Tutorial: Rosetta tools for structure determination in cryoEM density

Brandon Frenz, Ray Y.-R. Wang, Frank DiMaio
Last updated: August 2018

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 **fragment-based refinement protocol** for refinement against 3-5Å EM density
3. Our **model rebuilding protocol** (RosettaCM), where one wishes to recombine homologous structures, and rebuild *small* missing regions (<12 residues)
4. Our **de novo model-building tools**, where one wishes to rebuild missing regions of a structure (for example, from a homology model
5. 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.

Which section should one read?

- If you want a straightforward introduction to scoring and basic refinement of structures in Rosetta, *read Section 1*
- If you have a model that you want to refine into a 3 - 4.5Å density map, *read Section 2.*
- If you have a density map at 3 - 4.5Å (or worse!), one or more partial models, and you want to combine the models and rebuild short insertions and deletions, *read Section 3.*
- If you have a 3 - 4.5Å density map with no homology information available, and want to build a model, *read Section 4.*
- If you have a 3 - 4.5Å density map and a very incomplete model you want to complete, *read Section 5.*

Other notes.

For all the applications in this tutorial, it is recommended that you download the *latest weekly release* of Rosetta.

Also, for brevity, some of the command lines and XML scripts have been trimmed in this document. The tutorial files contain the full command lines and XML scripts; if something is omitted in this document, it should not be changed from the value in the tutorial file.

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>
    <OUTPUT />
</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 (described in part 4): 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.

Example 1B: 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.

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>
```

2) Model refinement via iterative local rebuilding

In this section, we introduce our **cryoEM refinement protocol**, which uses an iterative local rebuilding procedure to escape local minima during refinement. The section is divided into two parts, in the first, we introduce the method for non-symmetric systems; in the second, we describe how to use this method for symmetric systems.

As a running example, we refine models of the transmembrane region of the TRPV1 ion channel, using a 3.4 Å cryoEM single particle reconstruction (M. Liao, E. Cao, D. Julius, Y. Cheng, *Nature*, 2013), and the deposited model (id: 3j5p) as a starting model. We will first refine this asymmetrically, and then introduce symmetric refinement.

Example 2A: Asymmetric refinement into cryoEM density

A summary of the XML used for refinement (*2_cryoem_refinement/A_asymm_refine.xml*) is shown below. Following, a brief description of the movers and options available is provided.

```
<ROSETTASCRIPTS>
   <SCOREFXNS>
      <ScoreFunction name="cen" weights="score4_smooth_cart">
         <Reweight scoretype="elec_dens_fast" weight="20"/>
      </ScoreFunction>
      <ScoreFunction name="dens_soft" weights="beta_soft">
         <Reweight scoretype="cart_bonded" weight="0.5"/>
         <Reweight scoretype="pro_close" weight="0.0"/>
         <Reweight scoretype="elec_dens_fast" weight="%%denswt%%"/>
      </ScoreFunction>
      <ScoreFunction name="dens" weights="beta_cart">
         <Reweight scoretype="elec_dens_fast" weight="%%denswt%%"/>
         <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="%%map%%"/>

      <SwitchResidueTypeSetMover name="tocen" set="centroid"/>

      <MinMover name="cenmin" scorefxn="cen" type="lbfgs_armijo_nonmonotone"
         max_iter="200" tolerance="0.00001" bb="1" chi="1" jump="ALL"/>

      <CartesianSampler name="cen5_50" automode_scorecut="-0.5" scorefxn="cen"
         mcscorefxn="cen" fascorefxn="dens_soft" strategy="auto" fragbias="density"
         rms="%%rms%%" ncycles="200" fullatom="0" bbmove="1" nminsteps="25" temp="4"
         fraglens="7" nfrags="25"/>
      <CartesianSampler name="cen5_60" automode_scorecut="-0.3" scorefxn="cen"
         mcscorefxn="cen" fascorefxn="dens_soft" strategy="auto" fragbias="density"
         rms="%%rms%%" ncycles="200" fullatom="0" bbmove="1" nminsteps="25" temp="4"
         fraglens="7" nfrags="25"/>
      <CartesianSampler name="cen5_70" automode_scorecut="-0.1" scorefxn="cen"
         mcscorefxn="cen" fascorefxn="dens_soft" strategy="auto" fragbias="density"
         rms="%%rms%%" ncycles="200" fullatom="0" bbmove="1" nminsteps="25" temp="4"
         fraglens="7" nfrags="25"/>
      <CartesianSampler name="cen5_80" automode_scorecut="0.0" scorefxn="cen"
         mcscorefxn="cen" fascorefxn="dens_soft" strategy="auto" fragbias="density"
         rms="%%rms%%" ncycles="200" fullatom="0" bbmove="1" nminsteps="25" temp="4"
         fraglens="7" nfrags="25"/>

      <ReportFSC name="report" testmap="%%testmap%%" res_low="10.0" res_high="%%reso%%"/>
      <BfactorFitting name="fit_bs" max_iter="50" wt_adp="0.0005" init="1" exact="1"/>

      <FastRelax name="relaxcart" scorefxn="dens" repeats="1" cartesian="1"/>
   </MOVERS>


   <PROTOCOLS>
      <Add mover="setupdens"/>
      <Add mover="loaddens"/>
      <Add mover="tocen"/>
      <Add mover="cenmin"/>
      <Add mover="relaxcart"/>
      <Add mover="cen5_50"/>
      <Add mover="relaxcart"/>
      <Add mover="cen5_60"/>
      <Add mover="relaxcart"/>
      <Add mover="cen5_70"/>
      <Add mover="relaxcart"/>
      <Add mover="cen5_80"/>
      <Add mover="relaxcart"/>
      <Add mover="relaxcart"/>
      <Add mover="report"/>
   </PROTOCOLS>
   <OUTPUT scorefxn="dens"/>

</ROSETTASCRIPTS>
```

The protocol is a bit involved, but is described in the following. The first thing to note is the use of macros like "%%denswt%%". These are command arguments that may be specified from the command line through the flag –parser::script_vars denswt=25.0. The protocol above uses these macros in place of parameters that users would most like to change; other parameters should be left as is except for advanced users.

The main addition is the *CartesianSampler* mover. This mover iteratively locally rebuilds the structure in user-specified or automatically determined regions. A brief description of the arguments to this mover:

name="cen5_70": the name of the mover, referred to in the <PROTOCOLS> block (this can be anything)
strategy="auto": the strategy to use in selecting what to rebuild. One of:

- **auto**: select regions automatically based on density fit & local strain (using the cutoff in **automode_scorecut,** e.g., **automode_scorecut="-0.1"**)
- **user**: manually specify residues (using the flag **residues**, e.g., **residues="32A-48A"**)
- **rama**: select regions automatically based on rama score only

rms="1.5": the cutoff on similarity when locally rebuilding. Increasing this value will increase model diversity (allowing worse starting models to be refined but requiring additional sampling)
ncycles="200": the number of rebuilding cycles to consider. Increasing this will increase runtime and slightly increase model diversity.
fraglens="7": the segment length to replace. This must be an odd number from 5-13, and increasing this value will increase model diversity significantly.

The remaining options should never be changed.

Another option is passed to the density scoring via the *<Set scale_sc_dens_byres=.../>* tag. In the refinement protocol, this sets a per-amino-acid sidechain reweighing. The weights shown in this example were determined by fitting these parameters into refined structures into several 3-5Å cryoEM density maps; the end result is a slight downweighing of sidechain density, particularly for charged sidechains. This should not be changed except by advanced users.

The *MinMover* first minimizes the structure using a low-resolution energy function (*cen*). We have found this step is most useful for improving protein backbone geometry, particularly with hand-traced models. This low-resolution scorefunction uses the *centroid* representation, which is enabled by the *SwitchResidueTypeSet* mover.

The *FitBFactors* mover fits real-space atomic B factors to maximize model-map correlation. A constraint enforcing nearby atoms to take the same B factors is also employed, and the weight on this term is controlled with the *wt_adp* term (0.0005 is generally well-behaved). Finally, *init=1* means to do a quick scan of overall B factors before beginning refinement; if there is more than one call to this mover in a single trajectory, then only the first needs to have *init=1*. *Exact=1* should always be used.

Finally, the *ReportFSC* mover assesses model agreement to the map used for fitting as well as an independent map using the integrated FSC over high-resolution shells. We have found integrating from 10Å to the resolution of the data is best for model discrimination.

Finally, this command is executed using the following (*scenario2_cryoem_refinement/A_asymm_refine.sh)*:

```
$ROSETTA3/source/bin/rosetta_scripts.macosclangrelease \
  -database $ROSETTA3/database/ \
  -in::file::s 3j5p_transmem_A.pdb \
  -parser::protocol A_asymm_refine.xml \
  -parser::script_vars denswt=35 rms=1.5 reso=3.4 map=half1_34A.mrc testmap=half2_34A.mrc \
  -ignore_unrecognized_res \
  -edensity::mapreso 3.4 \
  -default_max_cycles 200 \
  -edensity::cryoem_scatterers \
  -beta \
  -out::suffix _asymm \
  -crystal_refine
```

In bold are the parameters that should be changed in adapting this run for other systems. The first is the input structure, which should be specified with the argument –in::file::s. The second are the parameters to be passed through to the script (using the macro replacement machinery of RosettaScripts). Three of these describe the input maps:

map=half1_34A.mrc – the map to refine against
testmap=half2_34A.mrc – an independent map for validation
(if not using split maps, just provide the same map as the previous argument)
reso=3.4 – the resolution of the data
(note that this needs to be provided twice in the command line, once for scoring and once for reporting)

The other two are parameters to the algorithm:

denswt=35 – the weight on the experimental density data
rms=1.5 – the amount of deviation to allow in fragment insertion moves
(larger values will lead to more model deviation)

The density weight of 25 works reasonably well as a starting point, but one might want to explore several different values using an independent reconstruction. Manual inspection of output models for molprobity score, free FSC, and (free FSC – work FSC) should provide clues as to which weight works best.

A description of much of the work in this section is described in the reference:

Wang RY, Song Y, Barad BA, Cheng Y, Fraser JS, DiMaio F. Automated structure refinement of macromolecular assemblies from cryo-EM maps using Rosetta. *Elife*. 2016 Sep 26;5. pii: e17219.

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 ./A_asymm_refine.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 . ./B1_rosettaCM_singletarget.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 free FSC values. Model geometry may be evaluated using either: a) Molprobity, or b) Rosetta energies after subtracting density energies. The latter may be done by inspecting the score*.sc files produced as output.

Using the above script, the free FSC value may be determined from the output PDB files. The header contains a line like:

```
REMARK   1 FSC[mask=4.45657](10:3) = 0.590966 / 0.591017
```

The two numbers at the end of the line indicate the "work" and "free" integrated FSC values.

Generally, we select the best 20% of models by geometry, and selecting the best overall by free FSC. The top 5 models should be inspected for model convergence as well as visually inspected for density map agreement.

Example 2B: Symmetric refinement into cryoEM density

As this is a symmetric system, to correctly evaluate the energetics of the system, we need to model with symmetry-related copies present. This may be done through a two-step process: first, we run the *make_symmdef_file.pl* script to prepare a description of the symmetry of the system in a Rosetta-readable format. Next, we enable symmetric scoring and optimization within the XML file.

The information that Rosetta needs to know about a symmetric system is encoded in the *symmetry definition file*. It tells Rosetta: (a) how to score a structure symmetrically from only asymmetric unit interactions, and (b) how the rigid-body degrees of freedom are allowed to move to maintain the symmetry of the system.

To aid in creating a symmetry definition file from a symmetric (or near-symmetric) PDB, an application, *make_symmdef_file.pl*, has been included in src/apps/public/symmetry. To generate the symmetry definition file for TRPV1, we run the command in *2_cryoem_refinement/B1_make_symmdef.sh*.

```
$ROSETTA3/source/src/apps/public/symmetry/make_symmdef_file.pl \
   -m NCS -a A -i B \
   -p 3j5p_transmem.pdb –r 1000 > TRPV1.symm
```

This script needs a few pieces of information: with –m, the type of symmetry to generate (here NCS), with –a, the primary chain (here A), and with –i, an adjacent chain in each symmetry group, separated by spaces (here just B). For C*n* symmetries, only one adjacent chain is given; for D*n*, two are given. Finally, with –r, we give the contact distance between a neighbor chain and the primary chain necessary to include that subunit explicitly (here, 1000, to ensure every symmetrically related copy is included). If the input system is asymmetric, the script will make a symmetrical version of it (sometimes significantly perturbing it in the process). There are a lot of other options, including forcing symmetrical order and helical and higher-order symmetries, see the documentation!

In addition to the definition file written to STDOUT, the script will also write a file **3j5p_transmem_symm.pdb**, containing the symmetrized version of the input file, and a file **3j5p_transmem_INPUT.pdb**, that contains only the mainchain, to be used as input (in addition to the symmetry definition file).

The symmetry definition file looks something like this:

```
symmetry_name TRPV1__4
E = 4*VRT0_base + 4*(VRT0_base:VRT3_base) + 2*(VRT0_base:VRT2_base) anchor_residue CoM
...
set_dof JUMP0_to_com x(11.7023996817515)
set_dof JUMP0_to_subunit angle_x angle_y angle_z
...
```

The omitted sections describe a system of virtual residues that maintain the symmetry of the system, and they generally should remain unedited.

The two *set_dof* lines should be edited. There are two possibilities when refining symmetric structures into density:

A) we **don't want to refine the rigid body orientation of the entire system**;
we know the symmetry axes and we don't want them to move
B) we **do want to refine the orientation of the entire system**, including symmetry axes

Generally, in cryoEM, where the maps are symmetrically averaged, and the symmetry is known, we want to use strategy A. However, in some cases (for example, if our starting model was not perfectly symmetric) we want to use strategy B. In both cases, a minor edit to the set_dof lines in the symmdef file is necessary.

For strategy **A**, because we are using density, we need to change the first set_dof line to the following:

```
set_dof JUMP0_to_com x y z
```

For strategy **B**, we leave the two lines unchanged and instead add a third line:

```
set_dof JUMP0 x y z angle_x angle_y angle_z
```

For D*n* symmetries, the changes are similar, except in **A** the jump name is JUMP0_0_to_com. The rest of this section uses strategy **A**; the edited symmetry definition file is in *scenario1_cryoem_refinement/C4_edit.symm*

Once a symmetry definition file has been generated, then refining structures in Rosetta symmetrically is straightforward. The changes to the previous XML file are indicated below (see *2_cryoem_refinement/B2_symm_refine.xml*):

```
...
<ScoreFunction name="cen" weights="score4_smooth_cart" symmetric="1">
<ScoreFunction name="dens_soft" weights="eta_soft" symmetric="1">
<ScoreFunction name="dens" weights="talaris2013_cart" symmetric="1">
...
<SetupForSymmetry name="setupsymm" definition="%%symmdef%%"/>
...
<SymMinMover name="cenmin" scorefxn="cen" type="lbfgs_armijo_nonmonotone"
         max_iter="200" tolerance="0.00001" bb="1" chi="1" jump="ALL"/>
```

In all three declared scorefunctions, symmetric=1 must be given. Additionally, the *SetupForDensityScoring* mover must be replaced with the *SetupForSymmetry* mover. Finally, the *MinMover* must be replaced with its symmetric counterpart, *SymMinMover*.

The command for running this script is largely the same as before, with a few additions:

```
$ROSETTA3/source/bin/rosetta_scripts.macosclangrelease \
  -database $ROSETTA3/database/ \
  -in::file::s 3j5p_transmem_A.pdb \
  -parser::protocol A_asymm_refine.xml \
  -parser::script_vars \
    denswt=25 rms=1.5 reso=3.4 map=half1_34A.mrc \
    testmap=half2_34A.mrc symmdef=TRPV1_edit.symm \
  -ignore_unrecognized_res \
  -score_symm_complex false \
  -edensity::mapreso 3.4 \
  -default_max_cycles 200 \
  -edensity::cryoem_scatterers \
  -beta \
  -out::suffix _symm \
  -crystal_refine
```

The command **symmdef=TRPV1_edit.symm** passes the symmetry definition file to Rosetta. The flag -score_symm_complex false depends on whether strategy (a) or (b) was employed above. If (a), then *false* should be used; if (b), then *true* should be used.

Note: The input PDB (-in::file::s) is of the monomer (that is, the asymmetric unit).

3) Model rebuilding guided by experimental density data

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 the 20S proteasome (Xueming Li *et al.*, *Nature Methods*, 2013), where only a subset of particles were used, resulting in a 4.1Å reconstruction. We are building models starting from a homologous structure (pdb id: 1iru) as the starting model (25%/32% sequence identity to chains A/B).

Example 3A: 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. In this case, we first convert our alignment to Rosetta format (*scenario2_model_rebuilding/20S_1iru.ali*):

```
## 1XXX_ 1iruAH_thread
# hhsearch
scores_from_program: 0 1.00
0 TVFSPDGRLFQVEYAREAVKK-GSTALGMKFANGVLLISDKKVRSRLIEQNSIEKIQLIDDYVAAVTSGLVADAR...
0 TIFSPEGRLYQVEYAFKAINQGGLTSVAVRGKDCAVIVTQKKVPDKLLDSSTVTHLFKITENIGCVMTGMTADSR...
--
```

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.

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 $ROSETTA3/database/ \
  -in::file::fasta t20s.fasta \
  -in::file::alignment 20S_1iru.ali \
  -in::file::template_pdb 1iruAH_aln.pdb
```

This will output a partially threaded model in *1iruA_thread.pdb* that is correctly numbered for input into RosettaCM.

Next, we need to set up symmetric modeling with RosettaCM. As in Scenario 1, we use the *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. We need a single "primary chain" and a couple of an adjacent chain in each point group; since the proteasome features D7 symmetry, that means we need an adjacent chain in the 7-fold complex, as well as a chain in the opposite ring. An example has been created in *scenario2_model_rebuilding/setup_symm.pdb* where three copies of the threaded model have been docked into density with Chimera. To generate our D7 symmetry file from this input, we then simply have to run the command (*3_model_rebuilding/B_make_symmdef.sh)*:

```
~/rosetta_source/src/apps/public/symmetry/make_symmdef_file.pl \
  -m NCS –a A -i B C \
  -p setup_symm.pdb –r 1000 > D7.symm
```

Since we have already created the input templates using the *partial_thread* application, we can ignore the *setup_symm_INPUT.pdb* file and use the output of partial thread as the input. However, we still need to align all the threaded models to this input structure. This can either be done with the program TMalign (external to Rosetta) or by using Chimera to dock the individual threaded models into density. In this case, where we have just one template, it has already been aligned to the template in *scenario2_model_rebuilding/tmpl_thread_aln.pdb.*

As in Scenario 1, we need to make a small edit to the symmetry definition file for density refinement. Change the following lines:

```
set_dof JUMP0_0_to_com x(35.3434689631743)
set_dof JUMP0_0_to_subunit angle_x angle_y angle_z
set_dof JUMP0_0 x(39.2905097135684) angle_x
```

To (*3_model_rebuilding/D7_edit.symm*):

```
set_dof JUMP0_0_to_com x y z
set_dof JUMP0_0_to_subunit angle_x angle_y angle_z
```

Note: The 20S proteasome case we are using contains two chains in the asymmetric unit. To specify this as inputs to RosettaCM, we need to list the fasta file, **separating the chains by the slash character '/'**. This is really only necessary in the fasta provided as input to RosettaCM (next step) however, there is no harm is doing this in every step.

Example 3B: Running RosettaCM using a single template model as input.

Like the methods introduced in Scenario 1, RosettaCM is controlled through an XML script using RosettaScripts. The XML is as follows (*3_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" stage1_increase_cycles="1.0"
                   stage2_increase_cycles="1.0" linmin_only="0" realign_domains="0">
            <Template pdb="1iruA_thread.pdb" weight="1.0"
                      cst_file="AUTO" symmdef="D7_edit.symm"/>
        </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); and an (optional) *symmetry definition* file.

Note: Be sure that your templates are aligned to the density!

Given this XML, RosettaCM is then run with the following command line (*3_model_rebuilding/ B1_rosettaCM_singletarget.sh)*:

```
$ROSETTA3/source/bin/rosetta_scripts.macosclangrelease \
 -database $ROSETTA3/database/ \
 -in:file:fasta t20s.fasta \
 -parser:protocol C_rosettaCM_singletarget.xml \
 -nstruct 50 \
 -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 50 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.

Running without symmetry

Running without symmetry requires only two small changes to the XML file:

- Remove the tag: symmdef="D7_edit.symm"
- Remove the three tags symmetric=1

Job distribution

As with section 2, a combination of -out::suffix and GNU parallel is useful for distributing jobs. For example, one may replace the –out::suffix line above with –out::suffix _$1, then launch jobs with:

```
parallel –j16 ./ B1_rosettaCM_singletarget.sh {} ::: {1..16}
```

The total number of structures generated is the number of structures specified with –nstruct times the number of jobs launched (in this instance, 50 structure times 16 jobs = 800 structures). Depending on the runtime per structure (variable depending on structure size) and the number of CPUs available, both of these numbers may be adjusted.

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 . ./ B1_rosettaCM_singletarget.sh {}
::: {1..48}
```

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

Example 3C: Running RosettaCM using multiple template models as input.

One of the strengths of RosettaCM is its ability to make use of multiple template structures, and to recombine portions of these models during conformational sampling. This is particularly useful when multiple homologous structures are available, some with closer sequence identity, and some with more complete coverage. The protocol allows the combination of features of both models.

To make use of this feature, we simply add additional template lines in the input XML. In this case, we add the template 1ryp (*scenario3_model_rebuilding/ C1_rosettaCM_multitarget.xml)*:

```
 ...
      <Hybridize name="hybridize" stage1_scorefxn="stage1" stage2_scorefxn="stage2"
                 fa_scorefxn="fullatom"  batch="1" stage1_increase_cycles="1.0"
                 stage2_increase_cycles="1.0" linmin_only="0" realign_domains="0">
         <Template pdb="1iruA_thread.pdb" weight="1.0"
                cst_file="auto" symmdef="D7_edit.symm"/>
         <Template pdb="1rypA_thread.pdb" weight="1.0"
                cst_file="auto" symmdef="D7_edit.symm"/>
      </Hybridize>
...
```

The rest is handled automatically by the protocol. However, there are a few caveats when using multiple input structures:

- With density, we need to ensure that all input models are aligned to the density. This can be done using either **TMalign** or **Chimera's alignment tools**.
- 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.**

Example 3D: Running RosettaCM with user specified constraints.

Another strength of RosettaCM is the ability to make use of additional experimental information that provides restraints over conformational space. While previously, we have used *cst_file=auto* to automatically generate constraints from template structures, if experiments provide distance constraints (or some other positional restraint, we may make use of them in model rebuilding as well.

The Rosetta documentation provides a good overview of the types of constraints that may be used, with a number of different constraint types and functional forms possible. For this demo, we will assume we have knowledge on the distance between residues 107 and 143 that we want to use during rebuilding.

This information can be encoded in a constraint file (*scenario3_model_rebuilding/D1_constraints.cst)*:

```
AtomPair CA 107 CA 143 HARMONIC 5.0 1.0
```

Note: The numbering of residues is based upon the order in the input fasta file (and does not reset between chains!).

We then replace the *cst_file=auto* lines in the XML with our own constraint file (*scenario3_model_rebuilding/D1_constraints.xml)*:

```
...
   <Template pdb="tmpl_thread_aln.pdb" weight="1.0"
         cst_file="D1_constraints.cst" symmdef="D7_edit.symm"/>
...
```

We can then rebuild and refine as before.

Example 3E: Model selection and running RosettaCM iteratively

With possibly hundreds of generated models, there are a few strategies to identify the best-sampled models. Generally, models should be filtered on two different criteria – the *total score* and the *density score* – in some way. We often select the best 10-20% of models based on total score, and the sort these models by density score, but visual inspection of the best by both criteria can often be beneficial in difficult cases.

Finally, one strategy for solving difficult structures is to apply RosettaCM iteratively. Using the above criteria, we can select the best 5-10 models from the first round of refinement, and feed them as inputs into the next round. Models can be selected by energy by looking at the score column in the output .sc files.

This is very briefly illustrated in the following XML (*scenario3_model_rebuilding/E1_rosettaCM_iter.xml)*:

```
 ...
      <Hybridize name="hybridize" stage1_scorefxn="stage1" stage2_scorefxn="stage2"
             fa_scorefxn="fullatom" batch="1">
         <Template pdb="expected_outputs/S_multitgt_0001_A.pdb" weight="1.0"
                cst_file="NONE" symmdef="D7_edit.symm"/>
         <Template pdb="expected_outputs/S_multitgt_0002_A.pdb" weight="1.0"
                cst_file="NONE" symmdef="D7_edit.symm"/>
         <Template pdb="expected_outputs/S_multitgt_0003_A.pdb" weight="1.0"
                cst_file="NONE" symmdef="D7_edit.symm"/>
      </Hybridize>
...
```

There is also some manipulation of input models that can prove beneficial. If one wants to force rebuilding some segment of backbone, they can simply delete those residues in all input models. Similarly if one wants to force some region to adopt a conformation taking in one input model, they can delete all other conformations from all models.

Example 3F: Running with Ligands and/or Nucleic Acids

RosettaCM can be run with ligands as well as nucleic acids. While nucleic acids are read by Rosetta natively, ligands require an additional input to Rosetta, a *params* file.

For ligands, a mol2 file of the ligand which contains hydrogen atoms is required. The program OpenBabel (http://openbabel.org/wiki/Main_Page) is capable of both converting from PDB to mol2 as well as adding hydrogens to a molecule given a PDB file of the ligand.

Given a mol2 file, *molfile_to_params.py*, included in Rosetta, at $ROSETTA3/source/scripts/python/public/ will generate a params file. To use:

```
python $ROSETTA3/source/scripts/python/apps/public/molfile_to_params.py \
      --keep-names --clobber -centroid XXX.mol2 -p XXX -n XXX
```

Replace "XXX" with the ligands three letter code. This script will create an XXX.params file as well as several other files. This file can be passed into Rosetta using the flag:

```
-extra_res_fa XXX.fa.params
-extra_res_cen XXX.cen.params
```

If there is more than one unique ligand, run the script on each unique ligand, and pass a list of params files using each of the flags.

When modelling with ligands or nucleic acids in RosettaCM, two additional things are needed:
1. The ligand or nucleic acids must be added to the END of each template file with a weight > 0
2. An additional flag needs to get added to Hybridize:

```
<Hybridize name="hybridize" stage1_scorefxn="stage1" stage2_scorefxn="stage2"
             fa_scorefxn="fullatom" batch="1" add_hetatm="1">
```

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).

5) Completing partial models guided by experimental density data

While the previous workflows have address model building and model refinement, none of the aforementioned tools deal with 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_denisty* application, an incomplete homology model, or any other starting structure.

RosettaES runs best when working with data at resolutions 5 Å or better with segments to rebuild shorter than 50 residues. However, with very large amounts of sampling (e.g., ensemble sizes > 250), reliable models may be produced with segments longer than 100 residues. At resolutions worse than 5 Å, this tool may be unreliable. The method can be used on both segmented and unsegmented density maps, however, *removal of density belonging to parts of the structure not being modeled may improve results*.

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.

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 5A. Fragment Picking

The first step – much like Scenario 4 – involves selection of "fragment files," which predict backbone conformation from local sequence. Unlike Scenario 4, we have a custom algorithm for fragment picking. These fragments will need to be generated before running RosettaES; the following command will generate these files (*5_rosettaES/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 5B. Generate Possible Conformations For Each 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 of each of the two segments, generating an ensemble of putative solutions for each. This can be done calling the command (*5_rosettaES/B1_SampleSegment1.sh*):

```
python RunRosettaES.py \
     -rs runES.sh \
     -x RosettaES.xml \
     -f t20sA.fasta \
     -p input.pdb \
     -d T20S_48A_alpha_chainA.mrc \
     -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, *5_rosettaES/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
- *taboo/beamX.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).

Note: This process should then be repeated for all remaining segments to rebuild In the tutorial, the command *5_rosettaES/B2_SampleSegment2.sh* builds conformations for the second segment in this file. All segments can be sampled independently of one another, so if many compute nodes are available, each segment can be sampled simultaneously on separate nodes.

Finally, while in most cases, users will want to take these models into the assembly step (part 5C), if there is only one segment to rebuild, *or* if the sampling results want to be inspected, the final output ensemble can be saved as PDB files with the command (*5_rosettaES/B1.2_InspectIntermediates.sh*):

```
python RunRosettaES.py \
     -rs runES.sh \
     -x RosettaES.xml \
     -f t20sA.fasta \
     -p input.pdb \
     -d T20S_48A_alpha_chainA.mrc \
     -l 1 \
     -db loop_1/taboo/beam17.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., beam14.txt shows solutions after 14 of the 17 residues have been rebuilt).

Step 5C. Find a set of consistent conformations.

In cases where there are multiple interacting segments, we want to find all nonclashing combinations. This step will take the loop partial solution (lps) files generated in step B and use a Monte Carlo Assembly (MCA) algorithm in order to identify sets of solutions that are self-consistent. **This section assumes that all missing segments have been built in step B**. To run this assembly, we perform conformational sampling using the script, passing the .lps files generated in step B (*5_rosettaES/C_AssembleResults.sh*):

```
python RunRosettaES.py \
     -rs runES.sh \
     -x RosettaES.xml \
     -f t20sA.fasta \
     -p input.pdb \
     -d T20S_48A_alpha_chainA.mrc \
     -lps loop_*/lps*.txt
```

The flag -lps points to the outputs of the individual segment jobs' output. As before, the script will use (and modify) an input XML file. For assembly, only a single parameter is important to modify: *comparatorrounds*="100". This parameter controls the number of Monte Carlo trajectories (it is unlikely you will need to change this parameter).

The output of this step is PDB files, that will be placed in the working directory with the prefix *aftercomparator_RRR_XXX.pdb*, where *RRR* is the trajectory id, and *XXX* is the energy. A text file *recommendation.txt* will be written that reports the clash score of the best model.

If the number in *recommendation.txt* is-above +100 it is recommend you perform additional rounds of sampling. To do so, additional potential solutions should be sampled by repeating step B and providing the same directory name as input, without deleting the intermediate files. The script will then enter that directory and use the already-computed solutions (stored in the folder "*taboo*") to guide sampling toward previously unexplored regions. See section D for more details.

If the number in *recommendation.txt* is below +100, then models should be run through Rosetta refinement to accurately rank them. That can be done using this command (*5_rosettaES/D_RefineOutput*):

```
python RunRosettaES.py \
     -rlxs runrelax.sh \
     -x RosettaES.xml \
     -p input.pdb \
     -d T20S_48A_alpha_chainA.mrc \
     -c 16 \
     -rp aftercomparator_*.pdb
```

Where runrelax.sh contains a command for relaxing structures in Rosetta.

Step 5D. Interpreting results

RosettaES will produce a 100 models as output (or whatever is specified in *comparatorrounds*), with scores in the file *score.sc*. When first looking at the output from a run, it is good to visually inspect the lowest-energy 5-10 structures. Ideally, these lowest-energy models will be very tightly converged, but the lack of convergence does not indicate failure in sampling, and the presence of convergence does not necessary indicate the solution is correct. Instead, the lowest-scoring models should be examined with attention to the following:

Insufficient sampling. Models should initially be inspected for clearly incorrect features that might not have been properly penalized by Rosetta. These include:

- unexplained density, particularly small sidechains placed into a large density protrusions
- regions with poor fit to density
- unresolved clashes

If these features are present in the lowest-energy models, it suggests that the conformational sampling performed by RosettaES may not have been sufficient. There are several ways to address this issue. The first is to increase conformational sampling. This can be done by either: a) increased the beamwidth parameter and rerunning anew, b) and/or performing additional rounds of taboo search (taboo sampling occurs automatically if the "taboo/" folder in the running directory is populated with beam files"), or c) reducing the search space by eliminating regions of density.

The latter can also be performed by manually removing parts of the map you do not wish to sample or by including additional portions of the model, for example if you are building a model that has 4 missing regions and you believe you have accurately sampled 3 of them (they do not posses any of the pathologies listed above and fit the density well), you can combine them by treating them as templates for RosettaCM (use a fasta file with the missing segment removed) and use the result as input for RosettaES to build the last region.

Unresolved residues. RosettaES will always attempt to build all residues present in the fasta file, however, in many cases not all residues will be resolved in the map (particularly at termini). Because RosettaES will heavily penalize models that do not fit the density, you will often find models that are "overly compacted" at termini or internal loops, to try to squeeze these residues into density.

If this happens *at termini* it is suggested that you examine intermediate structures that have yet to attempt to assign these unresolved residues in order to find a good model. If this happens internally (or if you have a good idea *a priori* what residues should be modeled), these regions can be removed from the input fasta. If internal segments are deleted, **be sure to treat the deletion correctly, by putting a '/' in the fasta file**.

Step 5E. Customizing job distribution.

Job distribution in RosettaES is complicated, since each "growing step" can be parallelized, but subsequent steps need all the information from the previous step. Consequently, the script provided (RosettaES.py) manages jobs, by calling Rosetta jobs at each round, collecting and combining the results of the previous round, then splitting input files for the next round. On systems where it is not possible to run this script, this section describes in some detail what the script is doing.

To manage this the python script uses the provided XML file as input, rewriting several parameters depending on the protocol step. These varying parameters include:

- *readbeams*, a boolean option that tells Rosetta whether it should load intermediate solutions
- *beamfile,* the name of the beamfile that stores the intermediate solutions
- *steps,* how many residues to add (0 means only do filtering, otherwise set to 1, setting to "-1" will build all missing residues)
- *pcount,* used to uniquely tag output files from each core
- *filterprevious,* a boolean option that controls whether intermediate solutions should be read for taboo search
- *filterbeams,* the filename of the intermediate solutions for use in taboo.

In order to build a missing segment the script will perform the following steps:

1. Launch a job with
   readbeams="0", *beamfile*="na", *steps*="1", *pcount*="1", *filterprevious*="0", *filterbeams*="na".
   This will produce a file named **beam1.1.txt** from Rosetta.
2. Parse the **beam1.1.txt** file and split into *N* files (where *N* is the number of parallel jobs to run).
   New files are labeled **beam_$r.$i.txt** where **$r** is the number of residues added so far,
   and **$i** ranges from 1 to *N*.
3. Submit *N* rosetta jobs with *pcount* set as the job number, *readbeams* set to "true", and *beamfile* set to the corresponding output of step 2.
4. Output files are parsed by the script and compiled into a single file with the name **beam$r.txt**, with $r the number of residues grown.
5. Rosetta is run on a single core to filter the aggregate solution set. Here,
   readbeams="1", *beamfile*="beam$r.txt", and *steps*="0".
   A file named **beam_0.txt** will have the filtered results.
6. Repeat steps 2-6 using the **beam_0.txt** as the input for step 2. This is done until the segment is complete and Rosetta produces an empty file called **finished.txt**. The presence of this file triggers the program to perform one final round of filtering and exit.

The last step of RosettaES will additionally produce a file with the name "lpsfile_$s.0.txt," used for assembly. To run assembly with these files first combine them into a single file, described below, and run with readfromfile="*filename*". This new file should start with the a number corresponding to the total number of missing segments, then for each missing segment provide a number for the total solutions in that segmented followed by the solution information contained in the lpsfile_$s.0.txt described above. Segments should be arranged to match the order in which they occur in the fasta.