Engenharia Computacional de Proteínas

Instrutores: Roberto Lins, Danilo Coêlho, Elton Chaves

Tutorial elaborado por Danilo Coêlho, Roberto Lins e Elton Chaves, inspirado no tutorial ***Creating protocols with RosettaScripts*** por Rocco Moretti e Vikram K. Mulligan

(https://docs.rosettacommons.org/demos/latest/tutorials/scripting_with_rosettascripts/scripting_with_rosettascripts)

Getting Familiar with RosettaScripts

What is RosettaScripts?

Originally, the interface for Rosetta3 functionality was individual applications, each made specifically for a particular use. One drawback of this approach was that customization of protocols was difficult. If the protocol author properly anticipated users' needs, then they may have put in options which allowed users to change the protocol.

However, these applications are typically limited in the extent to which they allow users to modify the protocol. To allow for greater flexibility, RosettaScripts was created. RosettaScripts allows users to create and modify protocols using an XML based syntax. Broadly, RosettaScripts is based around the paradigm of having a single structure (the pose) that enters the protocol, a series of steps performed, each modifying the pose in some way (movers) or evaluating some property of the pose (filters), and a single structure written out. The protocol can then be run repeatedly to generate large ensembles of output structures, or to process large ensembles of input structures. Even more broadly, RosettaScripts lets a user link individual Rosetta modules together in a linear sequence.

This tutorial is intended to take you through the process of creating a new protocol with RosettaScripts. It should also give you a good grounding in how you can modify existing RosettaScripts protocols. Note that you can certainly run RosettaScripts without modifying the XML - ***the most common use case of RosettaScripts is probably re-using an XML produced by someone else***.

Your first RosettaScript

The simplest RosettaScript XML is one which does nothing. You can obtain a skeleton XML file, which does nothing, in one of two ways. You go to the

RosettaScripts documentation page and find the skeleton XML file there, then copy and paste it into a new file (*nothing.xml*). You can also generate a skeleton XML file by running the rosetta_scripts application without any parameters. Run the following command **(do not copy it, type it yourself)**:

```
$ROSETTA3/bin/rosetta_scripts.linuxgccrelease -database path/to/rosetta/database
```

In the above, the ".default.linuxgccrelease" may need to be changed for your build operating system and compiler (*e.g.* ".macosgccrelease" for build using the gcc compiler on the MacOS operating system).

Also, you should properly replace *path/to/rosetta/database* to indicate where is located the Rosetta Database in your PC. If you run the above, it will produce output similar to the following:

```
core.init: Checking for fconfig files in pwd and ./rosetta/flags

core.init:     Rosetta     version:     rosetta.source.release-334     r334     2022.45+release.20a5bfe
20a5bfe160d0310bb7266ea8124fa2b8c5d99c1c https://www.rosettacommons.org 2022-11-07T19:45:32.206293

core.init: command: rosetta_scripts.macosclangrelease

basic.random.init_random_generator:  'RNG  device'  seed  mode,  using  '/dev/urandom',  seed=-1323852309
seed_offset=0 real_seed=-1323852309

basic.random.init_random_generator: RandomGenerator:init: Normal mode, seed=-1323852309 RG_type=mt19937

core.init:          found          database          environment          variable          ROSETTA3_DB:
/opt/programs/rosetta/3.13/rosetta.source.release-334/main/database

core.init:

core.init: USEFUL TIP: Type -help to get the options for this Rosetta executable.

core.init:

protocols.RosettaScripts.util:  No  XML  file  was  specified  with  the  "-parser:protocol  <filename>"
commandline option.  In order for RosettaScripts to do something, it must be provided with a script.

protocols.RosettaScripts.util: The following is an empty (template) RosettaScripts XML file:


<ROSETTASCRIPTS>
        <SCOREFXNS>
        </SCOREFXNS>
        <RESIDUE_SELECTORS>
        </RESIDUE_SELECTORS>
        <PACKER_PALETTES>
        </PACKER_PALETTES>
        <TASKOPERATIONS>
        </TASKOPERATIONS>
        <MOVE_MAP_FACTORIES>
        </MOVE_MAP_FACTORIES>
        <SIMPLE_METRICS>
        </SIMPLE_METRICS>
        <FILTERS>
        </FILTERS>
        <MOVERS>
        </MOVERS>
        <PROTOCOLS>
        </PROTOCOLS>
        <OUTPUT />
</ROSETTASCRIPTS>


At any point in a script, you can include text from another file using <xi:include href="filename.xml"
/>.

protocols.RosettaScripts.util:  Variable  substituion  is  possible  from  the  commandline  using  the  -
"parser:script_vars varname=value" flag.  Any string of the pattern "%%varname%%" will be replaced with
"value" in the script.

protocols.RosettaScripts.util:

protocols.RosettaScripts.util: The rosetta_scripts application will now exit.
```

Copy the lines from `</ROSETTASCRIPTS>` to `<ROSETTASCRIPTS>`, use a text editor to create a new file called *nothing.xml*, and paste the copied lines.

An important feature of XML language is that anything not in angle brackets (*i.e.* < >) is a comment. This makes it easy to temporarily disable things by deleting just the first angle bracket.

The *nothing.xml* file is also provided in the inputs directory. You can copy it to your current working directory if needed:

```
cp ECP_TUTORIAL_RSCRIPTS/inputs/nothing.xml .
```

As you haven't further defined any protocol, this XML does nothing to the structure. As a test, let's just run a structure through RosettaScripts with this XML. RosettaScripts takes the standard input and output flags. In addition, the *-parser:protocol* option specifies which XML file to use.

But first copy the test structure *1ubq.pdb* to your current working directory:

```
cp ECP_TUTORIAL_RSCRIPTS/inputs/1ubq.pdb .
```

Now, run the following command **(do not copy it, type it yourself)**:

```
$ROSETTA3_BIN/bin/rosetta_scripts.default.linuxgccrelease       -database
path/to/rosetta/database -s 1ubq.pdb -parser:protocol nothing.xml
```

In the tracer output, Rosetta should print its interpretation of the XML input:

```
<ROSETTASCRIPTS>
	<SCOREFXNS/>
	<RESIDUE_SELECTORS/>
	<PACKER_PALETTES/>
	<TASKOPERATIONS/>
	<MOVE_MAP_FACTORIES/>
	<SIMPLE_METRICS/>
	<FILTERS/>
	<MOVERS/>
	<PROTOCOLS/>
	<OUTPUT/>
</ROSETTASCRIPTS>
```

The first thing to notice is that the comments added to the XML (everything outside the angle brackets) is ignored.

Secondly, this demonstrates different ways of writing XML tags. XML tags are surrounded by angle brackets (greater/less than signs). A tag must be closed by a slash. Tags can be nested in other tags (like SCOREFXNS is nested within ROSETTASCRIPTS), in which case the outer tag must be closed by something like `</ROSETTASCRIPTS>` . If the tags are not nested, they can be closed by putting the slash at the end of the tag, like `<SCOREFXNS/>` . The following two statements are perfectly equivalent:

```
	<SCOREFXNS>
	</SCOREFXNS>
```

or

```
	<SCOREFXNS/>
```

Additionally, whitespace is largely ignored in RosettaScripts.

Conventionally, tags are indented in proportion to their level of nesting, but this is for human readability, not for machine parsing; the rosetta_scripts application disregards tabs entirely. The one case in which whitespace matters is when setting options within a tag. When a tag contains an option that accepts a comma-separated list, these must not have whitespace within them:

```
<PackRotamers name="pack1" task_operations="task1,task2,task3" /> #This is allowed
<PackRotamers name="pack2" task_operations="task2, task2, task3" /> #This will be misinterpreted
```

This brings up another RosettaScripts syntax convention: generally, we have blocks that define types of objects, and within these blocks, we define individual instances of objects of the type, giving each one a unique name. For example, the `<MOVERS> ... </MOVERS>` block is the place to define movers. Within this, we define specific instances of specific types of movers, and we set options for these movers, including a unique name by which each mover will be addressed at later points in the script. For example:

```
<MOVERS>
In this section, movers are defined. The following is a particular mover of the "PackRotamers" type, which we give the unique name "pack1".  It takes, as an option, a list of previously - defined TaskOperation objects (a type of object that will be introduced later in this tutorial).  We assume that task1, task2, and task3 were defined and given these unique names prior to this point in the script.

      <PackRotamers name="pack1" task_operations="task1,task2,task3" />

From now on, we can refer to the mover defined above using the unique name "pack1".
</MOVERS>
```

Looking at the output PDB, the output structure (**1ubq_0001.pdb**) should be nearly identical to the input structure. The major difference should be the presence of hydrogens which were not in the input structure. This is not something that is specific to RosettaScripts - in general Rosetta will add missing hydrogens and repack sidechain atoms missing in the input PDB.

Controlling RosettaScripts File Output

Before we explore the full power of RosettaScripts, let's make sure that we understand how to control the rosetta_scripts application's output. There are two ways to do this. The first is modifying the <OUTPUT/> tag typically found at the end of a script, and the second is by setting flags.

Let's look at a typical usage case for the <OUTPUT/> tag, first. Sometimes you may want to use different energy functions during different scoring. For

example, you may want to change constraint weights, or to use a lower resolution energy function. In order to do this, we:

1. Add a named custom scoring function in the SCOREFXNS section of the XML.

2. Add this to the <OUTPUT/> tag.

Each custom scorefunction is defined by different sub-tags in the SCOREFXNS section.

```
<ROSETTASCRIPTS>
      <SCOREFXNS>
          <ScoreFunction name="molmech" weights="mm_std_fa_elec_dslf_fa13" />
          <ScoreFunction name="r15_cart" weights="ref2015" >
              <Reweight scoretype="pro_close" weight="0.0" />
              <Reweight scoretype="cart_bonded" weight="0.625" />
          </ScoreFunction>
      </SCOREFXNS>
      <RESIDUE_SELECTORS>
      </RESIDUE_SELECTORS>
      <TASKOPERATIONS>
      </TASKOPERATIONS>
      <FILTERS>
      </FILTERS>
      <MOVERS>
      </MOVERS>
      <APPLY_TO_POSE>
      </APPLY_TO_POSE>
      <PROTOCOLS>
      </PROTOCOLS>
      <OUTPUT scorefxn="r15_cart" />
  </ROSETTASCRIPTS>
```

The script ***scoring.xml*** gives an example of defining different scorefunctions. It defines two scorefunctions. The first one (molmech) is a molecular mechanics scorefunction that is included in the Rosetta database, used as-is, and the second (r15_cart) is the ref2015 scorefunction modified by changing the weights (coefficients) for certain score terms. (One can also use patch files, or locally-specified weights files; additionally, other scorefunction options can be set, such as soft Lennard-Jones potentials or whatnot. See the documentation on the Set tag in the RosettaScripts documentation for more on this.)

The molmech scorefunction is never used in this script, which is not a problem -- RosettaScripts does not object to objects that are defined but never used (though the unnecessary allocation of these objects in memory is probably best avoided if one can help it). The r15_cart score function is used, however, in the OUTPUT tag. This tells RosettaScripts to rescore the output structures with the custom r15_cart scorefunction, rather than with the default (command line) scorefunction.

First, copy the scoring.xml file to your current working directory:

```
cp ECP_TUTORIAL_RSCRIPTS/inputs/scoring.xml .
```

Now, run 1ubq.pdb through the script **(do not copy the command line, type it yourself)**:

```
$ROSETTA3/bin/rosetta_scripts.linuxgccrelease -database path/to/rosetta/database -s
1ubq.pdb -parser:protocol scoring.xml -out:prefix scoring_
```

If you open the scoring_1ubq_0001.pdb output file, you should see that the score table includes columns for the cart_bonded term, and no pro_close term.

Altering the Pose: Movers

Minimization

The core of a RosettaScript XML is the movers. Movers are what will change the structure. Technically, movers are anything that changes the pose. While this includes changes to the atomic coordinates, it also includes changes to other features of the pose, such as constraints, sequence, or covalent connectivity. There are certain movers which will change just this auxiliary information, without altering atomic coordinates at all.

As an initial demonstration, we're going to start by writing a script that uses a mover to do gradient-descent energy minimization of the pose, the ***MinMover***, minimizes sidechains and/or backbone.

As mentioned previously, one option that should be in the tag for each mover is the "name" option, with which the user creates a unique handle for referring to that particular instance of the mover (in the PROTOCOLS section). The

value given should be unique to each instance of a mover. You can have multiple MinMovers as long as their names are different.

The other options in the tag control the mover's behavior. Most of the options in a tag will have default values associated with them. These are the values which will be used if the option is not provided with the tag. The default values are frequently (though not always) the recommended values for the option, so if you are unsure as to what the option value should be, omitting the option and having it revert to the default value is a good choice. This is what we'll do with the type, tolerance, and max_iter options in this case.

For our example script, we'll make two MinMovers. One we'll call "min_torsion", which will have the cartesian option set to false (so it will use the default torsional minimization) and will use the molmech scorefunction. The other we'll call "min_cart", and it will have the cartesian option set to true and use the r15_cart scorefunction. Both will have bb and chi set to true.

Make a copy of ***scoring.xml*** called ***minimization.xml***, and then copy the following tags to a place between the <MOVERS> and </MOVERS> tags.

```
<MinMover name="min_torsion" scorefxn="molmech" chi="true" bb="1" cartesian="F" />
<MinMover name="min_cart" scorefxn="r15_cart" chi="true" bb="1" cartesian="T" />
```

Declaring the movers in the MOVERS section only tells Rosetta that the movers exist and configures their options; however, it doesn't tell Rosetta that they should be applied to the pose (or in what order, or the number of times). The PROTOCOLS section is used to define the sequence of steps that the rosetta_scripts application will carry out. When RosettaScripts runs on a structure, it will run sequentially through all the entries in the PROTOCOLS section, executing each in order, the output of the previous mover becoming the input to the next.

In our protocols section we'll add the "min_cart" mover. Since this is the only mover in the PROTOCOLS section, this is the only mover which will be run. The min_torsions mover will be defined, but will not be applied to the pose.

Add the following tags to a place between the <PROTOCOLS> and </PROTOCOLS> tags.

```
<Add mover="min_cart" />
```

A copy of the minimization.xml file is provided in ***ECP_TUTORIAL_RSCRIPTS/inputs.***

Run 1ubq.pdb through the minimization.xml script **(do not copy the command line, type it yourself)**:

```
$ROSETTA3/bin/rosetta_scripts.linuxgccrelease -database
path/to/rosetta/database -s 1ubq.pdb -parser:protocol
minimization.xml -out:prefix minimize_
```

Within the tracer output you should see indications that your movers are being used (e.g. "BEGIN MOVER MinMover - min_cart"). Also, if you look at the total scores from the output PDB, you should get much better scores for the minimized 1ubq than the one just rescored with r15_cart. (about -155 versus +460).

Now let's add the other minimization mover, to demonstrate how movers can be placed in series. Add the following tags to a place between the <PROTOCOLS> and </PROTOCOLS> tags, using any text editor:

```
<Add mover="min_torsion" />
```

Save the new script as minimization2.xml.

A copy of the minimization2.xml file is also provided in ***ECP_TUTORIAL_RSCRIPTS/inputs.***

Run 1ubq.pdb through the minimization2.xml script **(do not copy the command line, type it yourself)**:

```
$ROSETTA3/bin/rosetta_scripts.linuxgccrelease -database
path/to/rosetta/database -s 1ubq.pdb -parser:protocol minimization2.xml
-out:prefix minimize2_
```

This time, when you run the application, you'll find that the torsion-space minimization is carried out first (using the molecular mechanics scorefunction), and the Cartesian-space minimization is carried out on the output structure from the torsion-space minimization (using the ref2015 scorefunction, modified with the cart_bonded term turned on and the pro_close term turned off). Note that Rosetta does not write out any structures until the end of the protocol.

Packing

The ***PackRotamersMover*** is another commonly-used Rosetta mover. Because it calls the packer, another core Rosetta algorithm, the *PackRotamersMover* is a good mover to use to demonstrate the *RosettaScripts* interface for controlling the packer. We do this by defining ***TaskOperations***.

Let's create a new XML, defining the ref2015 scorefunction in it. Additionally, in the movers section, let's create a *PackRotamersMover*. Don't forget to add it to the protocols section, as well.

The standard behavior of *PackRotamersMover* would call the packer, and the packer would use all rotamers for all 20 canonical amino acids at every position -- that is, it would try to design the entire protein, which is not what we want.

TaskOperations are the means by which the user controls the packer. They specify which residue to repack and/or design, and how to do it. *TaskOperations* are defined in the TASKOPERATIONS section of the XML.

In addition to controlling which positions are designed or repacked, *TaskOperations* also control details about how sidechains are sampled. The default is strictly for on-rotamer sampling, but it's frequently useful to add additional sub-rotameric samples. The ***ExtraRotamersGeneric*** *TaskOperation* allows you to control the rotamer sampling levels. Generally, adding some additional rotamers to chi1 and chi2 is useful, though the cost is a more complex packing problem and longer convergence time.

Let's create two *TaskOperations*. The first, ***RestrictToRepacking***, will tell the packer to use only the current amino acid type at each position, and consider only alternative rotamers for that type (*i.e.* disabling design). The second, ***ExtraRotamersGeneric***, will enable some extra rotamers.

Your MOVERS and PROTOCOLS sections should look something like this:

```
<ROSETTASCRIPTS>
	<SCOREFXNS>
		<ScoreFunction name="r15" weights="ref2015" />
	</SCOREFXNS>
	<RESIDUE_SELECTORS>
	</RESIDUE_SELECTORS>
	<TASKOPERATIONS>
		<RestrictToRepacking name="no_design" />
		<ExtraRotamersGeneric name="extrachi" ex1="1" ex2="1" />
	</TASKOPERATIONS>
	<FILTERS>
	</FILTERS>
	<MOVERS>
		<PackRotamersMover name="pack1" scorefxn="r15" task_operations="no_design,extrachi" />
	</MOVERS>
	<APPLY_TO_POSE>
	</APPLY_TO_POSE>
	<PROTOCOLS>
		<Add mover="pack1" />
	</PROTOCOLS>
	<OUTPUT scorefxn="r15" />
</ROSETTASCRIPTS>
```

Now let's run this script. (If you need, copy the ***ECP_TUTORIAL_RSCRIPTS/inputs/repack_only.xml*** file)

```
$ROSETTA3/bin/rosetta_scripts.linuxgccrelease -database
path/to/rosetta/database -s 1ubq.pdb -parser:protocol
repack_only.xml -out:prefix repack_only_
```

This should generate approximately 2305 rotamers, and take on the order of a second or two to run.

Filters

Because Rosetta runs are typically stochastic, early stages will often sample conformations which will not be productive. That is, the randomness introduced by initial movers will result in conformations which will never lead to useful final models. To speed up the protocol, it is sometimes helpful to

abandon some samples before the final stages of sampling when early stages result in conformations which are known to be unproductive. To facilitate this, RosettaScripts provides ***Filters***, which can stop a job based on measured properties of the protein structure, allowing the rosetta_scripts application to continue to the next job (i.e. the next replicate of the protocol with the current input or the next input structure).

Let's consider the case, now, of repacking just the surface (i.e. solvent-exposed) residues of ubiqutin, followed by full minimization. In this case, we're passing *TaskOperations* for preventing design and for enabling extra rotamers to a *PackRotamersMover*. We also define a *MinMover* to do minimization. In the protocols section, we call the *PackRotamersMover* first, then the *MinMover*.

In the original structure, there is a salt bridge between K11 and E34, and we probably want to preserve that. The packer may or may not keep that, though -- sometimes in a packer run, we may not get that. It's the case that if we start with sidechain configurations which are too far apart, minimizing will never pull K11 and E34 back together to re-form the salt bridge. So if we definitely want the salt bridge in our output structures, the time spent on minimizing the non-salt bridged packing output is effectively wasted. While this may be seconds in a single run, if we're doing large-scale sampling (say, tens of thousands of trajectories), this could add up to quite a lot of wasted CPU-time. In many cases, later steps might take minutes or hours, so avoiding unnecessary computation is definitely worthwhile. Additionally, given that one often manually looks at output structures as a final step, it is good to have a way to reduce the amount of output to a managable number of structures. In this case, we will use a filter to abandon those jobs that fail to form the salt bridge before we minimize.

To enforce the salt bridge in this case, we will filter based on the distance between the two atoms: if they're close enough, we can continue. If they're too far apart, we'll throw out the structure. The *AtomicDistance* ("Filter based on the distance between two atoms.") looks to be what we want.

For the *AtomicDistance* filter, you can specify either the specific atom name, or you can specify a Rosetta atom type. If an atom type is specified, then the closest distance for any atom of the relevant type is used. This latter behavior is what we want; we don't care which of the carboxylate oxygens are paired with the lysine side-chain nitrogen. Therefore we can specify the atom types: the "OOC" oxygens from E34 pairing with the "Nlys" nitrogen from K11.

Most filters work by computing some structural metric, and then comparing it to a threshold value to determine if the filter passes or fails. The *AtomicDistance* filter uses the "distance" options to set the threshold: distances below this pass, distances above fail.

We want to set the distance threshold large enough such that it will pass all the structures which have the salt bridge, but also narrow enough that it will fail the structures which don't have it. For this tutorial will use a possibly too narrow distance of 3.0 Å.

```
...
    <FILTERS>
                <AtomicDistance  name="salt_bridge"  residue1="11A"  atomtype1="Nlys"
residue2="34A" atomtype2="OOC" distance="3.0" />
    </FILTERS>
...
```

Again, this only defines the filter. To actually apply it, we have to add it to the protocols section.

```
...
    <PROTOCOLS>
        <Add mover="pack" />
        <Add filter="salt_bridge" />
        <Add mover="min" />
    </PROTOCOLS>
...
```

Within the PROTOCOLS section, movers and filters are listed in the order in which they are to be evaluated. That is, the structure will first be packed, then the filter will be applied, and then, if and only if the filter passes, it will be minimized. If the filter fails, a message is printed to the output log, and the rosetta_scripts application will continue to the next job.

Let's try this out. This time, we'll tell the rosetta_scripts application to repeat the job 10 times with the ***-nstruct 100*** option at the commandline. We expect that some fraction of the jobs will succeed and some will fail to form the salt bridge and will be abandoned.

A script to do this might look something like the following (a copy of the filter.xml file is also provided in ***ECP_TUTORIAL_RSCRIPTS/inputs***).

```
<ROSETTASCRIPTS>
	<SCOREFXNS>
		<ScoreFunction name="r15" weights="ref2015" />
	</SCOREFXNS>
	<RESIDUE_SELECTORS>
	</RESIDUE_SELECTORS>
	<TASKOPERATIONS>
		<RestrictToRepacking name="repackonly" />
		<ExtraRotamersGeneric	name="extrachi"	ex1="1"	ex2="1"
ex1_sample_level="1" ex2_sample_level="1" />
	</TASKOPERATIONS>
	<FILTERS>
		<AtomicDistance  name="salt_bridge"  residue1="11A"  atomtype1="Nlys"
residue2="34A" atomtype2="OOC" distance="3.0" />
	</FILTERS>
	<MOVERS>
		<MinMover	name="min"	scorefxn="r15"	chi="true"	bb="true"
cartesian="false" />
		<PackRotamersMover		name="pack"		scorefxn="r15"
task_operations="repackonly,extrachi"/>
	</MOVERS>
	<APPLY_TO_POSE>
	</APPLY_TO_POSE>
	<PROTOCOLS>
		<Add mover="pack" />
		<Add filter="salt_bridge" />
		<Add mover="min" />
	</PROTOCOLS>
	<OUTPUT scorefxn="r15" />
</ROSETTASCRIPTS>
```

Run the following command (**do not copy it, type it yourself**):

```
$ROSETTA3/bin/rosetta_scripts.linuxgccrelease -database
path/to/rosetta/database -s 1ubq.pdb -parser:protocol filter.xml
-out:prefix filter_ -nstruct 100
```

Running the above, you'll probably find that about 90% of jobs returned a structure, and 10% failed to pass the filter. A filter could have any pass rate, though.

In addition to stopping the run, filters can also be used as metric evaluators. For example, we can make filters to compute the heavy atom RMSD of the sidechains for specific residues. Let's say, for example, that we're interested in the aromatic residues F45 and Y59. We can use the ***SidechainRmsd*** filter. (We'll use the input structure as the reference pose.) The key to using Filters as metric evaluators instead of as trajectory-stoppers is the "confidence" option for all filters. This tells the filter what random fraction of the time it should act as a filter, and for which it should be just a metric evaluator. The default of "1.0" means always act as a filter. If you set this to "0.0" the filter will never filter, instead it will just act like a metric evaluator, meaning that it reports the value of whatever it calculates, but doesn't ever stop a trajectory based on that value.

Let's add some metric-evaluating filters to the script that we just ran:

```
...
        <FILTERS>
                <AtomicDistance  name="salt_bridge"  residue1="11A"  atomtype1="Nlys"
residue2="34A" atomtype2="OOC" distance="3.0" />
                <SidechainRmsd name="F45_rmsd" res1_pdb_num="45A" res2_pdb_num="45A"
include_backbone="1" confidence="0.0" />
                <SidechainRmsd name="Y59_rmsd" res1_pdb_num="59A" res2_pdb_num="59A"
include_backbone="1" confidence="0.0" />
        </FILTERS>
...
        <PROTOCOLS>
                <Add mover="pack" />
                <Add filter="salt_bridge" />
                <Add mover="min" />
                <Add filter="F45_rmsd" />
                <Add filter="Y59_rmsd" />
        </PROTOCOLS>
...
```

A script to do this might look something like the following (a copy of the filter2.xml file is also provided in ***ECP_TUTORIAL_RSCRIPTS/inputs***).

```
<ROSETTASCRIPTS>
	<SCOREFXNS>
		<ScoreFunction name="r15" weights="ref2015" />
	</SCOREFXNS>
	<RESIDUE_SELECTORS>
	</RESIDUE_SELECTORS>
	<TASKOPERATIONS>
		<RestrictToRepacking name="repackonly" />
		<ExtraRotamersGeneric	name="extrachi"	ex1="1"	ex2="1"
ex1_sample_level="1" ex2_sample_level="1" />
	</TASKOPERATIONS>
	<FILTERS>
		<AtomicDistance  name="salt_bridge"  residue1="11A"  atomtype1="Nlys"
residue2="34A" atomtype2="OOC" distance="3.0" />
		<SidechainRmsd name="F45_rmsd" res1_pdb_num="45A" res2_pdb_num="45A"
	include_backbone="1" confidence="0.0" />
		<SidechainRmsd name="Y59_rmsd" res1_pdb_num="59A" res2_pdb_num="59A"
include_backbone="1" confidence="0.0" />
	</FILTERS>
	<MOVERS>
		<MinMover	name="min"	scorefxn="r15"	chi="true"	bb="true"
cartesian="false" />
		<PackRotamersMover		name="pack"		scorefxn="r15"
task_operations="repackonly,extrachi"/>
	</MOVERS>
	<APPLY_TO_POSE>
	</APPLY_TO_POSE>
	<PROTOCOLS>
		<Add mover="pack" />
		<Add filter="salt_bridge" />
		<Add mover="min" />
		<Add filter="F45_rmsd" />
		<Add filter="Y59_rmsd" />
	</PROTOCOLS>
	<OUTPUT scorefxn="r15" />
</ROSETTASCRIPTS>
```

Run the following command (**do not copy it, type it yourself**):

```
$ROSETTA3/bin/rosetta_scripts.linuxgccrelease -database
path/to/rosetta/database -s 1ubq.pdb -parser:protocol filter2.xml
-out:prefix filter2_ -nstruct 100
```

In addition to printing the results of the metric evaluation to the tracer (output log), the results of the filter will be placed in a column of the scorefile. The name of the column is the same as the name of the filter. Additionally, the values for the filters will be written at the end of the PDB file, after the score table.

Conclusion

This tutorial was intended to give you a brief introduction to creating an XML protocol. The process we went through is similar to that used by most RosettaScripts developers when writing an XML file from scratch: protocols are built iteratively, starting with a simple protocol and progressively adding different and more complex stages. For each stage, it's important to have an idea about the effect you wish to accomplish, and then to skim the documentation for existing movers/filters/task operations/etc. which will accomplish it. This may involve multiple RosettaScripts objects, Rosetta modules that require other Rosetta modules as inputs (e.g. movers that require task operations that require residue selectors).

There are, of course, many more RosettaScripts objects than we have discussed, most of which should be covered in the RosettaScripts documentation. There are also additional sections of the XML, which are used for more specialized applications. (For example, grafting a motif, our next tutorial.)

A final note - even if you can create an XML from scratch, it may be easier not to. If you already have an example XML that does something close to what you want to do, it's probably easier to start with that XML, and alter it to add in the functionality you want.

The hard part is not necessarily in putting together the XML, but in determining the optimal protocol (the logical steps) you should use to accomplish your modeling goals, and then in benchmarking that protocol to make sure it does what you hoped.