4) De novo model-building guided by experimental density data

In this scenario, we introduce a tool, *denovo_density*, aimed at automatically building backbone and placing sequence in 3-4.5 Å cryoEM density maps. This tool is primarily intended for cases where a model is to be built with no known structural homologues. It is relatively expensive computationally, and consists of four basic steps:

- Search for local backbone "fragments" in the density map
- Score the "compatability" of sets of placed fragments
- Monte Carlo sampling for the "maximally compatable" fragment set
- Consensus assignment from the best-scoring Monte Carlo trajectories

The **input** of the method is a segmented density map, and the sequence contained in this region. The output of the method is a "partial model" that – ideally – places 70-80% of the sequence into the density map. This partial model can then be used as input for RosettaCM (see section 3 of the tutorial).

This method is under constant development. Current limitations of the approach – hopefully to be addressed in future revisions – include:

- The code assumes segmented density maps. It cannot handle symmetry, and poorly handles unsegmented density maps. Results are best when the map is segments to only contain one copy of the residues getting built.
- It has only been tested building proteins ~600 residues or less. While it should conceptually scale to larger proteins, this is untested. Furthermore, with larger proteins, the memory usage of steps 2 and 3 increases significantly, so care must be taken.
- It currently does not identify and build ligands or nucleic acids

This section of the tutorial walks through these four steps of the protocol. As a running example, we again use the 20S proteasome (Xueming Li *et al.*, *Nature Methods*, 2013), in this case using a 4.8Å reconstruction determined without using motion correction. We will pretend that known homologous structures are not available, and instead will build models into density denovo.

Input file preparation: download fragment files from Robetta

Before running the method, a user must first create a "fragment file" that predicts local backbone conformations given the amino-acid sequence. The easiest way to do so is to submit your sequence at http://robetta.bakerlab.org/.

Alternately, the Rosetta users guide describes how fragment files may be builtlocally : (https://www.rosettacommons.org/manuals/archive/rosetta3.5_user_guide/dc/d10/app_fragment_picker.html)

Step 4A. Local fragment search

In the first part of the procedure, we search the density map for each sequence-predicted backbone fragment. This part, like all the steps in this section, uses a Rosetta application *denovo_density*.

The command to run fragment searching for a single residue (*4_denovo_demo/A_search.sh*):

```
$ROSETTA3/source/bin/denovo_density.macosclangrelease \
       -in::file::fasta t20sA.fasta \
       -fragfile ./t001_.25.9mers \
       -mapfile ./T20S_48A_alpha_chainA.mrc \
       -n_to_search 500 -n_filtered 2500 -n_output 100 \
       -bw 16 \
       -atom_mask_min 2 \
       -atom_mask 3 \
       -clust_radius 3 \
       -clust_oversample 4 \
       -point_radius 3 \
       -movestep 1 \
       -delR 2 \
       -frag_dens 0.8 \
       -ncyc 3 \
       -min_bb false \
       -pos $1 \
       -out:file:silent round1/t20s.$1.silent
```

Most of the arguments shown here should be left as-is. However, there are a few – highlighted above in boldface – that you might want to change:

```
-n_to_search 500 -n_filtered 2500
```

These flags control the number of translations to search, and the number of intermediate solutions to keep. As a rule of thumb, these should be about **2 and 10 times the number of residues in the map, respectively**.

Make note of the following flag:

```
-pos $1
```

This flag tells the code to only search for fragments at the assigned positions. This allows for parallelization of the script, by running separate jobs for each position in the protein. ($1 means the script takes the position as an input argument).

For this step, you need to run the script once for each position in the protein. This can be done very simply with the bash command (for this case, the 221 indicates there are 221 residues in the protein):

```
for i in `seq 1 221`; do ./A_search.sh $i; done
```

The output of this script is a single file for each position in the protein, that identifies the placement and configuration of each docked fragment. These files are used as input for the next step of the process.

However, as each position runs independently, and each position might take 30 minutes to an hour for the search, you will probably want to parallelize this over many processors.

Job distribution

As this is the most computationally intensive step, it makes sense to parallelize this step, by run this procedure separately for each position in the protein. Using GNU parallel.

```
parallel –j16 ./A_search.sh {} ::: {1..221}
```

GNU parallel allows launching of jobs remotely if SSH keys have been set up for passwordless login. To run:

```
parallel -S 16/node1,16/node2,16/node3,16/node4 --workdir . ./A_search.sh {} ::: {1..221}
```

This will launch instead 48 jobs spread across four machines. See the GNU parallel documentation for more information.

Other useful options

There are two options controlling fragment placement that may also be useful in cases where there is some previous knowledge about the structure in question. The first of these deals with the case where the backbone structure is known (or at least somewhat known) but registration of the sequence with the backbone model is ambiguous. In this case, a known backbone model can be provided with the flag:

```
-ca_positions backbone.pdb
```

In this case, the code will only consider fragments centered on the C alpha positions from the input model. This offers a significant speedup as well as reduced search space.

Alternately, if part of the structure is known in advance, it may be provided with the flag:

```
-startmodel start.pdb
```

In this case, the matching routine will match only the native fragments covered by start model, ensuring that these positions will be maintained throughout the refinement. When using this options there are two important things to keep in mind:

- The numbering in the PDB file **must** match the numbering of the input fasta
- Any continuous segments **shorter than 9 amino acids** in the input file will get ignored

Step 4B. Placed fragment scoring

In this step, we want to take the placements from the previous step and score them for compatability. The outputs from step A are used as inputs in this step (*4_denovo_demo/B_score.sh*):

```
$ROSETTA3/source/bin/denovo_density.macosclangrelease \
      -mode score \
      -in::file::silent round1/t20s*silent \
      -scorefile round1/scores1 \
      -n_matches 50
```

Highlighted in bold are the input files (-in::file::silent) – the outputs from the previous step – and the score file to be written (-scorefile), the output of this step. If the output is written to a separate folder, you will need to point the command line to this alternate location.

This step is relatively fast (less than 5 minutes) and can be run on a single processor.

Step 4C. Monte Carlo fragment assembly

In the third step, we use the outputs from the previous two steps, and try to generate a "maximally

consistent" fragment assignment. It uses Monte Carlo sampling and a scorefunction assessing fragment compatibility to identify this fragment set. The command line (*4_denovo_demo/C_assemble.sh*):

```
$ROSETTA3/source/bin/denovo_density.macosclangrelease \
      -mode assemble \
      -nstruct 5 \
      -in::file::silent round1/t20s*silent \
      -scorefile round1/scores1 \
      -assembly_weights 4 20 6 \
      -null_weight -150 \
      -out:file:silent round1/assembled.$1 \
      -scale_cycles 1 \
      -mute core
```

As with step B, the outputs of the previous two steps need to be provided as inputs:

```
-in::file::silent round1/t20s*silent
-scorefile round1/scores1
```

The script then writes a single file for each independent trajectory:

```
-out:file:silent round1/assembled.$1
```

Finally, each job will generate several (in this case 5) independent trajectories:

```
-nstruct 5
```

It is recommended to generate a total of 1000 independent trajectories. As with step A, this can be somewhat time-consuming (though not as time-consuming as step A). Therefore it is recommended to parallelize this as before:

```
parallel -j16 ./C_assemble.sh {} ::: {1..200}
```

Or across multiple machines:

```
parallel -S 16/node1,16/node2,16/node3,16/node4 --workdir . ./C_assemble.sh {} ::: {1..200}
```

Step 4D. Consensus assignment

The final step of the protocol is to identify the consensus assignment from the lowest-scoring Monte Carlo trajectories. This is done using the following command (*4_denovo_demo/D_consensus.sh*):

```
$ROSETTA3/source/bin/denovo_density.macosclangrelease \
      -mode consensus \
      -in::file::silent round1/assembled.*silent \
      -consensus_frac 1.0 -energy_cut 0.05 \
      -mute core
```

The output trajectories of the previous step are provided as input to this script with –in::file::silent. This script looks for a consensus assignment in the best-scoring trajectories, and will output a PDB file, **S_0001.pdb**.

This PDB file contains sequence placed into density. Ideally, the model at this point is more than 70% complete, and this file can then be used as input to RosettaCM (see section 3). If instead this structure contains a reasonable partial model, but with less than 70% coverage, the iterative approach of the next section can further improve the coverage of the partial model.

Step 4E. Iterative assembly to increase model coverage

In some cases, it may be necessary to iterate refinement, as subsequent rounds of denovo building may trace portions of the model unable to be placed in previous rounds. The following command line illustrates how assembly may be iterated (*4_denovo_demo/E_search_iter.sh*):

```
$ROSETTA3/source/bin/denovo_density.macosclangrelease \
       -in::file::fasta t20sA.fasta \
       -fragfile ./t001_.25.9mers \
       -startmodel round1_model.pdb \
       -mapfile ./T20S_48A_alpha_chainA.mrc \
       -n_to_search 250 -n_filtered 1250 -n_output 50 \
       -bw 16 \
       -atom_mask_min 2 \
       -atom_mask 3 \
       -clust_radius 3 \
       -clust_oversample 4 \
       -point_radius 3 \
       -movestep 1 \
       -delR 2 \
       -frag_dens 0.8 \
       -ncyc 3 \
       -min_bb false \
       -pos $1 \
       -out:file:silent round2/t20s.$1.silent
```

This step is nearly identical to step A, with two key chages highlighted in bold. The first change indicates that the output of the previous step is to be used as an initial model:

```
-startmodel round1_model.pdb
```

Following inspection, this model may also be manually edited as well.

The second change makes sure the outputs don't clobber the outputs from the previous step:

```
-out:file:silent t20s.rd2.$1.silent
```

As with step A, this should be parallelized over many processors, e.g, using GNU parallel:

```
parallel -S 16/node1,16/node2,16/node3,16/node4 --workdir . ./ E_search_iter.sh {} ::: {1..221}
```

Once complete, steps B,C, and D may be then followed in order, making sure that the input and output files are updated to indicate the round 2 models.

NOTE: At each step, be certain that your outputs are not overwriting one another. Once the partial model has been calculated, it is safe to delete the intermediate files created as part of the construction process. In this case, the same output file names may be reused (thus, the same scripts can be used for each iteration aside from the first).