# Rosetta HTS ligand pre-processing

## Background

Virtual High Throughput screening in Rosetta poses a number of unique challenges. Large numbers of ligands must be prepared and managed. If multiple protein systems are being screened, the appropriate ligands must be matched with the appropriate proteins. Additionally, because Rosetta currently stores ligand information after it is required, the memory requirements of the protocol increase as more ligands are screened. The most straight forward way to address this is to limit the number of ligands screened in each process.

This protocol describes a series of scripts that can be used to rapidly prepare a set of proteins and ligand for virtual high throughput screening.

All scripts referenced can be found in tools/hts_tools

## Prerequisites

This protocol equires the following:

- Python
- The most recent weekly release of Rosetta
- A directory containing conformations of all the proteins in your screening study
- An SDF file containing all the conformers of all the ligands you want to dock. All conformers of the same ligand must have the same name.
  - All conformers of all ligands must have 3D coordinates and hydrogens. You will recieve no errors or warnings if you provide ligands with 2D coordinates or without hydrogens, but you will get very poor ligand docking results.

## Protocol

### Split ligands

The first step of the protocol is to split the ligand file. sdf_split_organize.py will accomplish this task. It takes as input a single sdf file, and will split that file into multiple files, each file containing all the conformers for 1 ligand. Different ligands must have different names in the sdf records, and all conformers for one ligand must have the same name. output filenames are based on the sha1 hash of the input filename, and are placed in a directory hashed structure. Thus, a ligand with the name "Written by BCL::WriteToMDL,CHEMBL29197" will be placed in the path `/41/412d1d751ff3d83acf0734a2c870faaa77c28c6c.mol`. The script will also output a list file in the following format:

```
ligand_id,filename
string,string
ligand_1,path/to/ligand1
ligand_2,path/to/ligand2
```

The list file is a mapping of protein names to sdf file paths.

Many filesystems perform poorly if large numbers of files are stored in the same directory. The hashed directory structure is a method for splitting the generated ligand files across 256 roughly evenly sized subdirectories, improving filesystem peformance.

The script is run as follows:

```
sdf_split_organize.py input.sdf output_dir/ file_list.csv
```

### Create "project database"

The ligand preparation pipline uses an sqlite3 database for organization during the pipeline. The database keeps track of ligand metadata and the locations of ligand files. The project database is created using the following command:

```
setup_screening_project.py file_list.csv output.db3
```

### Append binding information to project database

The next step is to create a binding data file. The binding data file should be in the following format:

```
ligand_id,tag,value
string,string,float
ligand_1,foo,1.5
ligand_2,bar,-3.7
```

The columns are defined as follows:

1. **ligand_id**

   ligand_id is the name of the ligand, which must match the ligand_id in the file_list.csv file created by sdf_split_organize.py.

2. **tag**

   tag is the name of the protein the ligand should be docked into. If a ligand should be docked into multiple proteins, it should have multiple entries in the binding data file. Note that this protocol makes a distinction between protein name, and file name. If you have 4 files: foo_0001.pdb, foo_0002.pdb, bar_0001.pdb, bar_0002.pdb, then you have two proteins with the names foo and bar. The scripts expect that the protein PDB files begin with the protein name.

3. **value**

   value is the activity of the ligand. If you are doing a benchmarking study and know the activity of your ligand, you should enter it here. If you are not doing a benchmarking study, or if ligand activity is not relevant to your study, value can be set to 1.0 (or anything else). This field is currently only used in a few specific Rosetta protocols that are in the experimental stages, and is typically ignored, so it is safe to set arbitrarily in almost every case. Regardless, the scripts require that you provide some value.

Once you have created this file, you can insert it into the project database with the following command:

```
add_activity_tags_to_database.py database.db3 tag_file.csv
```

### Generate params files

The next step is to generate params files. make_params.py is a script which wraps around molfile_to_params.py and generates params files in an automated fashion. params files will be given random names that do not conflict with existing Rosetta residue names (no ligands will be named ALA, for example). This script routinely results in warnings from molfile_to_params.py. This is acceptable. Occasionally, molfile_to_params.py is unable to properly process an sdf file. If this happens, the ligand will be skipped. In order to run make_params.py you need to specify the path to a copy of molfile_to_params.py, as well as the path to the Rosetta database. make_params.py should be run like this:

```
make_params.py -j 4 --database path/to/Rosetta/main/database \
--path_to_params path/to/molfile_to_params.py database.db3 output_dir/
```

In the command line above, the -j option indicates the number of CPU coare which should be used when generating params files. If you are using a multiple core machine, setting -j equal to the number of availible cpu cores.

The script will create a directory params/ containing all params files, pdb files and conformer files.

### Create job files

Because of the memory usage limitations of Rosetta, it is necessary to split the screen up into multiple jobs. The optimal size of each job will depend on the following factors

- The amount of memory availible per CPU
- The number of CPUs being used
- The number of atoms in each ligand
- The number of conformers of each ligand
- The number of protein residues involved in the binding site.

Because of the number of factors that affect RosettaLigand memory usage, it is usually necessary to determine the optimal job size manually. Jobs should be=be small enough to fit into availible memory.

To make this process easier, the make_evenly_grouped_jobs.py script will attempt to group your protein-ligand docking problem into a set of jobs that are sized as evenly possible. The script is run like this:

```
make_evenly_grouped_jobs.py --n_chunks=10 --max_per_job=1000 param_dir/ structure_dir/ output_prefix
```

If the script was run as written above, it would use param files from the directory param_dir/, and structure files from the directory structure_dir/. It would attempt to split the availible protein-ligand docking jobs into 10 evenly grouped job files (–n_chunks). The script will attempt to keep all the docking jobs involving one protein system in one job file. However, if the number of jobs in a group exceeds 1000, the jobs involving that protein system will be split across multiple files (–max_per_job). The script will output the 10 job files with the given prefix, so in the command above, you would get files with names like “output_prefix_01.js”. The script will output to the screen the total number of jobs in each file. All the numbers should be relatively similar. If a job file at the beginning of the list is much larger than the others, it is a sign that ou should reduce the value passed to –max_per_job. If the sizes of all jobs are larger than you want, increase –n_chunks.

**Job file specification** RosettaLigand Job files are JSON files which contain the paths to protein and ligand PDB files, the names of the protein systems, and the params files necessary to load the ligand PDB files. An example file is below:

```
"jobs": [
 {
  "proteins": [
   "set2_28_0001.pdb"
  ],
  "ligands": [
   "A9.pdb"
  ],
  "group_name": "set2_28"
 },
```

```
 {
  "proteins": [
   "set1_42_0001.pdb"
  ],
  "ligands": [
   "AJ.pdb"
  ],
  "group_name": "set1_42"
 }
],
"params": [
 "A9.params",
 "AJ.params"
]
```

**Submit jobs**

Once the jobs have been created they can be input into rosetta using the option -in:file:screening_job_file job_file.js If this option is being used, -s, -l, -list and -in:file:extra_res_fa do not need to be used.