Tutorial: Rosetta tools for structure determination in cryoEM density

Brandon Frenz, Ray Y.-R. Wang, Frank DiMaio
Last updated: March 2019

This tutorial is intended to introduce users to several different ways Rosetta may be used to solve various structure determination tasks given 3-5Å cryoEM density data. It is not intended to replace the user's guide, available at https://www.rosettacommons.org/manuals/latest/main/.

The tutorial is split up into four parts.

1. An **introduction to Rosetta** in general, showing how one may score structures and minimize structures guided by experimental density data
2. Our **model rebuilding protocol** (RosettaCM), where one wishes to recombine homologous structures, and rebuild *small* missing regions (<12 residues)
3. An **advanced application of RosettaCM** to determine the sequence threading of a model
4. Our **model completion tools**, where one wishes to complete a partial model built by the de novo tool *or* wishes to rebuild *large* missing regions (12 or more residues)

In each scenario, we present the most basic usage of Rosetta for the task, and then describe additional options that may be useful. Command-line flags and input scripts are provided in shaded boxes, with boldfaced text indicating parameters of note. These parameters are described in the text following the command line.

Note: in all sections, you will need to update the command scripts to point at your installation of Rosetta and the Rosetta database.

1) Rosetta and electron density basics

This section provides a brief introduction to using Rosetta, and an overview of using density data within Rosetta.

Overview of Rosetta

The Rosetta documentation is a good source of additional information on several of the tools described in this document. This is available at https://www.rosettacommons.org/docs/latest/Home.

The tools described in this document use the RosettaScripts framework, described at https://www.rosettacommons.org/docs/latest/scripting_documentation/RosettaScripts/RosettaScripts. Briefly, this allows protocols to be defined as a series of atomic "Movers" which manipulate a structure. The format is as follows:

```
<ROSETTASCRIPTS>
    <SCOREFXNS>
    </SCOREFXNS>
    <MOVERS>
    </MOVERS>
    <PROTOCOLS>
    </PROTOCOLS>
</ROSETTASCRIPTS>
```

Each block contains information in running the protocol: <SCOREFXNS> and <MOVERS> are used to *declare* score functions and movers; while <PROTOCOLS> is where the steps of the protocol are enumerated.

Rosetta tools are run via the command line, with flags controlling general program behavior. Many of the flags specifically for density refinement are outlined in the sections following.

Density scoring in Rosetta

Agreement to density is implemented in Rosetta as an additional energy term. Rosetta assesses agreement to density by computing the density that one would expect to see, given a model, and measuring the agreement of the expected and experimental density.

elec_dens_fast
This scoreterm is recommended for nearly all uses of density refinement in Rosetta. It uses interpolation on a precomputed grid of per-atom scores to approximate the density correlations. This version is significantly faster (~10x) then the exact scoring term below, and is very highly correlated.

These energy terms may be provided to Rosetta in two ways. First, it may be provided in a RosettaScript XML file as input:

```
<Reweight scoretype="elec_dens_fast" weight="35.0"/>
```

For non-Rosetta script applications, the following flag controls the density scoring function weight:

```
-edensity:fast_dens_wt 35.0
```

The recommended weights for each of these terms vary depending on the density map resolution, starting model quality, and protocol. Section 2 describes how the weights may be tuned. However, the following are

good rules of thumb for setting the density weight within Rosetta:

At resolutions better than 2.5Å: an *elec_dens_fast* weight of **65.0** is generally reasonable.
At resolutions between 2.5Å and 3.5Å: an *elec_dens_fast* weight of **50.0** is generally reasonable.
At resolutions worse than 3.5Å: an *elec_dens_fast* weight of **35.0** is generally reasonable.
In *centroid mode*: an *elec_dens_fast* weight of **10.0** is generally reasonable

At very low resolutions (worse than 6Å), the weight may need to be further reduced. In general, if the Rosetta energies are positive (or significant outliers are flagged by Molprobity or other validation programs) then the weights need to be reduced.

In addition to the score terms above, there are also several flags that control map scoring behavior. Maps are read into Rosetta using either the flag:

```
-edensity::mapfile mapfile.mrc
```

Or from XML:

```
<LoadDensityMap name="loaddens" mapfile="mapfile.mrc"/>
```

Maps may be in either CCP4 or MRC format (the map type is automatically detected from the header info).

The resolution of the map, used when comparing calculated to experimental density, is specified with the flag:

```
-edensity::mapreso 5.0
```

Maps may also be resampled to reduce memory usage and runtime. This is done through the flag:

```
-edensity::grid_spacing 2.0
```

Notice that this flag should *never be more than half the given resolution*, and if using the fast scoring function *never more than a third of the resolution*. For both parameters, **the default is generally fine** (don't resample, and assume the resolution is ~3x the grid sampling).

Finally, one may choose to calculate density using either cryoEM or X-ray scattering factors. At low resolution, this probably makes little difference, but might at resolutions better than about 3.5Å. The default is to use X-ray scattering factors; to turn on cryoEM scattering factors instead, use the following flag:

```
-edensity::cryoem_scatterers
```

Example 1A: Scoring a PDB in Rosetta with density

Most simply, one may wish to simply score a model using Rosetta's energy function including the density terms. This is easily accomplished using the *score_jd2* application. A sample command line to rescore the structure in density is given in *1_rosetta_basics/A_run_rescore.sh*. It illustrates the use of various density flags to provide Rosetta with experimental density information.

```
$ROSETTA3/source/bin/score_jd2.macosclangrelease \
  -database $ROSETTA3/database/ \
  -in::file::s 1isrA.pdb 1issA.pdb \
  -ignore_unrecognized_res \
  -edensity::mapfile 1issA_6A.mrc \
  -edensity::mapreso 5.0 \
  -edensity::grid_spacing 2.0 \
  -edensity::fastdens_wt 35.0 \
  -edensity::cryoem_scatterers \
  -crystal_refine
```

Some flags of note are boldfaced above. First, the input structure is provided with the command -in::file::s. **This is common to many Rosetta applications, and more than one input may be provided; each will be processed independently**. The flags beginning with –edensity:: tell Rosetta about the density map into which it is being fit. The name of the mapfile (in CCP4 or MRC format), the resolution of the map, the grid sampling of the map (*which should never be more than half the resolution*), and the weights on the various fit-to-density scoring functions. These same flags are reused for many different protocols in addition to relax. Finally, the flag -crystal_refine the flag turns on several density-related options related to PDB reading and writing, and should always be used when refining against density data.

Note: The input PDB must be aligned to the density map using some external tool. Rosetta will optionally rigid-body minimize the structure into density before rescoring by providing the flag –edensity::realign min to the application. If this is done, the flag –out::pdb will write the minimized PDB file to a PDB file.

This command line outputs a score file, *score.sc,* that gives, for each structure specified with -in::file::s, the score with respect to each term in Rosetta's energy function. The meaning of individual scoreterms as well as an overview of the Rosetta energy function can be found in the paper:

Alford RF, Leaver-Fay A, Jeliazkov JR, O'Meara MJ, DiMaio FP, Park H, Shapovalov MV, Renfrew PD, Mulligan VK, Kappel K, Labonte JW, Pacella MS, Bonneau R, Bradley P, Dunbrack RL Jr, Das R, Baker D, Kuhlman B, Kortemme T, Gray JJ. The Rosetta All-Atom Energy Function for Macromolecular Modeling and Design. *J Chem Theory Comput.* 2017 Jun 13;13(6):3031-3048.

Examples 1B and 1C: Simple refinement into density using RosettaScripts and relax

In this section we introduce RosettaScripts by way of a very simple refinement-into-density example. **RosettaScripts provides an XML scripting interface to Rosetta that allows fine-grained control of protocols.** The syntax is fully described in the Rosetta documentation; however, a very brief introduction is provided here. The basic syntax for the XML is illustrated here (*1_rosetta_basics/B_relax_density.xml*)

```
<ROSETTASCRIPTS>
   <SCOREFXNS>
      <ScoreFunction name="dens" weights="beta_cart">
         <Reweight scoretype="elec_dens_fast" weight="35.0"/>
         <Set scale_sc_dens_byres="R:0.76,K:0.76,E:0.76,D:0.76,M:0.76,
            C:0.81,Q:0.81,H:0.81,N:0.81,T:0.81,S:0.81,Y:0.88,W:0.88,
            A:0.88,F:0.88,P:0.88,I:0.88,L:0.88,V:0.88"/>
      </ScoreFunction>
   </SCOREFXNS>
   <MOVERS>
       <SetupForDensityScoring name="setupdens"/>
       <LoadDensityMap name="loaddens" mapfile="1issA_6A.mrc"/>
       <FastRelax name="relaxcart" scorefxn="dens" repeats="2" cartesian="1"/>
   </MOVERS>
```

```
   <PROTOCOLS>
      <Add mover="setupdens"/>
      <Add mover="loaddens"/>
      <Add mover="relaxcart"/>
   </PROTOCOLS>
   <OUTPUT scorefxn="dens"/>
</ROSETTASCRIPTS>
```

There are three "blocks" of declarations in this script. In the first, <SCOREFXNS> … </SCOREFXNS>, the scorefunctions to be used throughout the protocol are declared; the second, <MOVERS> … </MOVERS>, movers – or atomic operations that modify a structure – are declared; finally, the third, <PROTOCOLS> … </PROTOCOLS>, a full protocol is declared as a sequence of movers.

In this particular example, we declare a single scorefunction, *dens*, which uses the score function *beta_cart* (a default score function, don't need to worry about it), and turns on *elec_dens_fast*, the fit-to-density score, with a weight of 35. We then declare three movers, *SetupForDensityScoring*, *LoadDensityMap*, and *FastRelax*, which sets up the loaded structure for density scoring, loads a map into memory, and then refines the structure using the *FastRelax* protocol. The declared scorefunction, *dens*, is used as an input to the *FastRelax* mover.

Finally, note the additional block:

```
<Set scale_sc_dens_byres="R:0.76,K:0.76,E:0.76,D:0.76,M:0.76,
           C:0.81,Q:0.81,H:0.81,N:0.81,T:0.81,S:0.81,Y:0.88,W:0.88,
           A:0.88,F:0.88,P:0.88,I:0.88,L:0.88,V:0.88"/>
```

This adjusts the per-residue sidechain density weights. It is recommended to always use these weights when refining against cryoEM density.

To run this script, we use the following command line (*1_rosetta_basics/B_relax_density.sh)*:

```
$ROSETTA3/source/bin/rosetta_scripts.macosclangrelease \
  -database $ROSETTA3/database/ \
  -in::file::s 1isrA.pdb \
  -parser::protocol ex_B1_run_RS_relax_density.xml \
  -ignore_unrecognized_res \
  -edensity::mapreso 5.0 \
  -edensity::cryoem_scatterers \
  -crystal_refine \
  -out::suffix _relax \
  -beta
```

Note: We do not have to specify the density weight or the map file on the command line, since they are handled within the XML file. However, other density options must be specified on the command line. **When using RosettaScripts, the density weights *must* be specified in the XML, the input map may be specified *either way*.**

Finally, in the previous XML file, the tag ***cartesian=1*** appears, *which refines the structure in Cartesian space*. Rosetta also allows refinement in torsional space, which may be better for capturing domain motion, and for further reduction in model parameters against low-resolution data. *To enable torsional refinement* (*1_rosetta_basics/C_relax_tors_density.xml*), we make three small changes to the XML:

```
<ROSETTASCRIPTS>
   <SCOREFXNS>
      <ScoreFunction name="dens" weights="beta">
         <Reweight scoretype="elec_dens_fast" weight="35.0"/>
         <Set scale_sc_dens_byres="R:0.76,K:0.76,E:0.76,D:0.76,M:0.76,
```

```
            C:0.81,Q:0.81,H:0.81,N:0.81,T:0.81,S:0.81,Y:0.88,W:0.88,
            A:0.88,F:0.88,P:0.88,I:0.88,L:0.88,V:0.88"/>
      </ScoreFunction>
   </SCOREFXNS>
   <MOVERS>
       <SetupForDensityScoring name="setupdens"/>
       <LoadDensityMap name="loaddens" mapfile="1issA_6A.mrc"/>
       <FastRelax name="relaxcart" scorefxn="dens" repeats="5" cartesian="0"/>
   </MOVERS>
   <PROTOCOLS>
      <Add mover="setupdens"/>
      <Add mover="loaddens"/>
      <Add mover="relaxcart"/>
   </PROTOCOLS>
   <OUTPUT scorefxn="dens"/>
</ROSETTASCRIPTS>
```

Cartesian versus torsional refinement

One of the strengths of Rosetta is its ability to perform torsion-space refinement, which can be incredibly valuable in capturing things like domain motion of proteins which are simple moves in torsion space but can be complex in cartesian space. The optimal type of refinement for a particular problem depends on the system itself, the map resolution, and the quality of the starting model. A few general tips:

- Several (2-4) repeats of torsion-space refinement followed by 1 repeat of Cartesian-space refinement is generally a good strategy
- For very large (1000 residue+) systems *or* very poor quality input models (many clashes) cartesian refinement alone is better behaved.

2) Model rebuilding with RosettaCM

In this scenario, we introduce a tool, RosettaCM, for building missing portions of a model guided by density data. While primarily geared towards comparative modeling, it may also be useful for building portions of a protein that are disordered when crystallized or difficult regions in hand-built models. In this scenario, we introduce the basic rebuilding protocol, then show how the tool may also be used to:

- Combine pieces from multiple template models guided by density
- Rebuild with user-defined restraints
- Iteratively rebuild models in difficult cases difficult cases

As a running example, we use horse spleen apoferritin and the deposited map (*EMD-2788*).

Example 2A: Preparing templates for use in RosettaCM

In many cases, much of the setup work is handled by a script, *setup_RosettaCM.py* in RosettaTools (a separate repository available from rosettacommons.org). This script takes an input alignment in a variety of formats, and prepares the inputs automatically. It is executed by running the command:

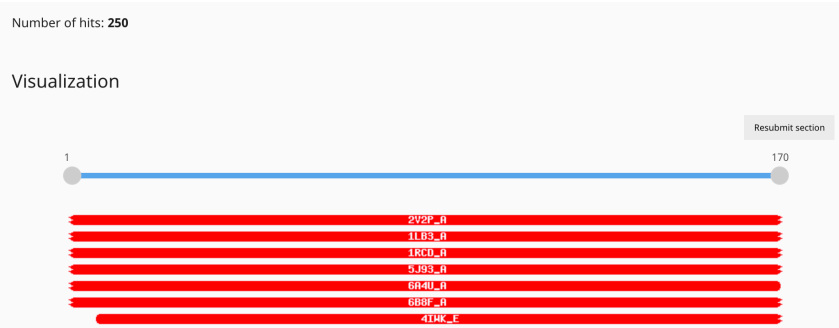
```
setup_RosettaCM.py \
--fasta t20s.fasta \
--alignment tmpl.fasta \
--alignment_format fasta \
--templates tmpl.pdb \
--rosetta_bin ~/Rosetta/main/source/bin \
--verbose
```

Inputs include the full-length fasta, an alignment file – in either fasta, ClustalW, or HHSearch format – and the corresponding template PDB files. This script will prepare all the necessary inputs in order to run RosettaCM.

Alternately, the setup may be performed manually. In this case, since we are using some nonstandard features (symmetry and density), and we have two chains in the asymmetric unit we will do this; alternately, the inputs from the previous step may be used as a starting point and subsequently modified.

Identifying alignments with hhpred

We first need to identify homologous sequences. To do this, we use the webserver hhpred (https://toolkit.tuebingen.mpg.de/). We enter our sequence (*seg.fasta*) into the web form and click submit. We get results:



In this case, there are many homologous

We need to convert this alignment to a format Rosetta can understand. I have included a script (*scripts/ prepare_hybridize_from_hhsearch.pl*) that automates this although it may be performed manually with a text editor as well.

Download the alignment by clicking "Raw Output" and then "Download" (or see the tutorial file *seq.hhr*).

Most of these hits are very high sequence identity, making the modelling problem trivial. We are going to focus on modelling starting from two distant structures of bacterioferritins:

```
…
60 3UOI_V Bacterioferritin (E.C.1  99.7   9E-18 1.9E-22  114.8  19.5  156    3-168     1-156
…
62 3GVY_C Bacterioferritin; bacte  99.7 3.6E-17 7.5E-22  112.0  19.3  154    6-169     2-155
…
```

Using a text editor, edit the file *seq.hhr*, removing all but these two alignments (or see the file *seq_edit.hhr*).

Next, convert these alignments to Rosetta format using the given script (*A_convert_hhr_file.sh*). In addition to converting the alignment file, it will also download the template files necessary for the next step. Run this script without input arguments, and an output, *alignment.filt*, is produced:

```
## 1XXX_ 3uoiV_201
# hhsearch
scores_from_program: 0 1.00
2
IRQNYSTEVEAAVNRLVNLYLRASYTYLSLGFYFDRDDVALEGVCHFFRELAEEKREGAERLLKMQNQRGGRALFQDLQKPSQDEWGTTLDAMK
AAIVLEKSLNQALLDLHALGSAQADPHLCDFLESHFLDEEVKLIKKMGDHLTNIQRLVGSQAGLGEYLFERL
0 --MQGDPDVLRLLNEQLTSELTAINQYFLHSKMQDN--WGFTELAAHTRAESFDEMRHAEEITDRILLLDGLPNYQRIGSLRI--
GQTLREQFEADLAIEYDVLNRLKPGIVMCREKQDTTSAVLLE-KIVADEEEHIDYLETQLELMDK-----LGEELYSAQCV
--
## 1XXX_ 3gvyC_202
# hhsearch
scores_from_program: 0 1.00
5
NYSTEVEAAVNRLVNLYLRASYTYLSLGFYFDRDDVALEGVCHFFRELAEEKREGAERLLKMQNQRGGRALFQDLQKPSQDEWGTTLDAMKAAI
VLEKSLNQALLDLHALGSAQADPHLCDFLESHFLDEEVKLIKKMGDHLTNIQRLVGSQAGLGEYLFERLT
0 QGDAKVIEYLNAALRSELTAVSQYWLHYRLQED--WGFGSIAHKSRKESIEEMHHADKLIQRIIFLGGHPNLQRLNPLRI--
GQTLRETLDADLAAEHDARTLYIEARDHCEKVRDYPSKMLFE-ELIADEEGHIDYLETQIDLMGS-----IGEQNYGMLNAK
--
```

In this format, the first line is '##' followed by a code for the target and one for the template. The second line identifies the source of the alignment; the third just keep as it is. The fourth line is the target sequence and the fifth is the template; the number is an 'offset', identifying where the sequence starts. However, the number doesn't use the PDB resid but just counts residues *starting at 0*. The sixth line is '--'. Multiple alignments may be concatenated in a single file, with the template code identifying the template corresponding to each alignment.

Example 2B: Run partial threading and dock models into density

RosettaCM takes as inputs *partially threaded* models, that is models where aligned positions have their residue identities remapped, and unaligned residues are not present. To generate these models from an alignment file and template, we can run the Rosetta command (*3_model_rebuilding/A_partialthread.sh*):

```
$ ROSETTA3/source/bin/partial_thread.macosclangrelease \
 -database ~/Rosetta/main/database/ \
 -in::file::fasta seq.fasta \
 -in::file::alignment alignments.filt \
 -in::file::template_pdb 3uoiV.pdb 3gvyC.pdb pdb
```

This will output a two partially threaded models – *3uoiV_201.pdb* and *3gvyC_202.pdb* – that will be used as input for RosettaCM.

The final step of the method is to align the partially threaded models into the density map. This can be done most easily using Chimera's "fit into map" tool. It may be easiest to align one partial thread into the density and then align the other model to that. Aligned versions of the templates are included as *3uoiV_201_aln.pdb* and *3gvyC_202_aln.pdb*.

Example 2C: Running RosettaCM as a monomer.

For our first step, we will be modelling the monomer structure using RosettaCM. While the assembly is symmetric, and the next part will be carried out in the context of the assembly, it may be useful in some cases to model individual components of larger assemblies. Such modelling is much faster, allowing for much greater conformational sampling, and it is often useful to model individual subunits before modelling the entire complex.

Like the methods introduced in Scenario 1, RosettaCM is controlled through an XML script using RosettaScripts. The XML is as follows (*2_model_rebuilding/C_rosettaCM_singletarget.xml)*:

```
<ROSETTASCRIPTS>
    <TASKOPERATIONS>
    </TASKOPERATIONS>
    <SCOREFXNS>
        <ScoreFunction name="stage1" weights="score3" symmetric="1">
            <Reweight scoretype="atom_pair_constraint" weight="0.1"/>
            <Reweight scoretype="elec_dens_fast" weight="10"/>
        </ScoreFunction>
        <ScoreFunction name="stage2" weights="score4_smooth_cart" symmetric="1">
            <Reweight scoretype="atom_pair_constraint" weight="0.1"/>
            <Reweight scoretype="elec_dens_fast" weight="10"/>
        </ScoreFunction>
        <ScoreFunction name="fullatom" weights="beta_cart" symmetric="1">
            <Reweight scoretype="atom_pair_constraint" weight="0.1"/>
            <Reweight scoretype="elec_dens_fast" weight="35"/>
                <Set scale_sc_dens_byres="R:0.76,K:0.76,E:0.76,D:0.76,M:0.76,
                    C:0.81,Q:0.81,H:0.81,N:0.81,T:0.81,S:0.81,Y:0.88,W:0.88,
                    A:0.88,F:0.88,P:0.88,I:0.88,L:0.88,V:0.88"/>
        </ScoreFunction>
    </SCOREFXNS>
    <FILTERS>
    </FILTERS>
    <MOVERS>
        <Hybridize name="hybridize" stage1_scorefxn="stage1" stage2_scorefxn="stage2"
                   fa_scorefxn="fullatom" batch="1">
            <Template pdb="3uoiV_201_aln.pdb" weight="1.0" cst_file="AUTO"/>
            <Template pdb="3gvyC_202_aln.pdb" weight="1.0" cst_file="AUTO"/>
        </Hybridize>
    </MOVERS>
    <PROTOCOLS>
        <Add mover="hybridize"/>
    </PROTOCOLS>
</ROSETTASCRIPTS>
```

The main work is done through a single mover, *Hybridize* which handles all stages of model-building. Input

structures are specified via *Template* lines (in this case there is only one). For each template line, we specify the pdb input, as well as a couple of other parameters: a *weight* (the relative frequency we sample each template with); a *constraint file* (setting this to "auto" sets up automatic constraints to the template, while setting this to "none" turns off all constraints, user-defined constraints are described later).

A few notes about using multiple models with hybridize:

- With density, we need to **ensure that all input models are aligned to the density**. This can be done using **Chimera's alignment tools**. It may be easier to align a single model to the density and then align all other models to this model.
- In each trajectory, a starting model is chosen at random; the constraints and symmetry from this selected model are chosen at the start of each run. **If we wish to use a portion of a model, but do not want to use its symmetry or constraints, we can assign it a weight of 0**: backbone conformations from this model will be used in conformational sampling, but the symmetry and constraints will never be used.
- Similarly, gaps in the selected starting model are rebuilt before recombination occurs. **If one of the templates has poor coverage, but provides valuable structural features, it should be used, but with weight 0.**

Given this XML, RosettaCM is then run with the following command line (*C_rosettaCM_singletarget.sh)*:

```
$ROSETTA3/source/bin/rosetta_scripts.macosclangrelease \
 -database $ROSETTA3/database/ \
 -in:file:fasta t20s.fasta \
 -parser:protocol C_rosettaCM_singletarget.xml \
 -nstruct 5 \
 -relax:jump_move true \
 -relax:dualspace \
 -out::suffix _singletgt \
 -edensity::mapfile t20S_41A_half1.mrc \
 -edensity::mapreso 5.0 \
 -edensity::cryoem_scatterers \
 -beta \
 -default_max_cycles 200
```

The input command is similar to those seen before, but with a few key differences. First, the input to Rosetta is specified with `-in:file:fasta` rather than `-in:file:s`. Also note that the input argument `-nstruct 5` is given, telling Rosetta to generate 50 models for each process. Generally, *hundreds to thousands of models* are necessary to sufficiently sample conformational space; more and longer regions to rebuild require more models.

Job distribution

It is generally useful to sample ~100 models from each starting point. For this purpose, it may be useful to run multiple jobs in parallel. To prevent output structures from clobbering one another, the flag –out::suffix may be useful, where each separate job is given a different suffix.

For example, on a 16-core machine, we may specify -out::suffix _$1, then (using GNU parallel) run the following:

```
parallel –j16 ./ C_rosettaCM_monomer.sh {} ::: {1..16}
```

Finally, 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 . ./C_rosettaCM_monomer.sh {} ::: {1..48}
```

This will launch instead 48 jobs spread across four machines. See the GNU parallel documentation (https://www.gnu.org/software/parallel/) for more information.

Analyzing results and model selection

While this is an active topic of research, generally – once a density weight has been chosen – to select the best models from among the full set, we want to select models optimizing both model geometry and fit-to-density values. Model geometry may be evaluated using Rosetta energies after subtracting density energies, which may be done by inspecting the score*.sc files produced as output. Density fit may be evaluated using the density energy in Rosetta as well as FSCs using the ReportFSC mover (not covered in this tutorial, see part 2 of the main tutorial)

No matter the selection criteria, the top models (5-10) should be inspected for model convergence as well as visually inspected for density map agreement.

Example 2D: Running RosettaCM with symmetry.

Next, we need to set up symmetric modeling with RosettaCM. We use a script, *make_symmdef_file.pl* script in order to generate a symmetry definition file for use in Rosetta. A straightforward way to do so is to use Chimera to dock the necessary chains into density. This script's required inputs depend on the underlying symmetry:

- For cyclic (*C)* and dihedral (*D)* symmetries, we only need a single "primary chain" and an adjacent chain in each point group;
- For helical symmetries, we need an adjacent chain in the layer (if there is one) and an adjacent chain up the helical axis
- For other symmetries we need all chains adjacent to a single subunit.

Since this case falls into the latter case, (for examples with *C* and *D* symmetry see the main tutorial), we need to create a PDB file that contains one chain plus all adjacent chains docked into density. An example, *3gvyC_symm_r.pdb,* is included.

Chimera can be useful here as well. From within chimera, we can run the following command to generate symmetry for this case:

```
sym #1 group O center 88.9,88.9,88.9
```

Either way, save the chimera files in a *single output file* and then relabel chains using the included script:

```
scripts/relabel_chains.pl 3gvyC_symm.pdb
```

To generate our Rosetta symmetry file from this input, we then simply have to run the command (*D_make_symmdef.sh)*:

```
$ROSETTA3/source/src/apps/public/symmetry/make_symmdef_file.pl \
  -m pseudo -a A \
  -p 3gvyC_symm_r.pdb > ferritin.symm
```

Since we have already created the input templates using the *partial_thread* application, we simply need to

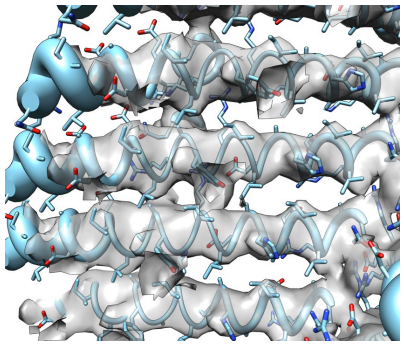
use the output of the partial threading together with the symmetry definition file.

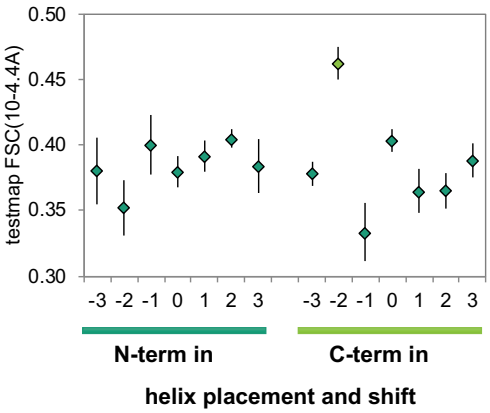
We then need to make two small modifications to our inputs:

```
...
        <Hybridize name="hybridize" stage1_scorefxn="stage1" stage2_scorefxn="stage2"
                   fa_scorefxn="fullatom" batch="1”>
            <Template pdb="3uoiV_201_aln.pdb" weight="1.0"
                      cst_file="AUTO" symmdef="ferritin.symm"/>
            <Template pdb="3gvyC_202_aln.pdb" weight="1.0"
                      cst_file="AUTO" symmdef="ferritin.symm"/>
        </Hybridize>
...
```

3) Advanced modelling: using *partial_thread* and *relax* to determine sequence threading

In this section, we will use the same tools introduced in the previous sections to tackle a more challenging problem, determining the alignment of sequence to a backbone model. This is based on Egelman *et al., Structure*, 2015.





For this example we have a map (left) that clearly identifies helices in the density. However, the threading of sequence is ambiguous: it is not known which is the N- and which is the C-terminus, and there are only 24 resolved residues, compared to 29 amino acids in the sequence.

However, since the helix orientations are straightforward, we can brute-force this problem. We create three models:

1. *polyA_symm.pdb*, in which two helices are docked, from which we can get the symmetry definition file
2. *polyA_ctermin.pdb*, a monomer in one orientation
3. *polyA_ntermin.pdb*, a monomer in the other orientation

Example 3A: Build the symmetry definition file.

As in section two, we start by building the symmetry definition file:

```
$ROSETTA3/source/src/apps/public/symmetry/make_symmdef_file.pl \
   -m HELIX -a J -b K \
   -p polyA_symm.pdb -r 1000 -t 8 > h.symm
```

Since we have helical symmetry, some of the options are a bit different. "-m HELIX" specifies we run in helical mode, and the arguments "-a J -b K" indicate the "primary chain" (J) and the chain up the helical axis (K). Finally, the argument "-t 8" indicates how many subunits to generate in each direction.

When running in this mode, note:

- Rosetta outputs a file, *polyA_symm_model_JK.pdb*, of the symmetry it identifies. You should ensure that this makes sense given the map.
- Rosetta outputs the helical parameters inferred from the model, including the helical rise and the subunits per turn. This should match what was determined experimentally.

Running this command produces a symmetry definition file, "h.symm", to be used as input in subsequent steps.

Finally, we need to make a small edit to this file for density refinement. Change:

```
set_dof JUMP_0_0_0 z(2.20334489451302) angle_z
set_dof JUMP_0_0_0_to_com x(17.509223919948)
set_dof JUMP_0_0_0_to_subunit angle_x angle_y angle_z
```

To:

```
set_dof JUMP_0_0_0_to_com x y z
set_dof JUMP_0_0_0_to_subunit angle_x angle_y angle_z
```

That is, delete the first line (which allows refinement of the symmetry operators)

Example 3B: Generate the partial threads.

In this case, we use the partial_thread tool introduced last section to generate all the of different sequence threadings we are going to model. The included script, *scripts/generate_threadings.pl* will be used to generate the input alignment file, though it may also be done manually using a text editor.

The resulting alignment file (*alignment.filt*):

```
## 1XXX ctermin_0
#
scores_from_program: 0.0
0 QARILEADAEILRAYARILEAHAEILRAQ
0 AAAAAAAAAAAAAAAAAAAAAAAA----
--
## 1XXX ntermin_0
#
scores_from_program: 0.0
0 QARILEADAEILRAYARILEAHAEILRAQ
0 AAAAAAAAAAAAAAAAAAAAAAAA----
--
…
## 1XXX ntermin_1
#
scores_from_program: 0.0
0 QARILEADAEILRAYARILEAHAEILRAQ
0 -AAAAAAAAAAAAAAAAAAAAAAAA---
--
…
## 1XXX ntermin_2
#
scores_from_program: 0.0
0 QARILEADAEILRAYARILEAHAEILRAQ
0 --AAAAAAAAAAAAAAAAAAAAAAAA--
--
…
```

The alignment file simply slides the sequence along the input poly-alanine model.

We then run the partial_thread application on this model, producing a total of 10 input models:

```
$ROSETTA3/source/bin/partial_thread.macosclangrelease \
 -database ~/Rosetta/main/database/ \
```

```
 -in::file::fasta seq.fasta \
 -in::file::alignment alignments.filt \
```

Example 3C: Refine all the models.

In the final step, we refine each of the models against the density map, using the same relax script that was used in part one of the tutorial (with some modifications for symmetry).

The command line (*C_relax_density.sh*):

```
$ROSETTA3/source/bin/rosetta_scripts.macosclangrelease \
 -database ~/Rosetta/main/database/ \
 -render_density \
 -in::file::s ntermin_*.pdb ctermin_*.pdb \
 -parser::protocol C_relax_density.xml \
 -ignore_unrecognized_res \
 -edensity::mapreso 3.8 \
 -edensity::cryoem_scatterers \
 -crystal_refine \
 -beta \
 -out::suffix _relax \
 -default_max_cycles 200
```

And the XML file:

```
<ROSETTASCRIPTS>
   <SCOREFXNS>
      <ScoreFunction name="dens" weights="beta_cart">
         <Reweight scoretype="elec_dens_fast" weight="35.0"/>
         <Set
scale_sc_dens_byres="R:0.76,K:0.76,E:0.76,D:0.76,M:0.76,C:0.81,Q:0.81,H:0.81,N:0.81,T:0.81,S:0
.81,Y:0.88,W:0.88,A:0.88,F:0.88,P:0.88,I:0.88,L:0.88,V:0.88"/>
      </ScoreFunction>
   </SCOREFXNS>

   <MOVERS>
       <SetupForSymmetry name="setupdens" definition="h_edit.symm"/>
       <LoadDensityMap name="loaddens" mapfile="emd_6123.map"/>
       <FastRelax name="relaxtors" scorefxn="dens" repeats="1" cartesian="0"/>
       <FastRelax name="relaxcart" scorefxn="dens" repeats="1" cartesian="1"/>
   </MOVERS>

   <PROTOCOLS>
      <Add mover="setupdens"/>
      <Add mover="loaddens"/>
      <Add mover="relaxtors"/>
      <Add mover="relaxcart"/>
   </PROTOCOLS>
   <OUTPUT scorefxn="dens"/>
</ROSETTASCRIPTS>
```

Note the way that symmetry information is loaded, with the bolded mover above. Additionally, looking at the protocol shows that we perform one cycle of torsion refinement, followed by one cycle of cartesian refinement.

Finally, we can analyze results by looking at the output *.sc score files. For each threading, they show a score breakdown of each of the threaded models. We can evaluate these results using the command:

```
grep SCORE: *.sc | grep -v desc| sort -nk 2
```

This command sorts the outputs by total energy. What does this show? What if we sort by density energy instead?

4) Building large segments using RosettaES

RosettaCM (section 2) is a powerful tool for rebuilding small segments guided by density. However, it poorly deals with model completion of large segments of protein. These may arise in several cases:

1. Homology models (particularly distant ones) may have large insertions, or even entire domains that are lacking.
2. The models produced from *denovo_density* may be missing significant fractions of the backbone
3. It may be difficult to manually trace long stretches of low local resolution into density

To address these issues, we have developed a tool called Rosetta Enumerative Sampling, which uses a ensemble search algorithm to determine a large number of conformations that are both consistent with the density and the Rosetta energy function. This tool can be used on a partial models from the *denovo_density* application, an incomplete homology model, or any other starting structure.

RosettaES model building consists of three steps. Initially, a preparation step builds the fragments that are to be used in conformational sampling. Then a rebuilding step will identify each unassigned segment in the initial model and build an ensemble of possible solutions for each. Finally, a combination step finds all the consistent subsets of interactions, and refines all such models (if there is only one segment, the script simply refines all structures in the ensemble). In this combination step, if assembly fails to find a consistent set of solutions, an additional round of sampling will be carried out, forcing different solutions than the previous model.

Note that a full tutorial of RosettaES is given in section five main tutorial; in this "mini tutorial," we will only be using this tool to rebuild a single missing segment from a model.

Compared to the other sections, the workflow is a bit more complicated when extended to multiple compute cores. To handle job distribution we have included a python script *RunRosettaES.py* that manages this job distribution among available CPUs on a single machine. (The script is included as part of Rosetta, in /main/source/scripts/python/public/EnumerativeSampling, as well as in this tutorial). For dealing with job schedulers or clusters incompatible with this script, section 5E gives an overview of job distribution with RosettaES.

Step 4A. Fragment Picking

The first step involves selection of "fragment files," which predict backbone conformation from local sequence. We have a custom algorithm for fragment picking in RosettaES. These fragments will need to be generated before running RosettaES; the following command will generate these files (*A_PickFragments.sh*):

```
$ROSETTA3/source/bin/grower_prep.default.macosclangrelease \
     -pdb input.pdb \
     -in::file::fasta t20sA.fasta \
     -fragsizes 3 9 \
     -fragamounts 100 20
```

This will generate 100 3 residue fragments and 20 9 residue fragments, named 100.3mers and 20.9mers, that are then used in subsequent steps of the rebuilding process.

Step 4B. Generate conformations for the missing segment

The grower considers assigning positions for each unassigned segment of density (that is, each stretch of amino acids present in the fasta file but missing from the input structure). Each segment is referred to using

a segment id, in which each segment is numbered from N- to C-terminus (with multiple chains given in order in the input fasta file). The script is run in two parts: first, the script is run once for each segment to rebuild; then, the script is run in "assembly mode" given the outputs produced by rebuilding each segment individually. Thus, for rebuilding the two segments in the test case, the script is called three times: once to build each segment, and once to assemble the results.

In the first step, we perform conformational sampling for a difficult segment in aopferritin, generating an ensemble of putative solutions. This can be done calling the command (*B_SampleSegment.sh*):

```
python RunRosettaES.py \
     -rs runES.sh \
     -x RosettaES.xml \
     -f seq.fasta \
     -p difficult_loop.pdb \
     -d ../2_rosettaCM_apoferritin/emd_2788.map \
     -l 1 \
     -c 16 \
     -n loop_1
```

The arguments to this program are as follows:

- *-rs runES.sh* - the script that is launched on each core and contains Rosetta flags and inputs
- *-x RosettaES.xml* - the XML script describing parameters for conformational sampling (see below)
- *-f t20sA.fasta* - the input fasta file (with chainbreaks specified by '/')
- *-p input.pdb* - the input pdb file. This needs to match the input sequence, and all residues present in the fasta but absent in the PDB will get built.
- *-d T20S_48A_alpha_chainA.mrc* - the input density map
- *-l 1* - the segment id of the segment to rebuild. This command should be called once for each segment to rebuild, varying this argument from 1 to N
- *-c 16* - the number of compute cores to use
- *-n loop_1* - the output tag for this job (results will be placed in a folder with this name). Tags should be unique for each segment.

The input XML file exposes key parameters for conformational sampling. In the tutorial, this file, *RosettaES.xml*, contains a block:

```
...
<FragmentExtension name="ext" fasta="full.fasta" scorefxn="dens"
    censcorefxn="cendens" beamwidth="32" dumpbeam="0" samplesheets="1" read_from_file="0"
    continuous_weight="0.3" looporder="1" comparatorrounds="100" windowdensweight="30"
    readbeams="%%readbeams%%" storedbeams="%%beams%%"
    steps="%%steps%%" pcount="%%pcount%%" filterprevious="%%filterprevious%%"
    filterbeams="%%filterbeams%%">
        <Fragments fragfile="100.3mers"/>
        <Fragments fragfile="20.9mers"/>
</FragmentExtension>
...
```

The sampling behavior of RosettaES is controlled by the block above. Many of the tags in this block – *fasta*, *dumpbeam*, *read_from_file*, *storedbeams*, *steps*, *pcount*, *filterprevious*, *comparitorrounds*, and *filterbeams* – are used by the job distribution script to pass results from one step to the next, and they should be left as-is.

Others are user-specified, and can be modified based on the size of the loop and resolution of the data:

- *beamwidth*: controls the maximum number of solutions to be held at each step. Setting the value higher will increase run time but may improve accuracy.
- *windowdensweight*: the relative contribution of density in model selection

For many cases, the default parameters are sufficient. However, if the segment to grow is long (50+ residues), you may need to increase *beamwidth*; if the density is low resolution, you might need to decrease *windowdensweight* to 15 or 20.

Several options should rarely be modified, but may need to be in specific cases:

- *samplesheets*: Controls whether or not beta sheet sampling should be performed.
  It is recommended to use this *except* when working with symmetric systems.
- *continuous_weight*: Controls the penalty on discontinuous density.
  Setting this value to 1 will completely remove any penalty on discontinuous density;
  setting it closer to 0 will increase the penalty. You may wish to raise this value to 0.7 (or more) if you anticipate the segment you are trying to model does not follow a continuous path of density.

Finally, the option *comparitorrounds* is used in multi-segment assembly (see section 5C)

After running the script with this XML, there are two important intermediate output files, placed in the folder loop_1 (the argument to -n):

- *.lps* (for loop partial solution) files, which are then combined in step 5C, in cases where there are multiple segments to model
- *loop_1/beam_X.txt* files, where X corresponds to the number of residues added to the segment. These are generated as the search adds residues, and are used to pass information from one step to the next (as additional residues are added in a single segment).

Finally, while in most cases, users will want to want for a run to finish to inspect the beam, if the sampling results want to be inspected as the code is running, the final output ensemble can be saved as PDB files with the command (*B2_InspectIntermediates.sh*):

```
python RunRosettaES.py \
     -rs runES.sh \
     -x RosettaES.xml \
     -f seq.fasta \
     -p difficult_loop.pdb \
     -d ../2_rosettaCM_apoferritin/emd_2788.map \
     -l 1 \
     -db loop_1/beam_17.txt
```

Note, the number of the beam file (17) corresponds to the total number of residues built. Intermediate results (after growing *N* residues) can be inspected by changing this to a lower number (e.g., beam_14.txt shows solutions after 14 residues have been rebuilt).