/inputs
```

Open the dipeptide stub 4EXS_Dcys_Lpro.pdb in pymol:

```
pymol 4EXS_Dcys_Lpro.pdb
```

Find the dipeptide stub that binds to the Zn catalytic site and will serve as the anchor.

Now that you understand where the dipeptide stub binds, we will extend the stub to form an 8 residues macrocyclic peptide. The rosetta scripts xml found in `extend/xml/NDM1i_1_design.xml` uses the PeptideStubMover to extend the stub and sets the torsion values of the dipeptide stub backbone to chemically senible values. Additionally, the xml add peptide cutpoints to N and C terminus and declares a bond between the termini so that Rosetta no longer has repulsive terms for the termini.

Now, move to the extend directory:

```
cd ../
```

Make the output directory. Visualize and run the command to extend the peptide stub (should take less than a minute to run):

```
mkdir output

~/rosetta_workshop/rosetta/main/source/bin/rosetta_scripts.default.linuxgccrelease \
    -in:file:s inputs/4EXS_Dcys_Lpro.pdb -parser:protocol xml/NDM1i_1_design.xml \
    @inputs/rosetta.flags -out:prefix output/ -nstruct 5
```

This command creates five extended peptides that can be found in the output directory. Visualize these structures in pymol:

```
pymol output/*.pdb
```

Notice that the geometry of the N to C terminal bond is incorect because have not yet closed the bond with loop closure (genkic), only declared that it exists.

3. Cyclization

In Rosetta, Generalized Kinematic Closure (GenKIC) is used to close/model loops and can go through covalent linkages such as disulfide bonds of N to C terminal cyclization. We will use genkic to close the terminal peptide bond, setting the angles and bond distances of this bond to the ideal value. Additionally, genkic is used to sample different backbone geometries of cyclic peptides that can be designed. The XML in `cyclize/xml/NDM1i_1_design.xml` closes the terminal peptide bond and filters for peptide internal hydrogen bonds - important for designing stable peptides that lack a hydrophobic core - and steric clashes with the receptor. This step is computationally expensive due to the high filter failure rate, so you should open a new terminal tab to run these commands and look at the results later.

From the macrocycle directory in a new tab (should take about 2 minutes for 5 backbones - all may not be sucessful):

```
cd ../cyclize
mkdir output

~/rosetta_workshop/rosetta/main/source/bin/rosetta_scripts.default.linuxgccrelease \
    -in:file:s inputs/4EXS_Dcys_Lpro.pdb -parser:protocol xml/NDM1i_1_design.xml \
    @inputs/rosetta.flags -out:prefix output/ -nstruct 5
```

Once the backbone cyclization completes, you can use these backbones as the starting point for design, but note that some designs are expected to fail filters. In an actual peptide search, you would generate thousands of designed peptide from different backbones. For now, move onto designing a given backbone in the design directory.

3. Design

The design protocol for cyclic peptides uses a combination of repacking a minimization to design a given backbone and filters for oversaturated hydrogen bond acceptors - the Rosetta energy function is pairwise and cannot detect oversaturated hydrogen bond acceptors - as well as shape complementarity and internal hydrogen bonds. The packer pallette for design includes the L amino acids and their D sterioisomers, but exludes GLY and CYS residues to aid with conformational stability of the design.

Additionally, we can include non-canonical amino acids, such as TFF from Part 1 of the tutorial in the design pallete.

Note that this protocol uses amino acid composition constraints to enforce among other things incorporation of hydrophobic and proline amino acids.

Edit the XML script to add TFF to the set of residues being designed:

```
cd ../design
gedit xml/NDM1i_1_design.xml
```

While editing the XML script, add TFF to the packer line, the line for L hydrophobic design (only needed because of the amino acid composition used - see Appendix), and write the change. Lines similar to the following should already exist in the XML – find them and edit in the changes

Under PACKER_PALETTES:

```
<CustomBaseTypePackerPalette name="design_palette"
    additional_residue_types="DALA,DASP,DGLU,DPHE,DHIS,DILE,DLYS,DLEU,DMET,DASN,DPRO,DGLN,
                              DARG,DSER,DTHR,DVAL,DTRP,DTYR,TFF"
/>
```

Under TASK_OPERATIONS:

```
<RestrictToSpecifiedBaseResidueTypes name="L_hydrophobic_design"
    base_types="PHE,ILE,LEU,MET,PRO,VAL,TRP,TYR,TFF"
    selector="select_L_hydrophobic_positions"
/>
```

We will be designing with provided backbones that can be found in `inputs/4EXS_Dcys_Lpro_native.pdb` and `../cyclize/output/*.pdb`. The PDB in inputs is the backone of one of the crystalized macrocycle inhibitors and the demo backbones are provided to ensure higher probability of sucessful design and incorporation of TFF.

To design with the given backbone (Will take about 20 minutes, but you can start the visualization as they are made approximatly every 2 minutes):

```
mkdir output

~/rosetta_workshop/rosetta/main/source/bin/rosetta_scripts.default.linuxgccrelease \
    -in:file:s inputs/4EXS_Dcys_Lpro_native.pdb ../demo/cyclize/output/*.pdb \
    -in:file:extra_res_fa ../../ncaa/output_files/TFF.params \
    -parser:protocol xml/NDM1i_1_design.xml \
    @inputs/rosetta.flags -out:prefix output/ -nstruct 1
```

Visualize these structures in pymol:

```
pymol output/*.pdb
```

Optional Monte Carlo Pertubation of Initial Designed Peptides

Mulligan et. al. used a monte carlo protocol to explore the local conformational space of initial designed peptides, optimizing the peptide - enzyme shape complementarity. The xml for this step is more complicated and the procedure is computationally expensive, so this section is optional to run and would require a deeper dive to fully understand.

To run the monte carlo protocol:

```
cd ../mc_sample
mkdir output

~/rosetta_workshop/rosetta/main/source/bin/rosetta_scripts.default.linuxgccrelease \
    -in:file:s inputs/4EXS_Dcys_Lpro_native_0001.pdb -parser:protocol xml/NDM1i_1_design.xml \
    @inputs/rosetta.flags -out:prefix output/ -nstruct 5
```

Full Protocol

The full protocol, combining all steps into one xml can be found in the og_scripts directory

Appendix

For the original github for these scripts: https://github.com/vmullig/ndm1_design_scripts

For more details on foldtrees, go here: https://docs.rosettacommons.org/demos/latest/tutorials/fold_tree/fold_tree

Other helpful movers for modeling peptides in Rosetta:

CycpepRigidBodyPermutationMover https://docs.rosettacommons.org/docs/latest/scripting_documentation/RosettaScripts/Movers/movers_pages/CycpepRigidBodyPermutationMover

Simple Cyclic Peptide Prediction (simple_cycpep_predict) Application https://docs.rosettacommons.org/docs/latest/structure_prediction/simple_cycpep_predict

PeptideCyclizeMover https://docs.rosettacommons.org/docs/latest/scripting_documentation/RosettaScripts/Movers/movers_pages/PeptideCyclizeMover

Amino Acid Composition:

- https://docs.rosettacommons.org/docs/latest/scripting__documentation/RosettaScripts/Movers/movers__pages/AddCompositionConstraintMover
- https://docs.rosettacommons.org/docs/latest/rosetta__basics/scoring/AACompositionEnergy

Cyclic Peptide AI Design Tutorial

This tutorial will guide you through the complete workflow for designing cyclic peptides using AI tools. We will first generte cyclic backbones with RFDiffusion, then design the backbone sequence design with ProteinMPNN, and finally predict and validate the peptide structure with AlphaFold2.

Step 1: Backbone Generation with RFDiffusion

We begin our design process by generating cyclic peptide backbones using RFDiffusion, which has been trained to understand protein geometry and can generate realistic backbone conformations. For cyclic peptide generation with RFDiffusion, peptides of sizes 10-14 are generally good sizes to use, though you can experiment with smaller or larger peptides as well. For an individual peptide, the RFDiffusion CPU code generally takes less than half a minute per structureon the molgraph machines, so fell free to increase the `num_designs` parameter. The benefit of staying at or under 14 amino acids is that you can re-predict the peptide structure using the orthogonal Rosetta simple_cyc_pep_predict application.

To generate the cyclic peptide backones, activate the RFDiffusion environment using `conda activate SE3nv` and create the output directory with `mkdir -p cyclic_gen`. Then run the following command:

```
~/rosetta_workshop/RFdiffusion/scripts/run_inference.py \
    'contigmap.contigs=[10-10]' \
    inference.output_prefix=./cyclic_gen/macrocycle \
    inference.num_designs=5 \
    inference.cyclic=True \
    inference.cyc_chains='a'
```

This command generates five different 10-amino acid cyclic peptide backbones, with the cyclic constraint applied to chain 'a'. The contigs parameter `[10-10]` specifies that we want peptides of exactly 10 residues, but you can modify this to generate different sizes such as `[8-8]`, `[12-12]`, or even ranges like `[10-14]`.

Workshop participants should feel free to change the peptide size in the `contigs` or design more peptides by increasing the `num_designs` parameter. The generated backbones will be saved to the `cyclic_gen` directory. When examining these structures, look for backbones that display clear secondary structure elements such as beta sheets or alpha helical character, as these are more likely to lead to successful designs for the next steps.

Step 2: Sequence Design with ProteinMPNN

Now, we will use ProteinMPNN do design your favorite peptide backbone from step 1. When making this selection, prioritize peptides with more internal hydrogen bonds and, those that have more beta sheet or alpha helical character, as these structural features are more likely to be successful for generating a stable peptide with this pipeline.

To begin sequence design, activate the ProteinMPNN environment with `conda activate ligandmpnn_env` and create the output directory using `mkdir -p mpnn_design`. Then run ProteinMPNN with the following command:

```
python ~/rosetta_workshop/LigandMPNN/run.py \
    --checkpoint_protein_mpnn ~/rosetta_workshop/LigandMPNN/model_params/proteinmpnn_v_48_020.pt \
    --checkpoint_path_sc ~/rosetta_workshop/LigandMPNN/model_params/ligandmpnn_sc_v_32_002_16.pt \
    --pdb_path cyclic_gen/macrocycle_1.pdb \
    --out_folder mpnn_design/ \
    --model_type "protein_mpnn" \
    --number_of_batches 1 \
    --batch_size 10 \
    --pack_side_chains 1 \
    --number_of_packs_per_design 1
```

Remember to update the `--pdb_path` parameter to point to your chosen backbone structure from the cyclic_gen directory. This command will generate 10 different sequence designs for your chosen backbone, with optimized side chain conformations. The `--pack_side_chains 1` option ensures that ProteinMPNN not only designs the sequence but also optimizes the side chain rotamers for better packing. The packed designs can be visualized using `pymol mpnn_design/packed/*.pdb`, where you can examine how well the designed sequences fill the backbone structure. The designed sequences themselves can be found by examining the contents with `cat mpnn_design/seqs/macrocycle_*.fa`, which contains the amino acid sequences in FASTA format.

Step 3: Structure Prediction with AlphaFold2

The final step in our workflow involves using AlphaFold2 via ColabDesign to predict the structure of our designed cyclic peptide. This structure prediction step will validate whether AlphaFold thinks our design will fold as intended. To predict the structure, we will use AlphaFold with a cyclic peptide offset via a Python script that uses ColabDesign to predict the structure of the designed cyclic peptide.

Before running the AlphaFold prediction, set the JAX platforms to CPU using `export JAX_PLATFORMS=cpu` as the molgraph machines to not have a GPU availiable. Extract your chosen designed sequence from the ProteinMPNN output by examining the sequences in the mpnn_design/seqs directory. Once you've selected a promising sequence, run the AlphaFold prediction using the provided script and the colabdesign python from sbgrid:

```
python.colabdesign af_cyc_pred.py KILPGVISEG -o af_output --model_num 1 \
    --params_path /programs/x86_64-linux/colabdesign/1.1.2/
```

Replace "KILPGVISEG" with your actual designed sequence. The script automatically applies cyclic peptide constraints by connecting the N-terminus to the C-terminus, ensuring that AlphaFold considers the cyclic nature of the peptide during structure prediction.

The resulting AlphaFold predicted peptide can be visualized at the default location using `pymol af_output/cyclic_peptide.pdb`. In PyMOL, the `spectrum b` command will color the peptide by B-factor, which corresponds to AlphaFold's confidence scores. The range of B-factors is printed to the screen. The script also generates a results file containing key metrics including pLDDT (overall confidence), pTM (template modeling score), and PAE (predicted aligned error), which help assess the quality and reliability of the prediction. For peptides, pLDDT is generally the best method to assess quality and pLDDT scores above 0.9 (this pLDDT is rescaled be be between 0 and 1) are considered good. Because peptides are so small, the pTM is not an accurate representation of a peptides stability.

Expanding this Workflow to Peptide Binder Design

This workflow can easily be expanded to designing peptide binders either de novo or that stabilize a known binding motif. In this use case, you may want to generate 10,000+ starting backbones for sequence design. After designing the peptide sequence and predicting its structure in complex with the target, iteratively redesigning and repredicting the peptide-target complex can optimize the peptide-target interactions into a design minima. The ColabDesign github (https://github.com/sokrypton/ColabDesign) has a designability_test.py script that can serve as a template for designing and validating peptide binders.

Alternatively, structure prediction methods based on AlphaFold3 such as boltz and RosettaFold3 can be used to validate peptide-protein complexes.