Pull Into Place Documentation

Release 1.2.2

Kale Kundert

Jan 08, 2020

Contents

Pull Into Place (PIP) is a protocol to design protein functional groups with sub-angstrom accuracy. The protocol is based on two ideas: 1) using restraints to define the geometry you're trying to design and 2) using unrestrained simulations to test designs.

The design pipeline orchestrated by PIP has the following steps:

1. Define your project. This entails creating an input PDB file and preparing it for use with rosetta, creating a restraints file that specifies your desired geometry, creating a resfile that specifies which residues are allowed to design, and creating a loop file that specifies where backbone flexibility will be considered:

   ```
   $ pull_into_place 01_setup_workspace ...
   $ pull_into_place 02_setup_model_fragments ...
   ```

2. Build a large number of models that plausibly support your desired geometry by running flexible backbone Monte Carlo simulations restrained to stay near said geometry. The goal is to strike a balance between finding models that are realistic and finding models that satisfy your restraints:

   ```
   $ pull_into_place 03_build_models ...
   ```

3. Filter out models that don't meet your quality criteria:

   ```
   $ pull_into_place 04_pick_models_to_design ...
   ```

4. Generate a number of designs for each model remaining:

   ```
   $ pull_into_place 05_design_models ...
   ```

5. Pick a small number of designs to validate. Typically I generate 100,000 designs and can only validate 50-100. I've found that randomly picking designs according to the Boltzmann weight of their rosetta score gives a nice mix of designs that are good but not too homogeneous:

   ```
   $ pull_into_place 06_pick_designs_to_validate ...
   ```

6. Validate the designs using unrestrained Monte Carlo simulations. Designs that are "successful" will have a funnel on the left side of their score vs rmsd plots:

   ```
   $ pull_into_place 07_setup_design_fragments ...
   $ pull_into_place 08_validate_designs ...
   ```

7. Optionally take the decoys with the best geometry from the validation run (even if they didn't score well) and feed them back into step 4. Second and third rounds of simulation usually produce much better results than the first, because the models being designed are more realistic. Additional rounds of simulation give diminishing returns, and may be more effected by some of rosetta's pathologies (i.e. it's preference for aromatic residues):

   ```
   $ pull_into_place 04_pick_models_to_design ...
   $ pull_into_place 05_design_models ...
   $ pull_into_place 06_pick_designs_to_validate ...
   $ pull_into_place 07_setup_design_fragments ...
   $ pull_into_place 08_validate_designs ...
   ```

8. Generate a report summarizing a variety of quality metrics for each design. This report is meant to help you pick designs to test experimentally:

```
$ pull_into_place 09_compare_best_designs ...
```

CHAPTER 1

Installation

Typically, you would install Pull Into Place (PIP) both on your workstation and on your supercomputing cluster. On your cluster, you would run the steps in the pipeline that involve long rosetta simulations. On your workstation, you would run the analysis and filtering steps between those simulations. The reason for splitting up the work like this is that the analysis scripts have more dependencies than the simulation scripts, and those dependencies can be hard to install on clusters with limited internet access and/or out-of-date software. Some of the analysis scripts also require a GUI environment, which most clusters don't have. Also note that the simulation scripts require `python>=2.6` while the analysis scripts require `python>=2.7`.

Note: Right now, the pipeline will only work as written on the QB3 cluster at UCSF. There are two issues preventing more general use. The first issue is that all of the scripts that submit jobs to the cluster use Sun Grid Engine (SGE) commands to do so. This would be easy to generalize, but so far there hasn't been any need to do so. If you have a need, send an email to the maintainer describing your system and we'll do what we can to support it.

The second issue is that the fragment generation scripts contain a number of hard-coded paths to executables and databases that are specific to the QB3 cluster. For a variety of reasons, fixing this would be a fairly serious undertaking. Nonetheless, please let the maintainer know if you need this done, and we'll do what we can. In the meantime, you can try using simulations that don't require fragments (although these don't perform as well) or generating fragments yourself.

1.1 Installing PIP on your workstation

PIP is available on PyPI, so you can use `pip` to install it. (Sorry if the distinction between PIP and `pip` is confusing. PIP is the Pull Into Place pipeline, `pip` is the package manager distributed with modern versions of python):

```
# For reasons I don't fully understand, some dependencies can't be installed
# unless numpy is manually installed on its own step before everything else.
$ pip install numpy
$ pip install 'pull_into_place [analysis]'
```

The `[analysis]` part of the command instructs `pip` to install all of the dependencies for the analysis scripts. These dependencies aren't installed by default because they aren't needed for the rosetta simulation steps and they can be challenging to install on some clusters.

If the installation worked, this command should print out a nice help message:

```
$ pull_into_place --help
```

Note: If you don't have administrator access on your workstation, or if you just don't want to install PIP system-wide, you can use the `--user` flag to install PIP in your home directory:

```
$ pip install 'pull_into_place [analysis]' --user
```

This will install the PIP executable in `~/.local/bin`, which may not be on your `$PATH` by default. If the installation seemed to work but you get a "command not found" error when trying to run `pull_into_place`, you probably need to add `~/.local/bin` to `$PATH`:

```
echo 'export PATH=~/.local/bin:$PATH' >> ~/.bashrc
source ~/.bashrc
```

1.1.1 GTK and the `plot_funnels` command

The `plot_funnels` command creates an interactive GUI that can show score vs RMSD funnels, open structures corresponding to individual points in `pymol` or `chimera`, and keep track of your notes on different designs.

In order to use this command, you have to install `pygtk` yourself. This dependency is not included with the other `[analysis]` dependencies because it can't be installed with `pip` (except maybe on Windows). On Linux systems, your package manager should be able to install it pretty easily:

```
$ apt-get install python-matplotlib pygtk     # Ubuntu
$ yum install pygtk2                          # Fedora<=21
$ dnf install pygtk2                          # Fedora>=22
```

On Mac systems, the easiest way to do this is to use `homebrew` to install `matplotlib` with the `--with-pygtk` option:

```
$ brew install matplotlib --with-pygtk
```

If you want to install PIP in a virtual environment, you will probably have to create the environment with the `--system-site-packages` flag. This will allow the virtual environment to use the `pygtk` package installed by your package manager.

1.2 Installing PIP on your cluster

If `pip` is available on your cluster, use it:

```
$ pip install pull_into_place
```

Otherwise, you will need to install PIP manually. The first step is to download and install source distributions of `setuptools` and `klab`. PIP needs `setuptools` to install itself and `klab` to access a number of general-purpose tools developed by the Kortemme lab. Once those dependencies are installed, you can download and install a source distribution of pull_into_place. The next section has example command lines for all of these steps in the specific context of the QB3 cluster at UCSF.

1.2.1 Installing PIP on the QB3 cluster at UCSF

Because the UCSF cluster is not directly connected to the internet, it cannot automatically download and install dependencies. Instead, we have to do these steps manually.

1. Download the most recent source distributions for setuptools, klab, and pull_into_place from PyPI (those are links, in case it's hard to tell) onto your workstation. When I did this, the most recent distributions were:

   - setuptools-27.2.0.tar.gz
   - klab-0.3.0.tar.gz
   - pull_into_place-1.2.2.tar.gz

   Note: Three new dependencies were added to setuptools in version 34.0.0: six, packaging, and appdirs. You can either install these dependencies in the same way as the others, or you can just use an earlier version of setuptools.

2. Copy the source distributions onto the cluster:

```
$ scp setuptools-27.2.0.tar.gz chef.compbio.ucsf.edu:
$ scp klab-0.3.0.tar.gz chef.compbio.ucsf.edu:
$ scp pull_into_place-1.2.2.tar.gz chef.compbio.ucsf.edu:
```

3. Log onto the cluster and unpack the source distributions:

```
$ ssh chef.compbio.ucsf.edu
$ tar -xzf setuptools-27.2.0.tar.gz
$ tar -xzf klab-0.3.0.tar.gz
$ tar -xzf pull_into_place-1.2.2.tar.gz
```

4. Install setuptools:

```
$ cd ~/setuptools-27.2.0
$ python setup.py install --user
```

5. Install klab:

```
$ cd ~/klab-0.3.0
$ python setup.py install --user
```

6. Install pull_into_place:

```
$ cd ~/pull_into_place-1.2.2
$ python setup.py install --user
```

7. Make sure `~/.local/bin` is on your `$PATH`:

   The above commands install PIP into `~/.local/bin`. This directory is good because you can install programs there without needing administrator privileges, but it's not on your `$PATH` by default (which means that any programs installed there won't be found). This command modifies your shell configuration file to add `~/.local/bin` to your `$PATH`:

   ```
   $ echo 'export PATH=~/.local/bin:$PATH' >> ~/.bashrc
   ```

   This command reloads your shell configuration so the change takes place immediately (otherwise you'd have to log out and back in):

   ```
   $ source ~/.bashrc
   ```

7. Make sure it works:

   ```
   $ pull_into_place --help
   ```

1.3 Installing Rosetta

PIP also requires Rosetta to be installed, both on your workstation and on your cluster. You can consult this page for more information on how to do this, but in general there are two steps. First, you need to check out a copy of the source code from GitHub:

```
$ git clone git@github.com:RosettaCommons/main.git ~/rosetta
```

Note: You have to be registered as a "Rosetta developer" in order to be able to clone this repository (even if you never plan on developing any C++ code). This page has step-by-step instructions on how to register as a developer. You'll be asked for a password; anyone who's already a developer can give it to you.

Second, you need to compile everything:

```
$ cd ~/rosetta/source
$ ./scons.py bin mode=release -j8
```

Be aware that compiling Rosetta requires a C++11 compiler. This is much more likely to cause problems on your cluster than on your workstation. If you have problems, ask your administrator for help.

1.3.1 Installing Rosetta on the QB3 cluster at UCSF

Installing Rosetta on the QB3 cluster is especially annoying because the cluster has limited access to the internet and outdated versions of both the C++ compiler and python. As above, the first step is to check out a copy of the Rosetta source code from GitHub. This has to be done from one of the interactive nodes (e.g. `iqint`, `optint1`, `optint2`, or `xeonint`) because `chef` and `sous` are not allowed to communicate with GitHub:

```
$ ssh chef.compbio.ucsf.edu
$ ssh iqint
$ git clone git@github.com:RosettaCommons/main.git ~/rosetta
```

The second step is to install the QB3-specific build settings, which specify the path to the cluster's C++11 compiler (among other things):

```
$ ln -s site.settings.qb3 ~/rosetta/source/tools/build/site.settings
```

The final step is to compile Rosetta. This command has several parts: `scl enable python27` causes python2.7 to be used for the rest of the command, which `scons` requires. `nice` reduces the compiler's CPU priority, which helps the shell stay responsive. `./scons.py bin` is the standard command to build Rosetta. `mode=release` tells to compiler to leave out debugging code, which actually makes Rosetta 10x faster. `-j16` tells the compiler that `iqint` has 16 cores for it to use:

```
$ cd ~/rosetta/source
$ scl enable python27 'nice ./scons.py bin mode=release -j16'
```

CHAPTER 2

Pull into Place (PIP) Step-by-Step Tutorial

The most up-to-date version of this tutorial is available on GitHub.

2.1 Model system

For the purposes of this tutorial, we will be examining Ketosteroid isomerase (KSI), which is a model enzyme that catalyzes the rearrangement of a double bond in steroid molecules. Asp38 is the key catalytic residue responsible for taking a proton from one site and moving it to another:

The enzyme is 200x less active when Asp38 is mutated to Glu (D38E), even though Asp and Glu have the same carboxyl (COOH) functional group. The difference is that Glu has one more carbon in its side chain, so the COOH is shifted out of position by just over 1Å. This demo will show everything you would need to do to use Pull Into Place (PIP) to redesign the active site loop to correct the position of the COOH in the D38E mutant.

2.2 Before you start

This demo will demonstrate how to run each of the individual PIP stages. Output will be generated via extremely short trajectories to allow the commands to be run locally on a laptop. For a real problem, PIP will most likely need to be run on a large cluster (as described in the general PIP documentation).

You also need to have Rosetta compiled on your workstation. In general, PIP doesn't care where Rosetta is installed; it just needs a path to the installation. This tutorial will assume that Rosetta is installed in `~/rosetta` such that:

```
$ ls ~/rosetta/source/bin
...
rosetta_scripts
...
```

If your Rosetta installation location varies, you will need to change the ROSETTA= variable at the top of each run script to the correct location (the path should already be correct if you are using the tutorial virtual machine).

Finally, you need to download and extract the loop hash database we'll be using (this is already done in the tutorial virtual machine). Save it (or symlink it) directly into this folder with the folder name loophash_db.

2.3 Input information

All of the required input files have already been generated for you, but here are descriptions of the relevant file types for your future reference:

```
Input PDB file: A structure containing the functional groups to be positioned.
This file should already be parse-able by Rosetta, which often means it must be
stripped of waters and extraneous ligands.

Loops file: A file specifying which backbone regions will be allowed to move.
These backbone regions do not have to be contiguous, but each region must span
at least 4 residues.

Resfile: A file specifying which positions to design and which positions to
repack. It is recommended to design as few residues as possible outside the loops.

Path to resfile: ~/ksi_inputs/resfile

Restraints file: A file describing the geometry you're trying to design.  In
Rosetta parlance, this is more often (inaccurately) called a constraint file.
Note that restraints are not used during the validation step.

Score function: A file that specifies weights for all the terms in the score
function, or the name of a standard Rosetta score function.  The default is
talaris2014.  That should be OK unless you have some particular interaction
(e.g. ligand, DNA, etc.) that you want to score in a particular way.

Build script: An XML Rosetta script that generates backbones capable of
supporting the desired geometry.  The default version of this script uses KIC
with fragments in "ensemble-generation mode" (i.e. no initial build step).

Design script: An XML Rosetta script that performs design (usually on a fixed
backbone) to stabilize the desired geometry.  The default version of this
script uses fixbb.

Validate script: An XML Rosetta script that samples the designed loop to
determine whether the desired geometry is really the global score minimum.  The
default version of this script uses KIC with fragments in "ensemble-generation
mode" (i.e. no initial build step).

Flags file: A file containing command line flags that should be passed to every
invocation of Rosetta for this design.  For example, if your design involves a
ligand, put flags related to the ligand parameter files in this file.
```

2.4 Build initial backbone models

The first actual design step in the pipeline is to generate a large number of backbone models that support the desired side chain geometry. This will be done by running a flexible backbone simulation while applying the restraints we added to the workspace.

You can control which loop modeling algorithm is used for this step by manually editing `build_models.xml`. The current algorithm is loophash KIC (kinematic closure) , which samples conformations from a database of loop conformations hashed based on loop take off and ending points.

With the `fast` flag, which dramatically reduces both the number and length of the simulations, this step should take a few minutes. This (or similar) flags should not be used for production runs, but will continue to use it throughout this demo with the idea that your goal is just to run through the whole pipeline as quickly as possible.

Once the simulations finish, we can download the results to our workstation and visualize them

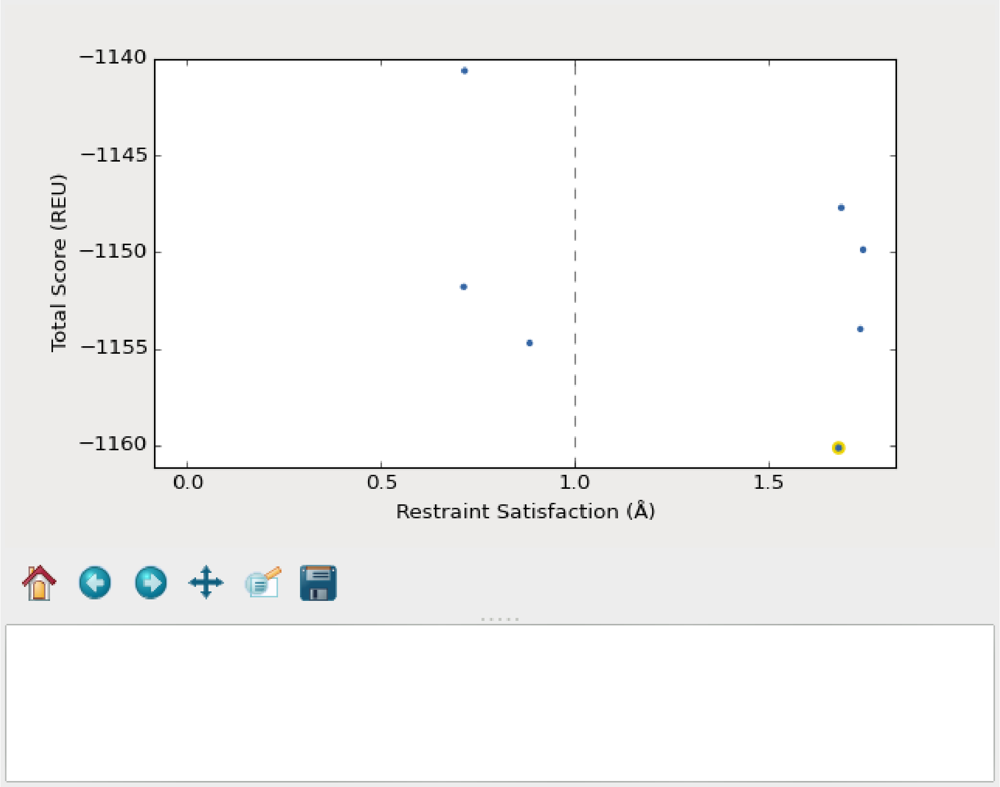

Fig. 1: A screenshot of the `plot_funnels` GUI used in the full PIP pipeline.

Remember that the purpose of this step is to generate physically realistic models with the geometry we want to design. These two goals are somewhat at odds with each other, in the sense that models that are less physically realistic should be able to achieve more ideal geometries. We can use the displayed score vs. restraint satisfaction plot to judge how wells these two goals were balanced. If too many models superimpose with the restraints too well, the restraints might be too strong. If too few models get within 1Å of the restraints, they might be too weak. You can tune the weights of the restraints by manually editing `restraints`.

> **Task 1:** Run the `build_models.sh` script

2.5 Stabilize good backbone models

The next step in the pipeline is to select a limited number of backbone models to carry forward and to generate a number of designed sequences for each of those models. It's worth noting that the first step in the pipeline already did some design, so the purpose of this step is more to quickly generate a diversity of designs than to introduce mutations for the first time.

Task 2: Run the `design_models.sh` script

2.6 Validate good designs

You could have hundreds of thousands of designs after the design step, but it's only really practical to validate about a hundred of those. Due to this vast difference in scale, picking which designs to validate is not a trivial task.

PIP approaches this problem by picking designs with a probability proportional to their Boltzmann-weighted scores. This is naive in the sense that it only considers score (although we are interested in considering more metrics), but more intelligent than simply picking the lowest scores, which tend to be very structurally homogeneous.

In a production run, PIP will open a window to show you how the scores are distributed and which were picked. As the command suggests, it worth looking at the cumulative distribution function (CDF) of the Boltzmann-weighted scores to make sure it's neither too flat nor too sharp. This is a subjective judgment, but one good rule of thumb is that the designs being picked (represented by the circles) should be mostly, but not exclusively, low-scoring. The example CDF below looks about like what you'd want:

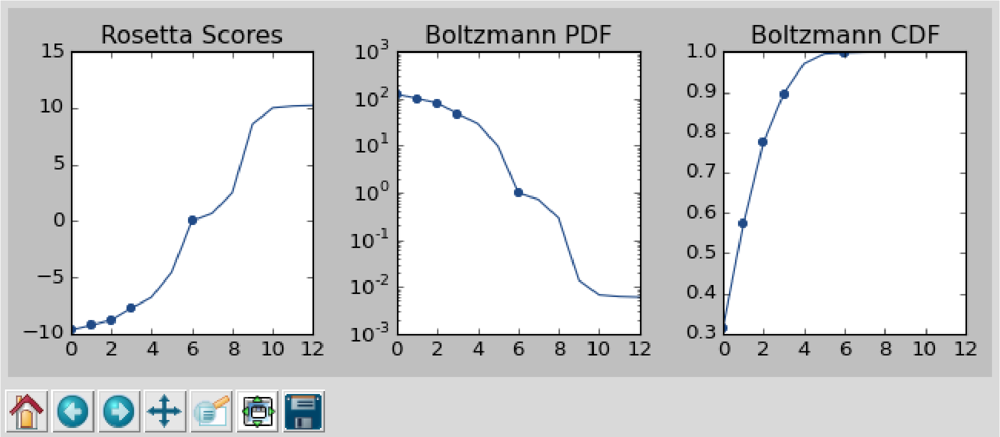

Fig. 2: A screenshot of the `06_pick_designs_to_validate` GUI.

A production validation step consists of 500 independent loop modeling simulations for each design, without restraints. We now will use the standard loop modeling algorithm KIC (instead of loophash KIC as in the initial build step).

Task 3: Run the `validate_designs.sh` script

2.7 Visualize designs

We provide a plugin script that visually compares wild type with mutant structures. The plugin's philosophy is to focus on each mutation one-at-a-time to try to understand what interactions the wild type residue was making and how those interactions are (or are not) being accommodated by the mutant residue. Visit this page for instructions on how to install and use `wt_vs_mut`.

Task 4: Use the `wt_vs_mut` PyMOL script to compare the output structure from the validation step `validate_designs_KSI_D38E_0001.pdb` with the wild type crystal structure `KSI_WT.pdb`. Follow the wt_vs_mut instructions to load the PyMOL visualization plugin (this is already installed in the tutorial virtual machine).

- In your terminal, launch PyMOL and load the two structures: `pymol 4l7k_wt.pdb validate_designs_KSI_D38E_0001.pdb`

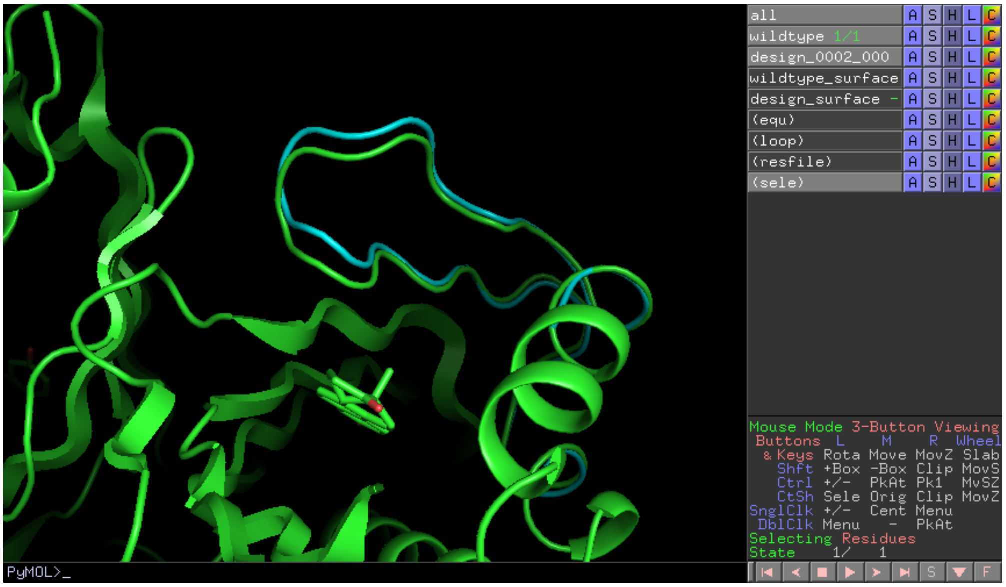

Fig. 3: A screenshot of the PyMOL scene created by the `compare_to_wildtype.sho` script in the full pipeline.

- In PyMOL's terminal:

```
run /path/to/wt_vs_mut.py (is /home/winter/TK/wt_vs_mut/wt_vs_mut.py in the virtual machine)
hide all
show car
show sticks, chain Y and not h.
align validate_designs_KSI_D38E_0001 and chain Y, KSI_WT and chain Y
wt_vs_mut KSI_WT, validate_designs_KSI_D38E_0001
```

2.8 Discussion questions

1. Did you find any mutations in your inspection of your validated design (from step `Validate designs`) that might actually make sense? (Note: we wouldn't expect this to occur after such short simulations, but it is possible).
2. Did you find any mutations that definitely do not look good?

2.9 Compare design structures

We will now compare the validated designs PDB you generated in a short simulation (`validate_designs_KSI_D38E_0001.pdb`) with designs generated from a full run of the PIP protocol.

Task 5: Examine the structures `sample_output/output_A.pdb.gz`, `sample_output/output_B.pdb.gz`, and your `validate_designs_KSI_D38E_0001.pdb` output, using whatever structure visualization methods you prefer. Can you determine if output structure "A" or output structure "B" corresponds to the good validation funnel above?

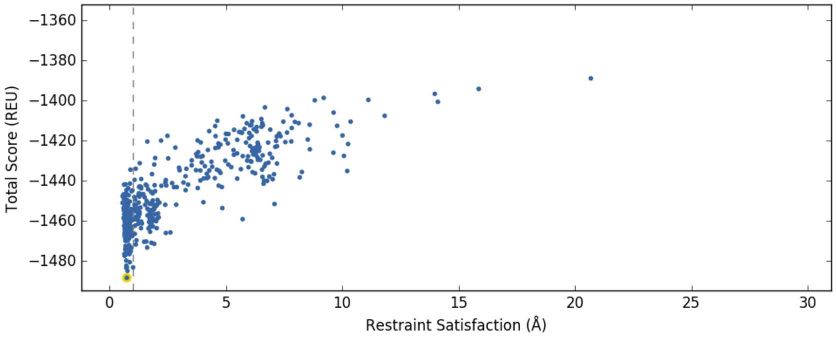

Fig. 4: An example of a validation PIP run that produces a good design funnel, where the singular energy minima (as measured in Rosetta Energy Units) coincides with highest restraint satisfaction.

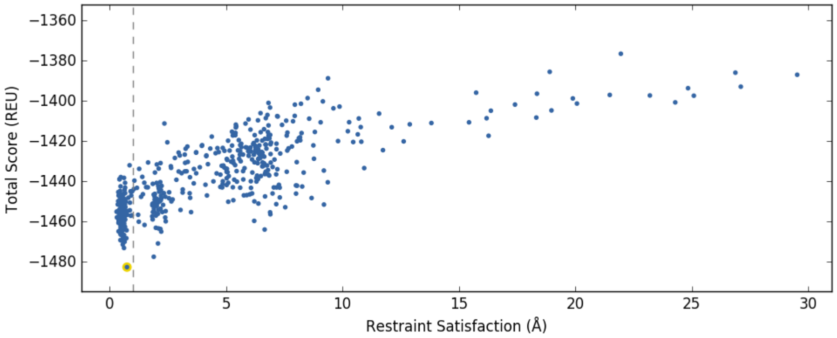

Fig. 5: An example of a validation PIP run that produces a bad design funnel with multiple energy minima.

CHAPTER 3

Rescue the D38E mutation in KSI

Ketosteroid isomerase (KSI) is a model enzyme that catalyzes the rearrangement of a double bond in steroid molecules. Asp38 is the key catalytic residue responsible for taking a proton from one site and moving it to another:

The enzyme is 200x less active when Asp38 is mutated to Glu (D38E), even though Asp and Glu have the same carboxyl (COOH) functional group. The difference is that Glu has one more carbon in its sidechain, so the COOH is shifted out of position by just over 1Å. This demo will show everything you would need to do to use Pull Into Place (PIP) to redesign the active site loop to correct the position of the COOH in the D38E mutant.

Note: This demo assumes that you're running the Rosetta simulations on the QB3 cluster at UCSF. If you're using a different cluster, you may have to adapt some of the commands (most notably the `ssh` ones, but maybe others as well). That said, I believe the `pull_into_place` commands themselves should be the same.

3.1 Before you start

Before starting the demo, you will need to install PIP on both your workstation and your cluster. See the *installation page* for instructions. If you installed PIP correctly, this command should produce a lengthy help message:

```
$ pull_into_place --help
```

You also need to have Rosetta compiled on both your workstation and your cluster. See the *Rosetta section of the installation page* for instructions. In general, PIP doesn't care where rosetta is installed; it just needs a path to the installation. This tutorial will assume that rosetta is installed in `~/rosetta` such that:

```
$ ls ~/rosetta/source/bin
...
rosetta_scripts
...
```

Finally, you need to download the example input files we'll be using onto your cluster. If your cluster has svn installed, you can use this command to download all these files at once:

```
# Run this command on ``iqint`` if you're using the QB3 cluster.
$ svn export https://github.com/Kortemme-Lab/pull_into_place/trunk/demos/ksi/inputs ~/
↪ksi_inputs
```

Otherwise, you can download them by hand onto your workstation and use scp to copy them onto your cluster. You can put the input files wherever you want, but the tutorial will assume that they're in ~/ksi_inputs such that:

```
$ ls ~/ksi_inputs
EQU.cen.params
EQU.fa.params
KSI_D38E.pdb
KSI_WT.pdb
compare_to_wildtype.sho
flags
loops
resfile
restraints
```

3.2 Set up your workspaces

Our first step is to create a workspace for PIP. A workspace is a directory that contains all the inputs and outputs for each simulation. We will call our workspace ~/rescue_ksi_d38e and by the end of this step it will contain all the input files that describe what we're trying to design. It won't (yet) contain any simulation results.

We will also use workspaces to sync files between our workstation and the cluster. The workspace on the cluster will be "normal" and will not know about the one on our workstation. In contrast, the workspace on our workstation will be "remote". It will know about the one on the cluster and will be able to transfer data to and from it.

Note: Pay attention to the ssh chef.compbio.ucsf.edu and exit commands, because they indicate which commands are meant to be run on your workstation and which are meant to be run on the cluster.

The ssh shef.compbio.ucsf.edu command means that you should log onto the cluster and run all subsequent commands on the cluster. The exit command means that you should log off the cluster and run all subsequent commands on your workstation. If you get errors, especially ones that seem to involve version or dependency issues, double check to make sure that you're logged onto the right machine.

First, create the "normal" workstation on the cluster:

```
$ ssh chef.compbio.ucsf.edu   # log onto the cluster
$ pull_into_place 01 rescue_ksi_d38e
Please provide the following pieces of information:

Rosetta checkout: Path to the main directory of a Rosetta source code checkout.
This is the directory called 'main' in a normal rosetta checkout.  Rosetta is
used both locally and on the cluster, but the path you specify here probably
won't apply to both machines.  You can manually correct the path by changing
the symlink called 'rosetta' in the workspace directory.

Path to rosetta: ~/rosetta

Input PDB file: A structure containing the functional groups to be positioned.
```

(continues on next page)

(continued from previous page)

```
This file should already be parse-able by rosetta, which often means it must be
stripped of waters and extraneous ligands.

Path to the input PDB file: ~/ksi_inputs/KSI_D38E.pdb

Loops file: A file specifying which backbone regions will be allowed to move.
These backbone regions do not have to be contiguous, but each region must span
at least 4 residues.

Path to the loops file: ~/ksi_inputs/loops

Resfile: A file specifying which positions to design and which positions to
repack.  I recommend designing as few residues as possible outside the loops.

Path to resfile: ~/ksi_inputs/resfile

Restraints file: A file describing the geometry you're trying to design.  In
rosetta parlance, this is more often (inaccurately) called a constraint file.
Note that restraints are not used during the validation step.

Path to restraints file: ~/ksi_inputs/restraints

Score function: A file that specifies weights for all the terms in the score
function, or the name of a standard rosetta score function.  The default is
talaris2014.  That should be ok unless you have some particular interaction
(e.g. ligand, DNA, etc.) that you want to score in a particular way.

Path to weights file [optional]:

Build script: An XML rosetta script that generates backbones capable of
supporting the desired geometry.  The default version of this script uses KIC
with fragments in "ensemble-generation mode" (i.e. no initial build step).

Path to build script [optional]:

Design script: An XML rosetta script that performs design (usually on a fixed
backbone) to stabilize the desired geometry.  The default version of this
script uses fixbb.

Path to design script [optional]:

Validate script: An XML rosetta script that samples the designed loop to
determine whether the desired geometry is really the global score minimum.  The
default version of this script uses KIC with fragments in "ensemble-generation
mode" (i.e. no initial build step).

Path to validate script [optional]:

Flags file: A file containing command line flags that should be passed to every
invocation of rosetta for this design.  For example, if your design involves a
ligand, put flags related to the ligand parameter files in this file.

Path to flags file [optional]: ~/ksi_inputs/flags

Setup successful for design 'rescue_ksi_d38e'.
```

Note: You don't need to type out the full names of PIP subcommands, you just need to type enough to be unambiguous. So `pull_into_place 01` is the same as `pull_into_place 01_setup_workspace`.

You may have noticed that we were not prompted for all of the files we downloaded. `EQU.cen.params` and `EQU.fa.params` are ligand parameters for centroid and fullatom mode, respectively. PIP doesn't specifically ask for ligand parameter files, but we still need them for our simulations because we referenced them in `flags`:

```
$ cat ~/rescue_ksi_d38e/flags
-extra_res_fa EQU.fa.params
-extra_res_cen EQU.cen.params
```

The paths in `flags` are relative to the workspace directory, because PIP sets the current working directory to the workspace directory for every simulation it runs. Therefore, in order for these paths to be correct, we have to manually copy the ligand parameters files into the workspace:

```
$ cp ~/ksi_inputs/EQU.*.params ~/rescue_ksi_d38e
$ exit   # log off the cluster and return to your workstation
```

`KSI_WT.pdb` is the structure of the wildtype KSI enzyme and `compare_to_wildtype.sho` is a script that configures and displays scenes in pymol that compare design models against `KSI_WT.pdb`. PIP itself doesn't need these files, but we will use them later on to evaluate designs. For now just copy them into the workspace:

```
$ cp ~/ksi_inputs/KSI_WT.pdb ~/rescue_ksi_d38e
$ cp ~/ksi_inputs/compare_to_wildtype.sho ~/rescue_ksi_d38e
```

`build_models.xml`, `design_models.xml`, and `validate_design.xml` are RosettaScripts that run the various simulations that make up the PIP pipeline. We don't need to do anything with these files (they're for a different tutorial) because the `01_setup_workspace` command already copied default versions of each into the workspace. We could provide different scripts if we wanted to change any of the algorithms PIP uses — for example, if we wanted to use a less aggressive backbone move like Backrub for a smaller remodeling problem, or if we wanted to use a design method like HBNet to prioritize getting a good H-bond network — but for this tutorial the defaults are fine.

Now that the workspace on the cluster is all set up, we can make the "remote" workspace on our workstation that will link to it:

```
$ cd ~
$ pull_into_place 01 -r rescue_ksi_d38e
Please provide the following pieces of information:

Rosetta checkout: Path to the main directory of a Rosetta source code checkout.
This is the directory called 'main' in a normal rosetta checkout.  Rosetta is
used both locally and on the cluster, but the path you specify here probably
won't apply to both machines.  You can manually correct the path by changing
the symlink called 'rosetta' in the workspace directory.

Path to rosetta: ~/rosetta

Rsync URL: An ssh-style path to the directory that contains (i.e. is one level
above) the remote workspace.  This workspace must have the same name as the
remote one.  For example, to link to "~/path/to/my_design" on chef, name this
workspace "my_design" and set its rsync URL to "chef:path/to".

Path to project on remote host: chef.compbio.ucsf.edu:
```

(continues on next page)

(continued from previous page)

```
receiving incremental file list
./
EQU.cen.params
EQU.fa.params
build_models.xml
design_models.xml
flags
input.pdb.gz
loops
resfile
restraints
scorefxn.wts
validate_designs.xml
workspace.pkl

sent 322 bytes  received 79,420 bytes  31,896.80 bytes/sec
total size is 78,647  speedup is 0.99
```

If this command was successful, all of the input files from the cluster, even the ligand parameters, will have been automatically copied from the cluster to your workstation. This workspace is also properly configured for you to use `pull_into_place push_data` and `pull_into_place fetch_data` to copy data to and from the cluster.

3.3 Build initial backbone models

The first actual design step in the pipeline is to generate a large number of backbone models that support the desired sidechain geometry. This will be done by running a flexible backbone simulation while applying the restraints we added to the workspace.

You can control which loop modeling algorithm is used for this step by manually editing `build_models.xml`. The default algorithm is kinematic closure (KIC) with fragments, which samples conformations from a fragment library and uses KIC to keep the backbone closed. This algorithm was chosen for its ability to model large conformational changes, but it does require us to make a fragment library before we can run the model-building simulation:

```
$ ssh chef.compbio.ucsf.edu
$ pull_into_place 02_setup_model_fragments rescue_ksi_d38e
```

Note: Generating fragment libraries is the most fragile part of the pipeline. It only works on the QB3 cluster at UCSF, and even there it breaks easily. If you have trouble with this step, you can consider using a loop modeling algorithm that doesn't require fragments.

This step should take about an hour. Once it finishes, we can generate our models:

```
$ pull_into_place 03 rescue_ksi_d38e --test-run
$ exit
```

With the `--test-run` flag, which dramatically reduces both the number and length of the simulations, this step should take about 30 minutes. This flag should not be used for production runs, but I will continue to use it throughout this demo with the idea that your goal is just to run through the whole pipeline as quickly as possible.

Once the simulations finish, we can download the results to our workstation and visualize them:

```
$ pull_into_place fetch_data rescue_ksi_d38e
$ pull_into_place plot_funnels rescue_ksi_d38e/01_restrained_models/outputs
```

Note: On Mac OS, you may have to give the `plot_funnels` command the `-F` flag. This flag prevents the GUI from detaching from the terminal and running in a background process. This behavior is normally convenient because it allows you to keep using the terminal while the GUI is open, but on Mac OS it seems to cause problems.

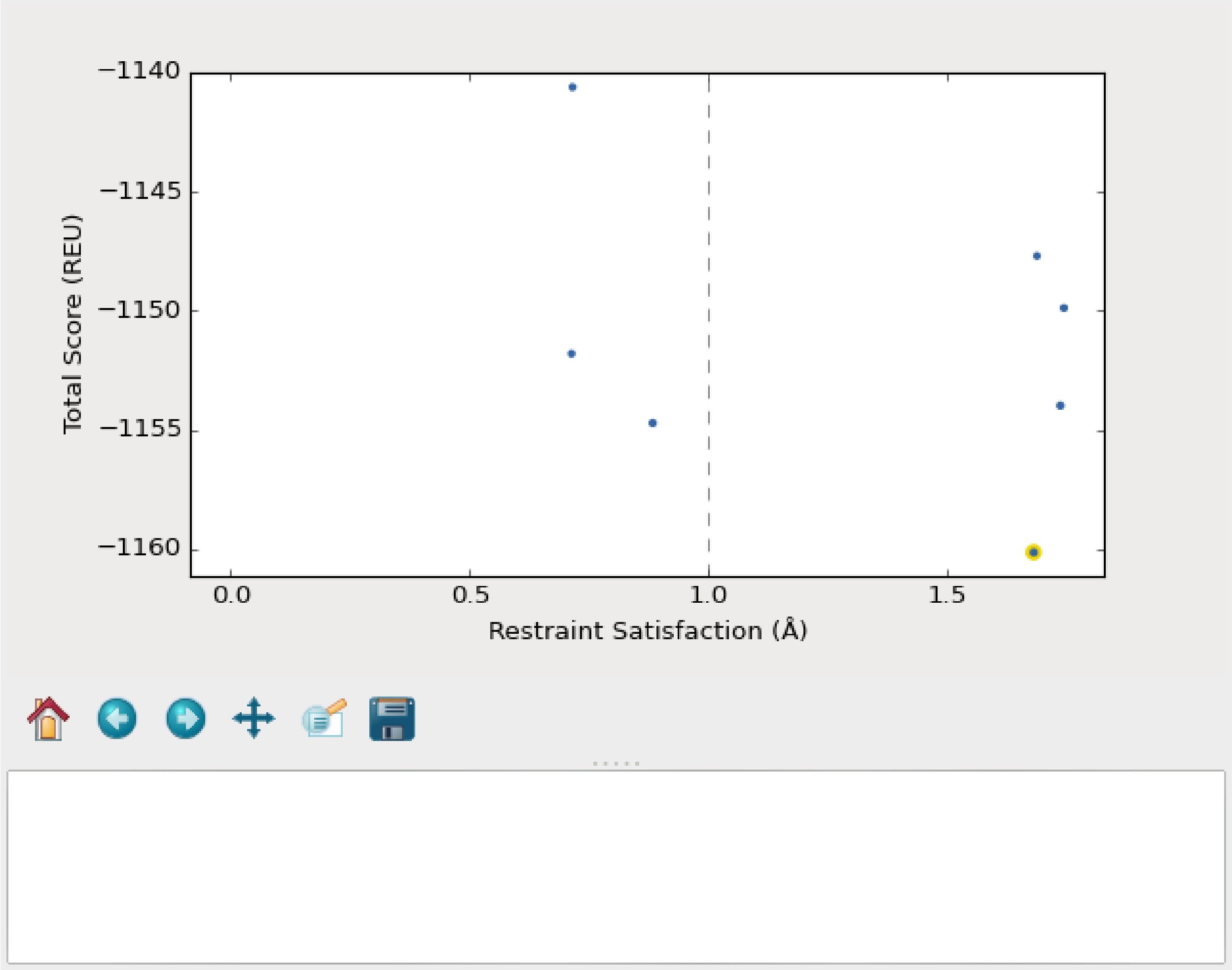

Fig. 1: A screenshot of the `plot_funnels` GUI.

Remember that the purpose of this step is to generate physically realistic models with the geometry we want to design. These two goals are somewhat at odds with each other, in the sense that models that are less physically realistic should be able to achieve more ideal geometries. The second command displays a score vs. restraint satisfaction plot that we can use to judge how wells these two goals were balanced. If too many models superimpose with the restraints too well, the restraints might too strong. If too few models get within 1Å of the restraints, they might be to weak. You can tune the weights of the restraints by manually editing `shared_defs.xml`.

3.4 Stabilize good backbone models

The next step in the pipeline is to select a limited number of backbone models to carry forward and to generate a number of designed sequences for each of those models. It's worth noting that the first step in the pipeline already did some design, so the purpose of this step is more to quickly generate a diversity of designs than to introduce mutations for the first time.

The following command specifies that we want to carry forward any model that puts its Glu within 1.0Å of where we restrained it to be:

```
$ pull_into_place 04 rescue_ksi_d38e 1 'restraint_dist < 1.0'
Selected 4 of 8 models
```

Note: This command just makes symlinks from the output directory of the model building command to the input directory of the design command. The models that aren't selected aren't deleted, and you run this command more than once if you change your mind about which models you want to keep.

This is a very relaxed threshold because we used `--test-run` in the previous step and don't have very many models to pick from. For a production run, I would try to set the cutoff close to 0.6Å while still keeping a couple thousand models. You can also eliminate models based on total score and a number of other metrics; use the `--help` flag for more information.

Also note that we had to specify the round "1" after the name of the workspace. In fact, most of the commands from here on out will expect a round number. This is necessary because, later on, we will be able to start new rounds of design by picking models from the results of validation simulations. We're currently in round 1 because we're still making our first pass through the pipeline.

Once we've chosen which models to design, we need to push that information to the cluster:

```
$ pull_into_place push rescue_ksi_d38e
```

Then we can log into the cluster and start the design simulations:

```
$ ssh chef.compbio.ucsf.edu
$ pull_into_place 05 rescue_ksi_d38e 1 --test-run
$ exit
```

With the `--test-run` flag, this step should take about 30 min. When the design simulations are complete, we can download the results to our workstation as before:

```
$ pull_into_place fetch_data rescue_ksi_d38e
```

3.5 Validate good designs

You could have hundreds of thousands of designs after the design step, but it's only really practical to validate about a hundred of those. Due to this vast difference in scale, picking which designs to validate is not a trivial task.

PIP approaches this problem by picking designs with a probability proportional to their Boltzmann-weighted scores. This is naive in the sense that it only considers score (although we are interested in considering more metrics), but more intelligent than simply picking the lowest scores, which tend to be very structurally homogeneous:

```
$ pull_into_place 06_pick rescue_ksi_d38e 1 -n5
Total number of designs:       39
    minus duplicate sequences: 13
    minus current inputs:      13

Press [enter] to view the designs that were picked and the distributions that
were used to pick them.  Pay particular attention to the CDF.  If it is too
flat, the temperature (T=2.0) is too high and designs are essentially being
picked randomly.  If it is too sharp, the temperature is too low and only the
```

(continues on next page)

(continued from previous page)

```
highest scoring designs are being picked.

Accept these picks? [Y/n] y
Picked 5 designs.
```

This command will open a window to show you how the scores are distributed and which were picked. As the command suggests, it worth looking at the cumulative distribution function (CDF) of the Boltzmann-weighted scores to make sure it's neither too flat nor too sharp. This is a subjective judgment, but one good rule of thumb is that the designs being picked (represented by the circles) should be mostly, but not exclusively, low-scoring. The CDF below looks about like what you'd want:

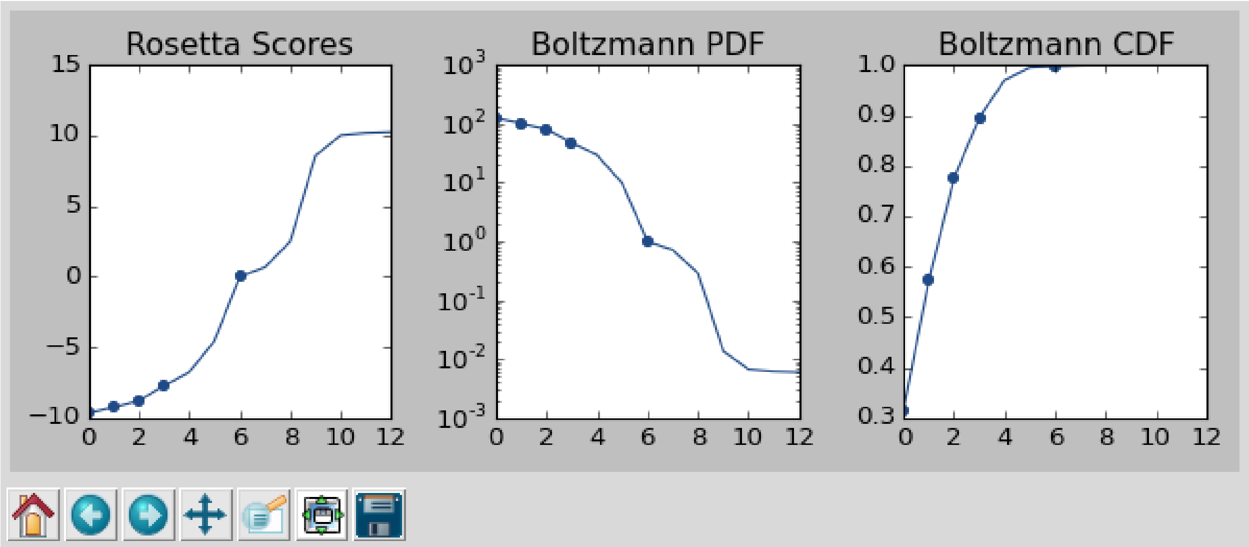

Fig. 2: A screenshot of the `06_pick_designs_to_validate` GUI.

The `-n5` argument instructs PIP to pick 5 designs to validate. The default is 50, which would be appropriate for a production run. However, in this demo we only have about 50 designs because we've been using the `--test-run` flag. The algorithm that picks from a Boltzmann weighted distribution gets very slow when the number of designs to pick is close to the number of designs to pick from, which is why we only pick 5.

It's also worth noting that there is a `06_manually_pick_designs_to_validate` command that you can use if you have a PDB file with a specific mutation (perhaps that you made in pymol) that you want to validate. This is not normally part of the PIP pipeline, though:

```
$ pull_into_place 06_man rescue_ksi_d38e 1 path/to/manual/design.pdb
```

We can push our picks to the cluster in the same way as before:

```
$ pull_into_place push rescue_ksi_d38e
```

The validation step consists of 500 independent loop modeling simulations for each design, without restraints. As with the model building step, the default algorithm is KIC with fragments and we need to create fragment libraries before we can start the simulations:

```
$ ssh chef.compbio.ucsf.edu
$ pull_into_place 07 rescue_ksi_d38e 1
```

Once the fragment libraries have been created (as before, this should take about an hour), we can run the validation simulations:

```
$ pull_into_place 08 rescue_ksi_d38e 1 --test-run
$ exit
```

We could wait for the simulations to finish (which as before will take about 30 min) then download the results to our workstation using the same `fetch_data` command as before. However, I generally prefer to use the following command to automatically download and cache the results from the validation simulations as they're running:

```
$ pull_into_place fetch_and_cache rescue_ksi_d38e/03_validated_designs_round_1/
↪outputs --keep-going
```

The simulations in production runs generate so much data that it can take several hours just to download and parse it all. This command gets rid of that wait by checking to see if any new data has been generated, and if it has, downloading it, parsing it, and caching the information that the rest of the pipeline will need to use. The `--keep-going` flag tells the command to keep checking for new data over and over again until you hit `Ctrl-C`, otherwise it would check once and stop.

Once we've downloaded all the data, we can use the `plot` command again to visualize the validation results:

```
$ pull_into_place plot rescue_ksi_d38e/03_validated_designs_round_1/outputs/*
```

The `plot` GUI has a number of features that can help you delve into your simulation results and find good designs. First, notice that there is a panel on the left listing all of the designs that were validated. You can click on a design to view the results for that design. You can also hit `j` and `k` to quickly scroll through the designs.

Second, notice that there is a place to take notes on the current design below the plot. The search bar in the top left can be used to filter designs based on these notes. One convention that I find useful is to mark designs with +, ++, +++, etc. depending on how much I like them, so I can easily select interesting designs by searching for the corresponding number of + signs.

Third, you can view the model corresponding to any particular point by right-clicking (or Ctrl-clicking) on that point and choosing one of the options in the menu that appears. For example, try choosing "Compare to wildtype". Behind the scenes, this runs the `compare_to_wildtype.sho` script that we copied into the workspace with the path to the selected model as the first and only argument. That script then runs `pymol` with the design superimposed on the wildtype structure, a number of useful selections pre-defined, the proteins rendered as cartoons, the ligands rendered as sticks, and the camera positioned with a good vantage point of the active-site loop.

Within `pymol`, I use a plugin I wrote called `wt_vs_mut` to see how the design model differs from the wildtype structure. The plugin's philosophy is to focus on each mutation one-at-a-time, to try to understand what interactions the wildtype residue was making and how those interactions are (or are not) being accommodated by the mutant residue. If this sounds useful to you, visit this page for instructions on how to install and use `wt_vs_mut`. "Compare to wildtype" pre-loads a shortcut to run `wt_vs_mut` with the correct arguments, so once you have the plugin installed, you can simply type `ww` to run it.

The `plot` command has several other features and hotkeys that aren't described here, so you may find it worthwhile to read its complete help text:

```
$ pull_into_place plot --help
```

3.6 Iterate the design process

Often, some of the models from the validation simulations will fulfill the design goal really well despite not scoring very well. These models are promising because they're clearly capable of supporting the desired geometry, and they may just need another round of design to make the conformation in question the most favorable.

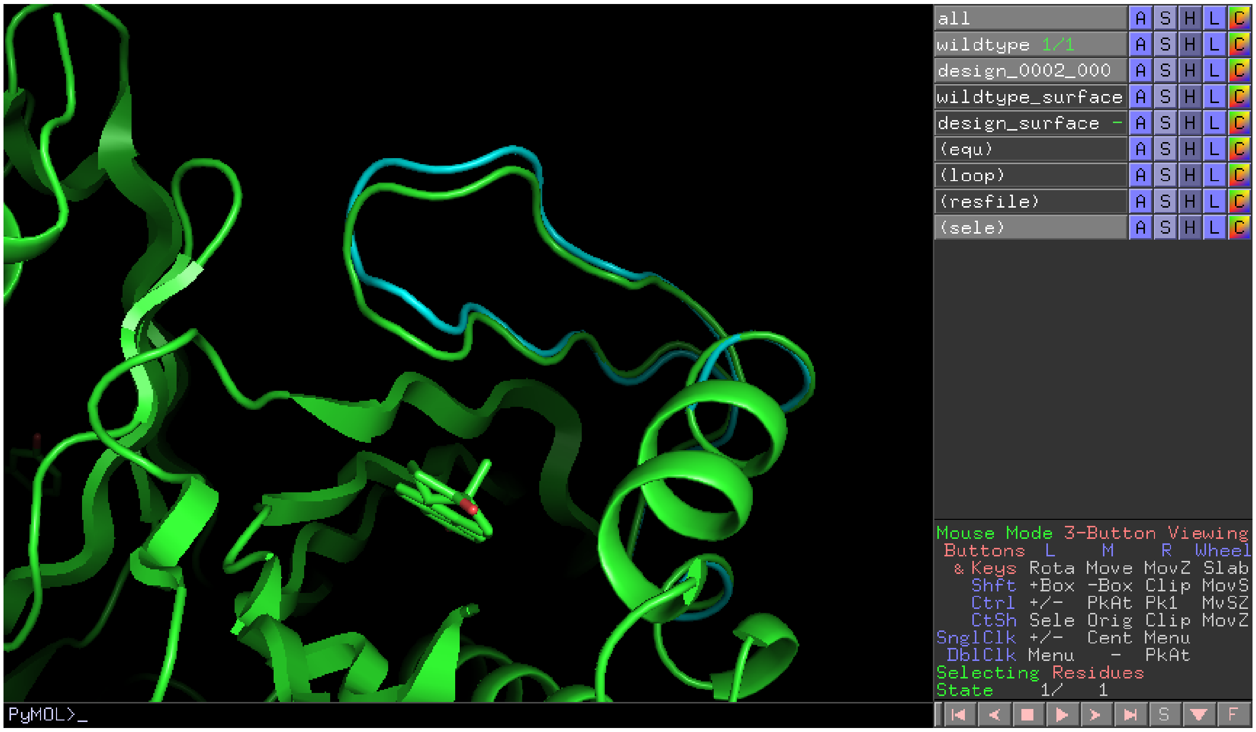

Fig. 3: A screenshot of the pymol scene created by `compare_to_wildtype.sho`.

You can use the `04_pick_models_to_design` command to pick models from the validation simulations to redesign. The command has exactly the same form as when we used it after the model building step, we just need to specify that we're in round 2 instead of round 1:

```
$ pull_into_place 04 rescue_ksi_d38e 2 'restraint_dist < 1'
```

I won't repeat the remaining commands in the pipeline, but they're exactly the same as before, with the round number updated as appropriate.

For a production run, I would recommend doing at least two rounds of design. I believe that models from the validation simulations – which are the basis for the later rounds of design – are more relaxed than the initial models, which makes them better starting points for design. At the same time, I would recommend against doing more than three or four rounds of design, because iterated cycles of backbone optimization and design seem to provoke artifacts in Rosetta's score function.

3.7 Pick designs to test experimentally

The final step in the PIP pipeline is to interpret the results from the validation simulations and to choose an experimentally tractable number of designs to test. The primary results from the validation simulations are the score vs. restraint satisfaction plots. Promising designs will have distinct "funnels" in these plots: the models with the best geometries (i.e. restraint satisfaction) will also be the most stable (i.e. Rosetta score).

However, there are other factors we might want to consider as well. For example, you might be suspicious of designs with large numbers of glycine, proline, or aromatic mutations. You might also want to know which designs are the most similar to each other – either in terms of sequence or structure – so you can avoid wasting time testing two designs that are nearly identical. Finally, you might be interested in some of general-purpose metrics of protein stability that are not well represented by score alone, like the number of buried unsatisfied H-bonds or the amount of strain in the designed sidechains.

The following command generates a spreadsheet that collects all this information in one place:

```
$ pull_into_place 09 rescue_ksi_d38e 1
```

This command will create a file called `quality_metrics.xlsx` that you can open with Excel or any other spreadsheet program. By default, the spreadsheet will only include entries for designs where the lowest scoring model is within 1.2Å of satisfying the restraints. Each column presents a different quality metric, and each cell is colored according to how favorable that value of that metric is. The precise meaning and interpretation of each metric is discussed below:

Resfile Sequence Show the amino acid identity of every position that was allowed to mutate in the design (although not all of the positions are necessarily different from wildtype). Use this information to look for specific sequence motifs that make you suspicious.

Sequence Cluster Show which designs have the most similar sequences. Only positions that were allowed to design are considered in this clustering, and no alignment is done. The sequences are simply compared using a score matrix like BLOSUM80. Use this metric to avoid picking too many designs that are too similar to each other.

Struct Cluster Show which designs are the most structurally similar. This metric works by creating a hierarchical clustering of the lowest scoring models for each design based on loop backbone RMSD. Clusters are then made such that every member in every cluster is within a certain loop RMSD of all its peers. Use this metric to avoid picking too many designs that are too similar to each other.

Restraint Dist (Å) Show how well each design satisfies the design goal, as embodied by the restraints given at the very beginning of the pipeline.

Score Gap (REU) Show the difference in score between the lowest scoring models with restraint distances less than and greater than 1Å and 2Å, respectively. Use this metric to get a rough idea of how deep the score vs. RMSD funnel is for each design.

% Subangstrom Show what percent of the models from the validation simulations had sub-angstrom restraint distances. Use this metric to get a rough idea of how well-populated the score vs. RMSD funnel is.

# Buried Unsats Show how many buried unsatisfied H-bonds each design has, relative to the input structure given at the very beginning of the pipeline. This is something that's not accounted for by the Rosetta score function, but that can do a very good job discriminating reasonable backbones from horrible ones.

Dunbrack (REU) Show the Dunbrack score for each residue that was part of the design goal (i.e. was restrained in the building step). High Dunbrack scores indicate unlikely sidechain conformations.

CHAPTER 4

Command Usage

```
Pull Into Place (PIP) is a protocol to design protein functional groups with
sub-angstrom accuracy.  The protocol is based on two ideas: 1) using restraints
to define the geometry you're trying to design and 2) using an unrestrained
simulations to test designs.

Usage:
    pull_into_place <command> [<args>...]
    pull_into_place --version
    pull_into_place --help

Arguments:
    <command>
        The name of the command you want to run.  You only need to specify
        enough of the name to be unique.  Broadly speaking, there are two
        categories of scripts.  The first are part of the main design pipeline.
        These are prefixed with numbers so that you know the order to run them
        in.  The second are helper scripts and are not prefixed.

        01_setup_workspace                      cache_models
        02_setup_model_fragments                count_models
        03_build_models                         fetch_and_cache_models
        04_pick_models_to_design                fetch_data
        05_design_models                        make_web_logo
        06_manually_pick_designs_to_validate   plot_funnels
        06_pick_designs_to_validate             push_data
        07_setup_design_fragments
        08_validate_designs
        09_compare_best_designs

    <args>...
        The necessary arguments depend on the command being run.  For more
        information, pass the '--help' flag to the command you want to run.

Options:
```

(continues on next page)

(continued from previous page)

```
    -v, --version
        Display the version of PIP that's installed.

    -h, --help
        Display this help message.

PIP's design pipeline has the following steps:

1. Define your project.  This entails creating an input PDB file and preparing
   it for use with rosetta, creating a restraints file that specifies your
   desired geometry, creating a resfile that specifies which residues are
   allowed to design, and creating a loop file that specifies where backbone
   flexibility will be considered.

   $ pull_into_place 01_setup_workspace ...

2. Build a large number of models that plausibly support your desired geometry
   by running flexible backbone Monte Carlo simulations restrained to stay near
   said geometry.  The goal is to find a balance between finding models that
   are realistic and that satisfy your restraints.

   $ pull_into_place 02_setup_model_fragments ...
   $ pull_into_place 03_build_models ...

3. Filter out models that don't meet your quality criteria.

   $ pull_into_place 04_pick_models_to_design ...

4. Generate a number of designs for each model remaining.

   $ pull_into_place 05_design_models ...

5. Pick a small number of designs to validate.  Typically I generate 100,000
   designs and can only validate 50-100.  I've found that randomly picking
   designs according to the Boltzmann weight of their rosetta score gives a
   nice mix of designs that are good but not too homogeneous.

   $ pull_into_place 06_pick_designs_to_validate ...

6. Validate the designs using unrestrained Monte Carlo simulations.  Designs
   that are "successful" will have a funnel on the left side of their score vs
   rmsd plots.

   $ pull_into_place 07_setup_design_fragments ...
   $ pull_into_place 08_validate_designs ...

7. Optionally take the decoys with the best geometry from the validation run
   (even if they didn't score well) and feed them back into step 3.  Second and
   third rounds of simulation usually produce much better results than the
   first, because the models being designed are more realistic.  Additional
   rounds of simulation give diminishing returns, and may be more effected by
   some of rosetta's pathologies (i.e. it's preference for aromatic residues).

   $ pull_into_place 04_pick_models_to_design ...
   $ pull_into_place 05_design_models ...
   $ pull_into_place 06_pick_designs_to_validate ...
   $ pull_into_place 07_setup_design_fragments ...
```

(continues on next page)

(continued from previous page)

```
   $ pull_into_place 08_validate_designs ...

8. Generate a report summarizing a variety of quality metrics for each design.
   This report is meant to help you pick designs to test experimentally.

   $ pull_into_place 09_compare_best_designs ...
```

4.1 Step 1: Setup workspace

```
Query the user for all the input data needed for a design.  This includes a
starting PDB file, the backbone regions that will be remodeled, the residues
that will be allowed to design, and more.  A brief description of each field is
given below.  This information is used to build a workspace for this design
that will be used by the rest of the scripts in this pipeline.

Usage:
    pull_into_place 01_setup_workspace <workspace> [--remote] [--overwrite]

Options:
    --remote, -r
        Setup a link to a design directory on a remote machine, to help with
        transferring data between a workstation and a cluster.  Note: the
        remote and local design directories must have the same name.

    --overwrite, -o
        If a design with the given name already exists, remove it and replace
        it with the new design created by this script.
```

4.2 Step 2: Setup model fragments

```
Generate fragments for the initial model building simulations.  Note that it's
a little bit weird to use fragments even though the models are allowed to
design in these simulations.  Conformations that are common for the current
sequence but rare for the original one might not get sampled.  However, we
believe that the improved sampling that fragments offer outweighs this
potential drawback.

Usage:
    pull_into_place 02_setup_model_fragments <workspace> [options]

Options:
    -L, --ignore-loop-file
        Generate fragments for the entire input structure, not just for the
        region that will be remodeled as specified in the loop file.  This is
        currently necessary only if multiple loops are being remodeled.

    -m, --mem-free=MEM  [default: 2]
        The amount of memory (GB) to request from the cluster.  Bigger systems
        may need more memory, but making large memory requests can make jobs
        take much longer to come off the queue (since there may only be a few
        nodes with enough memory to meet the request).
```

(continues on next page)

(continued from previous page)

```
    -d, --dry-run
        Print out the command-line that would be used to generate fragments,
        but don't actually run it.
```

4.3 Step 3: Build models

```
Build models satisfying the design goal.  Only the regions of backbone
specified by the loop file are allowed to move and only the residues specified
in the resfile are allowed to design.  The design goal is embodied by the
restraints specified in the restraints file.

Usage:
    pull_into_place 03_build_models <workspace> [options]

Options:
    --nstruct NUM, -n NUM   [default: 10000]
        The number of jobs to run.  The more backbones are generated here, the
        better the rest of the pipeline will work.  With too few backbones, you
        can run into a lot of issues with degenerate designs.

    --max-runtime TIME      [default: 12:00:00]
        The runtime limit for each model building job.

    --max-memory MEM        [default: 1G]
        The memory limit for each model building job.

    --test-run
        Run on the short queue with a limited number of iterations.  This
        option automatically clears old results.

    --clear
        Clear existing results before submitting new jobs.
```

4.4 Step 4: Pick models to design

```
Pick backbone models from the restrained loopmodel simulations to carry on
though the rest of the design pipeline.  The next step in the pipeline is to
search for the sequences that best stabilize these models.  Models can be
picked based on number of criteria, including how well the model satisfies the
given restraints and how many buried unsatisfied H-bonds are present in the
model.  All of the criteria that can be used are described in the "Queries"
section below.

Usage:
    pull_into_place 04_pick_models_to_design [options]
        <workspace> <round> <queries>...

Options:
    --clear, -x
        Remove any previously selected "best" models.
```

(continues on next page)

(continued from previous page)

```
    --recalc, -f
        Recalculate all the metrics that will be used to choose designs.

    --dry-run, -d
        Choose which models to pick, but don't actually make any symlinks.

Queries:
    The queries provided after the workspace name and round number are used to
    decide which models to carry forward and which to discard.  Any number of
    queries may be specified; only models that satisfy each query will be
    picked.  The query strings use the same syntax of the query() method of
    pandas DataFrame objects, which is pretty similar to python syntax.
    Loosely speaking, each query must consist of a criterion name, a comparison
    operator, and a comparison value.  Only 5 criterion names are recognized:

    "restraint_dist"
        The average distance between all the restrained atoms and their target
        positions in a model.
    "loop_dist"
        The backbone RMSD of a model relative to the input structure.
    "buried_unsat_score"
        The change in the number of buried unsatisfied H-bonds in a model
        relative to the input structure.
    "dunbrack_score"
        The average Dunbrack score of any sidechains in a model that were
        restrained during the loopmodel simulation.
    "total_score"
        The total score of a model.

    Some example query strings:

    'restraint_dist < 0.6'
    'buried_unsat_score <= 4'
```

4.5 Step 5: Design models

```
Find sequences that stabilize the backbone models built previously.  The same
resfile that was used for the model building step is used again for this step.
Note that the model build step already includes some design.  The purpose of
this step is to expand the number of designs for each backbone model.

Usage:
    pull_into_place 05_design_models <workspace> <round> [options]

Options:
    --nstruct NUM, -n NUM   [default: 100]
        The number of design jobs to run.

    --max-runtime TIME      [default: 0:30:00]
        The runtime limit for each design job.  The default value is
        set pretty low so that the short queue is available by default.  This
        should work fine more often than not, but you also shouldn't be
        surprised if you need to increase this.
```

(continues on next page)

(continued from previous page)

```
    --max-memory MEM        [default: 1G]
        The memory limit for each design job.

    --test-run
        Run on the short queue with a limited number of iterations.  This
        option automatically clears old results.

    --clear
        Clear existing results before submitting new jobs.
```

4.6 Step 6: Pick designs to validate

```
Pick a set of designs to validate.  This is actually a rather challenging task
because so few designs can be validated.  Typically the decision is made based
on sequence identity and rosetta score.  It might be nice to add a clustering
component as well.

Usage:
    pull_into_place 06_pick_designs_to_validate
            <workspace> <round> [<queries>...] [options]

Options:
    --num NUM, -n NUM           [default: 50]
        The number of designs to pick.  The code can gets stuck and run for a
        long time if this is too close to the number of design to pick from.

    --temp TEMP, -t TEMP        [default: 2.0]
        The parameter controlling how often low scoring designs are picked.

    --clear, -x
        Forget about any designs that were previously picked for validation.

    --recalc, -f
        Recalculate all the metrics that will be used to choose designs.

    --dry-run
        Don't actually fill in the input directory of the validation workspace.
        Instead just report how many designs would be picked.
```

4.7 Step 6': Manually pick designs to validate

```
Manually provide designs to validate.

The command accepts any number of pdb files, which should already contain the
mutations you want to test.  These files are simply copied into the workspace
in question.  The files are copied (not linked) so they're less fragile and
easier to copy across the network.

Usage:
    pull_into_place 06_manually_pick_designs_to_validate [options]
```

(continues on next page)

(continued from previous page)

```
        <workspace> <round> <pdbs>...

Options:
    --clear, -x
        Forget about any designs that were previously picked for validation.
```

4.8 Step 7: Setup design fragments

```
Generate fragments for the design validation simulations.  Each design has a
different sequence, so each input needs its own fragment library.  You can skip
this step if you don't plan to use fragments in your validation simulations,
but other algorithms may not perform as well on long loops.

Usage:
    pull_into_place 07_setup_design_fragments <workspace> <round> [options]

Options:
    -m, --mem-free=MEM  [default: 2]
        The amount of memory (GB) to request from the cluster.  Bigger systems
        may need more memory, but making large memory requests can make jobs
        take much longer to come off the queue (since there may only be a few
        nodes with enough memory to meet the request).

    -d, --dry-run
        Print out the command-line that would be used to generate fragments,
        but don't actually run it.
```

4.9 Step 8: Validate designs

```
Validate the designs by running unrestrained flexible backbone simulations.
Only regions of the backbone specified by the loop file are allowed to move.
The resfile used in the previous steps of the pipeline is not respected here;
all residues within 10A of the loop are allowed to pack.

Usage:
    pull_into_place 08_validate_designs <workspace> <round> [options]

Options:
    --nstruct NUM, -n NUM   [default: 500]
        The number of simulations to run per design.

    --max-runtime TIME      [default: 24:00:00]
        The runtime limit for each validation job.

    --max-memory MEM        [default: 1G]
        The memory limit for each validation job.

    --test-run
        Run on the short queue with a limited number of iterations.  This
        option automatically clears old results.
```

(continues on next page)

(continued from previous page)

```
    --clear
        Clear existing results before submitting new jobs.
```

4.10 Step 9: Compare best designs

```
Create a nicely organized excel spreadsheet comparing all of the validated
designs in the given workspace where the lowest scoring decoy within some
threshold of the target structure.

Usage:
    pull_into_place 09_compare_best_designs <workspace> [<round>] [options]

Options:
    -t, --threshold RESTRAINT_DIST   [default: 1.2]
        Only consider designs where the lowest scoring decoy has a restraint
        satisfaction distance less than the given threshold.

    -u, --structure-threshold LOOP_RMSD
        Limit how different two loops can be before they are placed in
        different clusters by the structural clustering algorithm.

    -q, --num-sequence-clusters NUM_CLUSTERS   [default: 0]
        Specify how many sequence clusters should be created.  If 0, the
        algorithms will try to detect the number of clusters that best matches
        the data on its own.

    -s, --subs-matrix NAME   [default: blosum80]
        Specify a substitution matrix to use for the sequence clustering
        metric.  Any name that is understood by biopython may be used.  This
        includes a lot of the BLOSUM and PAM matrices.

    -p, --prefix PREFIX
        Specify a prefix to append to all the files generated by this script.
        This is useful for discriminating files generated by different runs.

    -v, --verbose
        Output sanity checks and debugging information for each calculation.
```

4.11 Cache models

```
Cache various distance and score metrics for each model in the given directory.
After being cached, a handful of these metrics are printed to the terminal to
show that things are working as expected.

Usage:
    pull_into_place cache_models <directory> [options]

Options:
    -r PATH, --restraints PATH
        Specify a restraints file that can be used to calculate the "restraint
        distance" metric.  If the directory specified above was created by the
```

(continues on next page)

(continued from previous page)

```
        01_setup_pipeline script, this flag is optional and will default to the
        restraints used in that pipeline.

    -f, --recalc
        Force the cache to be regenerated.
```

4.12 Count models

```
Count the number of models meeting the given query.

Usage:
    pull_into_place count_models <directories>... [options]

Options:
    --query QUERY, -q QUERY
        Specify which models to include in the count.

    --recalc, -f
        Recalculate all the metrics that will be used to choose designs.

    --restraints PATH
        The path to a set of restraints that can be used to recalculate the
        restraint_distance metric.  This is only necessary if the cache is
        being regenerated in a directory that is not a workspace.

Queries:
    The query string uses the same syntax as the query() method of pandas
    DataFrame objects, which is pretty similar to python syntax.  Loosely
    speaking, each query must consist of a criterion name, a comparison
    operator, and a comparison value.  Only 5 criterion names are recognized:

    "restraint_dist"
        The average distance between all the restrained atoms and their target
        positions in a model.
    "loop_dist"
        The backbone RMSD of a model relative to the input structure.
    "buried_unsat_score"
        The change in the number of buried unsatisfied H-bonds in a model
        relative to the input structure.
    "dunbrack_score"
        The average Dunbrack score of any sidechains in a model that were
        restrained during the loopmodel simulation.
    "total_score"
        The total score of a model.

    Some example query strings:

    'restraint_dist < 0.6'
    'buried_unsat_score <= 4'
```

4.13 Fetch and cache models

```
Download models from a remote host then cache a number of distance and score
metrics for each one.  This script is meant to be called periodically during
long running jobs, to reduce the amount of time you have to spend waiting to
build the cache at the end.

Usage:
    pull_into_place fetch_and_cache_models <directory> [options]

Options:
    --remote URL, -r URL
        Specify the URL to fetch data from.  You can put this value in a file
        called "rsync_url" in the local workspace if you don't want to specify
        it on the command-line every time.

    --include-logs, -i
        Fetch log files (i.e. stdout and stderr) in addition to everything
        else.  Note that these files are often quite large, so this may take
        significantly longer.

    --keep-going, -k
        Keep attempting to fetch and cache new models until you press Ctrl-C.
        You can run this command with this flag at the start of a long job, and
        it will incrementally cache new models as they are produced.

    --wait-time MINUTES, -w MINUTES     [default: 5]
        The amount of time to wait in between attempts to fetch and cache new
        models, if the --keep-going flag was given.
```

4.14 Fetch data

```
Copy design files from a remote source.  A common application is to copy
simulation results from the cluster to a workstation for analysis.  The given
directory must be contained within a workspace created by 01_setup_workspace.

Usage:
    pull_into_place fetch_data <directory> [options]

Options:
    --remote URL, -r URL
        Specify the URL to fetch data from.  You can put this value in a file
        called "rsync_url" in the local workspace if you don't want to specify
        it on the command-line every time.

    --include-logs, -i
        Fetch log files (i.e. stdout and stderr) in addition to everything
        else.  Note that these files are often quite large, so this may take
        significantly longer.

    --dry-run, -d
        Output the rsync command that would be used to fetch data.
```

4.15 Make web logo

```
Create a web logos for sequences generated by the design pipeline.

Usage:
    pull_into_place make_web_logo <workspace> <round> <pdf_output>

It would be nice to pass all unparsed options through to weblogo.  I'll have to
think a bit about how to do that.
```

4.16 Plot funnels

```
Visualize the results from the loop modeling simulations in PIP and identify
promising designs.

Usage:
    pull_into_place plot_funnels <pdb_directories>... [options]

Options:
    -F, --no-fork
        Do not fork into a background process.

    -f, --force
        Force the cache to be regenerated.

    -q, --quiet
        Build the cache, but don't launch the GUI.

This command launches a GUI designed to visualize the results for the loop
modeling simulations in PIP and to help you identify promising designs.  To
this end, the following features are supported:

1. Extract quality metrics from forward-folded models and plot them against
   each other in any combination.

2. Easily visualize specific models by right-clicking on plotted points.
   Add your own visualizations by writing `*.sho' scripts.

3. Plot multiple designs at once, for comparison purposes.

4. Keep notes on each design, and search your notes to find the designs you
   want to visualize.

Generally, the only arguments you need are the names of one or more directories
containing the PDB files you want to look at.  For example:

    $ ls -R
    .:
    design_1  design_2 ...

    ./design_1:
    model_1.pdb  model_2.pdb ...

    ./design_2:
```

(continues on next page)

(continued from previous page)

```
    model_1.pdb  model_2.pdb ...

    $ pull_into_place plot_funnels design_*

This last command will launch the GUI.  If you specified more than one design
on the command line, the GUI will have a panel on the left listing all the
designs being compared.  You can control what is plotted by selecting one or
more designs from this list.  The search bar at the top of this panel can be
used to filter the list for designs that have the search term in their
descriptions.  The buttons at the bottom can be used to save information about
whatever designs are selected.  The "Save selected paths" button will save a
text file listing the path to the lowest scoring model for each selected
design.  The "Save selected funnels" button will save a PDF with the plot for
each selected design on a separate page.

The upper right area of the GUI will contain a plot with different metrics on
the two axes where each point represents a single model.  You can right-click
on any point to take an action on the model represented by that point.  Usually
this means visualizing the model in an external program, like pymol or chimera.
You can also run custom code by writing a script with the extension *.sho that
takes the path of a model as its only argument.  ``plot_funnels`` will search
for scripts with this extension in every directory starting with the directory
containing the model in question and going down all the way to the root of the
file system.  Any scripts that are found are added to the menu you get by
right-clicking on a point, using simple rules (the first letter is capitalized
and underscores are converted to spaces) to convert the file name into a menu
item name.

The tool bar below the plot can be used to pan around, zoom in or out, save an
image of the plot, or change the axes.  If the mouse is over the plot, its
coordinates will be shown just to the right of these controls.  Below the plot
is a text form which can be used to enter a description of the design.  These
descriptions can be searched.  I like using the '+', '++', ... convention to
rank designs so I can easily search for increasingly good designs.

Hotkeys:
    j,f,down: Select the next design, if there is one.
    k,d,up: Select the previous design, if there is one.
    i,a: Focus on the description form.
    z: Use the mouse to zoom on a rectangle.
    x: Use the mouse to pan (left-click) or zoom (right-click).
    c: Return to the original plot view.
    slash: Focus on the search bar.
    tab: Change the y-axis metric.
    space: Change the x-axis metric.
    escape: Unfocus the search and description forms.
```

4.17 Push data

```
Copy design files to a remote destination.  A common application is to copy
input files onto the cluster before starting big jobs.

Usage:
    pull_into_place push_data <directory> [options]
```

(continues on next page)

(continued from previous page)

```

Options:
    --remote URL, -r URL
        Specify the URL to push data to.

    --dry-run, -d
        Output the rsync command that would be used to push data.
```

CHAPTER 5

API documentation

5.1 `pipeline` — organize input and output files

This module defines the Workspace classes that are central to every script. The role of these classes is to provide paths to all the data files used in any part of the pipeline and to hide the organization of the directories containing those files. The base Workspace class deals with files in the root directory of a design. It's subclasses deal with file in the different subdirectories of the design, each of which is related to a cluster job.

class `pull_into_place.pipeline.`**`BigJobWorkspace`**(*root*)

Provide paths needed to run big jobs on the cluster.

This is a base class for all the workspaces meant to store results from long simulations (which is presently all of them except for the root). This class provides paths to input directories, output directories, parameters files, and several other things like that.

class `pull_into_place.pipeline.`**`FixbbDesigns`**(*root*, *round*)

exception `pull_into_place.pipeline.`**`PathNotFound`**(*path*, **directories*)

exception `pull_into_place.pipeline.`**`PipelineError`**(*message*)

class `pull_into_place.pipeline.`**`RestrainedModels`**(*root*)

exception `pull_into_place.pipeline.`**`RosettaNotFound`**(*workspace*)

exception `pull_into_place.pipeline.`**`UnspecifiedRemoteHost`**

class `pull_into_place.pipeline.`**`ValidatedDesigns`**(*root*, *round*)

class `pull_into_place.pipeline.`**`WithFragmentLibs`**

Provide paths needed to interact with fragment libraries.

This is a mixin class that provides a handful of paths and features useful for working with fragment libraries.

class `pull_into_place.pipeline.`**`Workspace`**(*root*)

Provide paths to every file used in the design pipeline.

Each workspace object is responsible for returning paths to files that are relevant to a particular stage of the design pipeline. These files are organized hierarchically: files that are relevant to many parts of the pipeline are stored in the root design directory while files that are relevant to specific stages are stored in subdirectories. You can think of each workspace class as representing a different directory.

The Workspace class itself represents the root directory, but it is also the superclass from which all of the other workspace derive. The reason for this is that the root workspace knows where all the shared parameter files are located, and this information is needed in every workspace.

When modifying or inheriting from this class, keep in mind two things. First, workspace objects should do little more than return paths to files. There are a few convenience functions that clear directories and things like that, but these are the exception rather than the rule. Second, use the @property decorator very liberally to keep the code that uses this API succinct and easy to read.

`cd`(**subpaths*)

Change the current working directory and update all the paths in the workspace. This is useful for commands that have to be run from a certain directory.

`find_path`(*basename*)

Look in a few places for a file with the given name. If a custom version of the file is found in the directory being managed by this workspace, return it. Otherwise return the path to the default version of the file in the root directory.

This function makes it easy to provide custom parameters to any stage to the design pipeline. Just place the file with the custom parameters in the directory associated with that stage.

`focus_dir`

The particular directory managed by this class. This is meant to be overridden in subclasses.

exception `pull_into_place.pipeline.`**`WorkspaceNotFound`**(*root*)

`pull_into_place.pipeline.`**`load_loops`**(*directory*, *loops_path=None*)

Return a list of tuples indicating the start and end points of the loops that were sampled in the given directory.

`pull_into_place.pipeline.`**`load_resfile`**(*directory*, *resfile_path=None*)

Return a list of tuples indicating the start and end points of the loops that were sampled in the given directory.

`pull_into_place.pipeline.`**`workspace_from_dir`**(*directory*, *recurse=True*)

Construct a workspace object from a directory name. If recurse=True, this function will search down the directory tree and return the first workspace it finds. If recurse=False, an exception will be raised if the given directory is not a workspace. Workspace identification requires a file called 'workspace.pkl' to be present in each workspace directory, which can unfortunately be a little fragile.

5.2 `structures` — load and cache PDB files

This module provides a function that will read a directory of PDB files and return a pandas data frame containing a number of score, distance, and sequence metrics for each structure. This information is also cached, because it takes a while to calculate up front. Note that the cache files are pickles and seem to depend very closely on the version of pandas used to generate them. For example, caches generated with pandas 0.15 can't be read by pandas 0.14.

class `pull_into_place.structures.`**`Design`**(*directory*)

Represent a single validated design. Each design is associated with 500 scores, 500 restraint distances, and a "representative" (i.e. lowest scoring) model. The representative has its own score and restraint distance, plus a path to a PDB structure.

exception `pull_into_place.structures.`**`IOError`**

`pull_into_place.structures.`**`load`**(*pdb_dir*, *use_cache=True*, *job_report=None*, *require_io_dir=True*)

Return a variety of score and distance metrics for the structures found in the given directory. As much information as possible will be cached. Note that new information will only be calculated for file names that haven't been seen before. If a file changes or is deleted, the cache will not be updated to reflect this and you may be presented with stale data.

`pull_into_place.structures.`**`read_and_calculate`**(*workspace*, *pdb_paths*)

Calculate a variety of score and distance metrics for the given structures.

`pull_into_place.structures.`**`xyz_to_array`**(*xyz*)

Convert a list of strings representing a 3D coordinate to floats and return the coordinate as a `numpy` array.

5.3 `big_jobs` — submit jobs to the cluster

`pull_into_place.big_jobs.`**`initiate`**()

Return some relevant information about the currently running job.

`pull_into_place.big_jobs.`**`submit`**(*script*, *workspace*, ***params*)

Submit a job with the given parameters.

CHAPTER 6

Contributing

If you find a bug or would like to suggest a new feature, open an issue or make a pull request. If you want to contribute code, be sure your code conforms with PEP8.

Python Module Index

Index