Enzyme design with RFdiffusionAA and LigandMPNN

Bold text means that these files and/or this information is provided.

Italicized text means that this material will NOT be conducted during the workshop

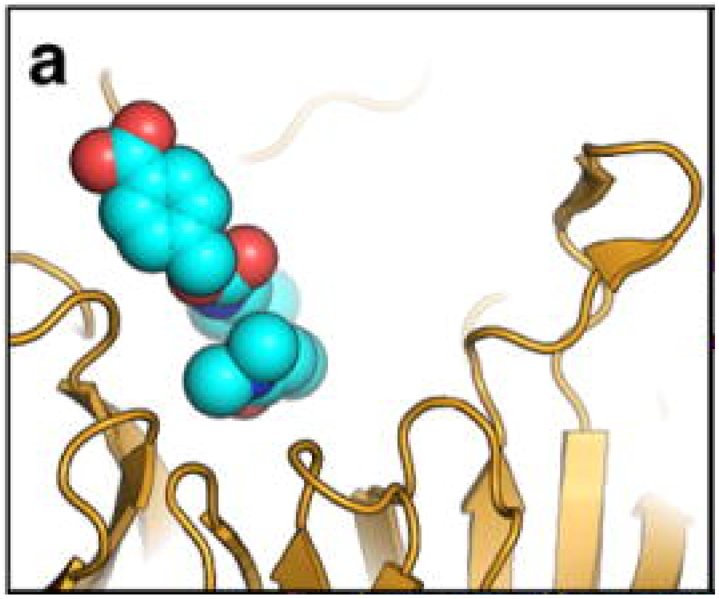
```
fixed width text means you should type the command into your terminal
```

If you want to try making files that already exist (e.g., input files), write them to a different directory (`mkdir my_dir`)! Copy & pasting can sometimes lead to weird errors, so when in doubt try to type the commands instead.

Tutorial

In this tutorial we'll be attempting to replicate the work of Eiben et al. 2012 https://doi.org/10.1038%2Fnbt.2109, but with recent machine-learning based tools.

The enzyme DA_20_10 is a *de novo* designed Diels Alderase enzyme from that was created using previous Rosetta techniques. (Sigel et al. 2010 https://doi.org/10.1126%2Fscience.1190239) Like many such similar *de novo* enzymes, it has low activity. Examination of the model of the putative transition state bound to the enzyme indicates that the small molecule is rather exposed in the active site.



It was hypothesized that attempting to redesign the loops of the TIM barrel to better cover the reacting molecule may help increase the rate of reaction, by increasing the number of substrate contacts and by better shielding the reacting ligand from the solvent.

Note that we're choosing a relatively small enzyme redesign project to minimize the time/computational requirements required. The same basic protocol can be performed with much more aggressive/expansive redesign. Indeed, the original design work that lead to DA_20_10 in the first place, one which started from just the transition state and interacting residues (the "theozyme") could theoretically be done with a similar sort of protocol. See https://github.com/ikalvet/heme_binder_diffusion for an example protocol from the RFdiffusion paper which builds heme proteins from just the ligand definition and a single interacting residue.

As always, start by making a working directory:

```
mkdir working
cd working
```

Input preparation

As we're doing a partial redesign of the structure, we need to have the template structure. A variant of this enzyme can be downloaded from https://rcsb.org as 3I1C. Alternatively, copy it from the inputs/ directory

```
cp ../inputs/3i1c.pdb .
```

This structure was crystalized without any ligand in the active site. A transition state mimic is provided for you in `inputs/LIG.pdb`. Note that the chain letter of the provided ligand is 'X', which does not conflict with the chain letter of the protein (which is 'A'), so we can combine the two structures by simple concatenation:

```
cat 3i1c.pdb ../inputs/TS.pdb > 3i1c_lig.pdb
```

Take a look at the structure in PyMOL. Notice how relatively open the pocket is, and how extending and rearranging some of the loops (notably chain A residues 35-47) may have the potential to "fold down" across the top of the pocket and cover the top of the ligand.

```
pymol 3i1c_lig.pdb
```

While in PyMol, be sure to click on the ligand and make note of the three letter code being used - we'll need that later.

RFdiffusionAA rebuilding of the loop.

RFdiffusionAA (RoseTTAFold Diffusion All Atom) is a ligand-aware version of RFdiffusion. It will attempt to build native-like protein backbones based on existing protein context and the presence of the ligand. In contrast with regular RFdiffusion, non-protein residues (ligands, RNA, DNA, metal ions, etc.) are present during the diffusion process, so the backbones generted by RFdiffusionAA will attempt to avoid making clashes with those non-protein residues, as well as being placed such that they're located with the potential for developing favorable interactions with those ligands.

To run RFdiffusionAA, we need our context structure ("3i1c_lig.pdb"), knowledge of which ligand(s) we're attempting to use during the process (here we're using a ligand named "X00"), as well as the 'contig' specification. The 'contig's for RFdiffusionAA are the same as in regular RFdiffusion, and specify which parts of the protein are coming from the template protein, and which are being generated from scratch. For our purpouses, we want to keep everything from the template (from chain A residue 1 to residue 314), except for the loop region of chain A residue 35-47. This we want to rebuild, adding anwhere up to 20 extra amino acids. As such, the contig map in our case would be as follows:

```
[A1-34,13-33,A48-314]
```

Designations with chain letters (e.g. "A1-34" and "A48-314" specify regions of the template protein we want to copy over. Specifications without a chain letter specify the length of new protein to generate. Here the "13-33" means to sample protein lengths from 13aa to 33aa long. Since we don't specify any chainbreak designations (`/0`, with the space), this will be modeled as a single chain.

The contig map only specifies the protein residues to build. The ligand(s) to be used are specified with the `inference.ligand` parameter.

RFdiffusionAA is most easily installed as an Apptainer (formerly Singularity) container, which is a system similar to Docker, but with adjustments which make it more suitable for running on academic cluster systems. For technical reasons, the current version of RFdiffusionAA must be run from the RFdiffusionAA directory, rather than the working directory. This means that we need to make explicit reference to the full path of the input and output locations.

For time reasons, we're only generating a single output structure (`inference.num_designs=1`). Generally, you'll want to create hundreds of different structures and then filter them for those with the properties which you desire. Also, we're reducing the number of diffusion timesteps taken (`diffuser.T=20`). While RFdiffusionAA was trained with 200 timesteps, experience with plain RFdiffusion indicates that as few as 20 denoising step gives acceptable results

```
pushd ~/rosetta_workshop/RFdiffusionAA # Temporarily move to the RFdiffusion directory

apptainer run --bind ${HOME} --nv rf_se3_diffusion.sif -u run_inference.py \
    inference.output_prefix=${HOME}/rosetta_workshop/tutorials/ml_enzyme_design/working/rfdaa \
    inference.input_pdb=${HOME}/rosetta_workshop/tutorials/ml_enzyme_design/working/3i1c_lig.pdb \
    contigmap.contigs=[\'A1-34,13-33,A48-314\'] \
    inference.ligand=X00 \
    inference.num_designs=1 \
    inference.design_startnum=0 \
    diffuser.T=20

popd # Return to the working directory
```

(We use `${HOME}` here as the `~` alias for your home directory is expanded by the shell, but only if it's preceeded by a space. The option parsing approach used by RFdiffusionAA does not allow us to add a space there.)

Analysis of generated backbones.

As there's only limited time to run the generation process, we've pre-generated a number of possible outputs for you. These are found in the `example_outputs/` directory. Even then, the number of outputs provided are relatively modest. In any actual production run, you'll likely want to generate more structures. This is particularly the case for design cases where you're doing more extensive generation. (e.g. if you're generating a full domain from scratch.)

While RFdiffusionAA attempts to come up with biologically relevant backbone conformations, the particular backbones it comes up with do not always accord with what *you* are interested in. While there are various ways to bias the generation process (see https://github.com/RosettaCommons/RFdiffusion for potential options), often the easiest approach is to simply generate a number of structures and then filter out those which are not relevant. Note that filtering at this stage can be useful, as throwing out backbones with obvious issues will reduce the computational cost of downstream steps.

The filtering process is going to be dependent on the particular design goals of the specific project you're doing. Factors such as radius of gyration, size and type of secondary structural elements, contact order, and burial of certain residues (e.g. the ligand) can all be assessed with tools such as RosettaScripts and PyRosetta, as well as other structural examination programs. The important thing to keep in mind when building filters is that sidechains and sequences have yet to be assigned. As such, all the filters at this stage of the pipeline should be ones which are based on the backbone only.

Note that often times the best filtering approach is a manual one – simply open up the structures in a viewing program like PyMol and use your biochemical knowledge about the system to determine which structures have the possibility to work, and which are obviously silly. Often times the filtering process is an iterative one, with manual examination suggesting automated filters which could be implemented, followed by further manual examination to pick up on additional issues.

Open up the provided example outputs and examine the backbone of the generated loop – what sorts of things do you like about some loops, and what features do you think aren't benefitial?

```
pymol ../example_outputs/rfdaa_*.pdb
```

LigandMPNN design of protein seqeunces.

Like regular RFdiffusion, RFdiffusionAA only creates backbone structures, with the added residues represented in the output structure as glycine. Before using these results, sequences need to be generated. Just like RFdiffusionAA is a version of RFdiffusion trained to be ligand-aware, LigandMPNN is a ligand-aware version of ProteinMPNN. We'll be using LigandMPNN to assign amino acid identities to the generated residues.

```
conda activate ligandmpnn_env

python ~/rosetta_workshop/LigandMPNN/run.py \
    --checkpoint_ligand_mpnn ~/rosetta_workshop/LigandMPNN/model_params/ligandmpnn_v_32_010_25.pt \
    --checkpoint_path_sc ~/rosetta_workshop/LigandMPNN/model_params/ligandmpnn_sc_v_32_002_16.pt \
    --pdb_path rfdaa_0.pdb \
    --out_folder mpnn_full/ \
    --model_type "ligand_mpnn" \
    --number_of_batches 1 \
    --batch_size 10 \
    --pack_side_chains 1 \
    --number_of_packs_per_design 1 \
    --pack_with_ligand_context 1
```

The "pack_side_chains" options can be used to generate output PDBs with the proper sidechain identities and atoms already present.

The designed sequences and the associated metrics can be found at `mpnn_full/seqs/`. The output is in FASTA multiple sequence alignment format, so we can look at the various designs with a multiple sequence alignemnt viewer

```
jalview

# File -> Input Alignment -> From File
# Open the rfdaa_0.fa file in the ml_enzyme_design/working/mpnn_full/seqs/ directory
# Color -> Percentage Identity
```

This should now display the difference between the various designs and the starting structure, colored by how close they are to the majority identity at that position.

Note that most of the positions have variations. This is because we did a full design, rather than focusing just on those residues which were added. If we want to just design those residues which were added, we can use a script to build an input file which will redesign all Glycine residues

```
../scripts/make_residue_list.py -o redesign.json rfdaa_*.pdb
```

This output file simply extract those residues which are listed as being generated in the RFdiffusionAA-generated .trb file, and formats them for LigandMPNN. The RFdiffusion generated residues aren't necessarily the only residues to redesign! You likely also want to add residues that are contacting the redesigned regions as well. (Which set of residues you'll want to include in your redesign will likely depend on the particular system you're working with.)

```
python ~/rosetta_workshop/LigandMPNN/run.py \
    --checkpoint_ligand_mpnn ~/rosetta_workshop/LigandMPNN/model_params/ligandmpnn_v_32_010_25.pt \
    --checkpoint_path_sc ~/rosetta_workshop/LigandMPNN/model_params/ligandmpnn_sc_v_32_002_16.pt \
    --pdb_path_multi redesign.json \
    --redesigned_residues_multi redesign.json \
    --out_folder mpnn_loop/ \
    --model_type "ligand_mpnn" \
    --number_of_batches 1 \
    --batch_size 10 \
    --pack_side_chains 1 \
    --number_of_packs_per_design 1 \
    --pack_with_ligand_context 1
```

Load these designed sequences (now in the mpnn_loop/seq directory) in jalview and examine the difference from before.

The mpnn_loop/packed/ directory contains structures with the sidechains applied to the backbone. You can open these with PyMol to examine them

```
pymol mpnn_loop/packed/*.pdb
```

Note that LigandMPNN will not have changed the backbone – you'll need to show sidechains in order to see the differences in the LigandMPNN designs.

Analysis of generated sequences

Now that you have sequences and sidechains, you can potentially run more detailed filtering of designs. However, when filtering keep in mind that the backbone structure hasn't been optimized for the particular sequence. Features which may be sensitive to small changes in backbone movement or precise sidechain location may change as the structure is further optimized.

One potential metric one can use in evaluating the sequence is the confidence metric that is reported for each design in the sequence file. Not only can this number be used to rank different designs on the same backbone against one another, it can also be used to evaluate different backbones – a backbone which can only generate low-confidence sequences may be structurally frustrated, and that backbone structure may not be achievable through any experimental condition.

Optimization of backbone structure with Relax

RFdiffusionAA creates a new backbone, and LigandMPNN assigns sequences to that fixed backbone. To better evaluate how well the ligand is bound by the designed protein, it may be helpful to relax and evaluate the interaction energy between the ligand and the protein. The relax_with_ligand.xml script combines an structural/energy optimization protocol (Rosetta FastRelax) with a mover which reports the energy of interaction of the ligand with the protein. When given a reference structure, it will also report the rmsd of the ligand residue from the reference. We also include a metric to report the solvent accessible surface area of the ligand to act as a proxy for how well the ligand is now shielded from the solvent. (Lower SASA means better coverage by the redesigned loop.)

In order for Rosetta to be aware of the ligand, it needs to be provided with a params file. Creating a params file for a small molecule can be done with the molfile_to_params.py script provided with Rosetta, assuming you have an SDF or a MOL2 file of the ligand. Creating one for a transition state analog is similar: molfile_to_params.py does not require synthesizable geometry and bond connections. So long as the atoms are present and connected with each other, it should work. We'll be using a pre-generated params file for this tutorial. See the ligand docking tutorial from https://meilerlab.org/tutorials/ for more details on how to generate your own.

```
~/rosetta_workshop/rosetta/main/source/bin/rosetta_scripts.linuxgccrelease \
    -parser:protocol ../inputs/relax_with_ligand.xml \
    -out:file:scorefile relax_results.tab \
    -s mpnn_loop/packed/rfdaa_?_packed_1_1.pdb \
    -extra_res_fa ../inputs/TS.params \
    -nstruct 2
```

Note that for time purposes we only relax a portion of the structures.

Analysis of relaxed structures

The scorefile output by rosetta_scripts (`relax_results.tab`) can be opened with any program which inteprets tabular data. (This includes things like Pandas in Python scripting.) For this tutorial, we can use loffice spreadsheets:

```
loffice --calc relax_results.tab
```

In the dialog box which pops up, pick "Separated by Space" and "Merge delimiters"

Columns to pay particular attention to are the `interface_delta_B` which is the InterfaceScoreCalculator's assessment of the binding energy of chain B (the ligand - RFdiffusionAA changed the designation; more negative is better), as well as the `lig_sasa` column which is the SasaMetric's assessment of how exposed the ligand is (lower SASA values are more buried and thus better). The `rmsd` column measures how much the structure moved during relaxation (lower values mean the design is more stable), and the `lig_rmsd` measures how much the ligand moved (lower is better).

It's also worth pointing out that the XML provided here is somewhat arbitrary. Depending on your design goals, you can add additional SimpleMetrics to help evaluate the structure. The FastRelax step is also optional, and the metrics can be run directly on the input structures.

Refolding

One approach which has been shown to work quite well to improve the success rate of designs is the concept of "forward folding". That is, after using a particular structure to design a sequence, can you recapitulate the structure from that sequence? The concept is that under experimental conditions, the only information you're providing is the input protein sequence. You're therefore looking for designs which can robustly produce the desired structure from that sequence.

The current way of doing this is to use an ML structure prediction approach such as AlphaFold, OpenFold or RoseTTAFold to take the designed sequence (e.g. from the FASTA files in `mpnn_loop/seqs`) and predict the structure. Good designs will result in predicted structures which match the starting structure. Designs which aren't confidently folded to the desired structure are likely of poor quality. While they may work, it's often best to go back to the design process and repeat the design and filtering to look for sequences which do reliably refold to the desired structure.

Note that it's often worth using orthogonal prediction approaches. RFdiffusion is based on RoseTTAFold, so there's a chance that RFdiffusion produces backbone structures which RoseTTAFold just happens to like, but which don't actually fold all that well. By using an orthogonal prediction approach (e.g. AlphaFold or OpenFold), you can minimize the potential that you've hit a blind spot in any one technique. With design, you don't have to stick with the one sequence - you can generate a large number of possible designs, and only experimentally test those designs which different techniques have a consensus on.

```
# Use the techniques from the previous tutorials to set up a refolding run for one of the designs.
```

Inserting the ligand

Current ML folding approaches aren't suited for refolding with ligands. While AlphaFold3 and RoseTTAFoldAA can theoretically handle ligands, the public version of AlphaFold3 does not allow use of arbitrary ligands (only a small number of pre-selected ones). While RoseTTAFoldAA is publicly available and does support arbitrary ligands, given the fact that RFdiffusionAA is based on RoseTTAFoldAA, its ability to independently verify the ligand positioning may be lacking.

If you have an apo structure from refolding, and wish to see if the ligand will robustly bind in the location where it has been designed to, it is helpful to use a ligand docking protocols to see if the refolded binding pocket is structured properly to support ligand binding.

```
# If you're interested in trying ligand redocking, take a look at the RosettaLigand small molecule docking
```

Final notes

In this tutorial a number of different validation and analysis approaches were discussed. Not all approaches would be necessary for all design protocols, and depending on what you want in your design, you may need to add other filters and analysis steps. A design pipeline is often an iterative approach, strengthening the analysis and filtering to address pathologies seen during the design process or loosening the filtering criteria to let sufficient number of designs through. You'll need to adjust the balance such that you get designs you're happy with.