Purpose

This document walks through the steps that created the internal_autodockvina_contestant package using the CELPPade template. D3R runs this package each week to simulate a participant using an AutoDock Vina-based pose prediction workflow. It is intended for:

- Users interested in running this package on their own machine
- CELPP participants who want to understand the CELPPade template
- CELPP participants who want to begin workflow development with an already-functional package

To run the finished product immediately

Try our fast_setup script. It currently only works for 64 bit Linux, with the "git", UCSF "chimera", and openbabel "babel" programs already in your PATH. To ensure a standard build, the script will download local copies of Autodock Vina, MGLTools, RDKit, and the D3R software package.

fast_setup.sh

```
# Make a clean anaconda build so we don't mess with the user's python

wget https://repo.continuum.io/miniconda/Miniconda2-latest-Linux-x86_64.sh
bash Miniconda2-latest-Linux-x86_64.sh -b -p miniconda2
source miniconda2/bin/activate
echo "source `readlink -e miniconda2/bin/activate`" > adv_celppade_env.sh


# Install latest RDKit.
# RDKit versions before 2016.03.01 can make rare ligand prep mistakes.
conda install -y -c rdkit rdkit-postgresql


# Install core D3R utilities for challengedata handling
pip install d3r

#Install Autodock Vina
wget http://vina.scripps.edu/download/autodock_vina_1_1_2_linux_x86.tgz
tar -xvzf autodock_vina_1_1_2_linux_x86.tgz
echo "export PATH=`readlink -e autodock_vina_1_1_2_linux_x86/bin/`:\$PATH" >> adv_celppade_env.sh


# Install MGLTools
wget http://mgltools.scripps.edu/downloads/downloads/tars/releases/REL1.5.6/mgltools_x86_64Linux2_1.5.6.tar.gz
tar -xvzf mgltools_x86_64Linux2_1.5.6.tar.gz
cd mgltools_x86_64Linux2_1.5.6
./install.sh
echo "export MGL_ROOT=`readlink -e .`" >> ../adv_celppade_env.sh
cd ../


# Get the internal_autodockvina_contestant code
git clone https://github.com/drugdata/internal_autodockvina_contestant.git
## You don't need to install the internal_autodockvina_contestant package

# Load up the environment with the installed packages
source adv_celppade_env.sh

# Test the build
cd internal_autodockvina_contestant/test_data
source test.sh
```

The script then runs on the test data. If successful, users should find 4 pairs of pdb and mol files in the 5-pack_docking_results/celpp_weekXX_XXXX_dockedresults_XXXXX/1fcz folder. The environment required to run this build can be recovered in a new terminal by running “source adv_celppade_env.sh” in the top-level directory.

If a successful build is supplied with a registered CELPP user’s Box.com credentials (in the form of an ftp_config file), then it is capable of running real challenge weeks and uploading the results for scoring.

Introduction

This document is organized so that users can follow the creation of a CELPP participant package using the CELPPade template. Also, users interested in beginning with a functional (though very naive!) workflow and making small tweaks can read this tutorial to understand how the internal_autodockvina_contestant package works.

CELPPade is designed to help you quickly jump into the CELPP challenge. It provides three template files to automate receptor preparation, ligand preparation, and docking. CELPPade also links in core D3R code to download weekly challengedata packages and upload your predictions.

CELPPade is not a requirement for participation in CELPP. Your CELPPade-derived workflow, though pre-formatted for GitHub, is not uploaded to GitHub by default. You have the option to keep your code completely local/private if you so choose.

Participants benefit from uploading their CELPPade-derived workflow to GitHub in two ways:

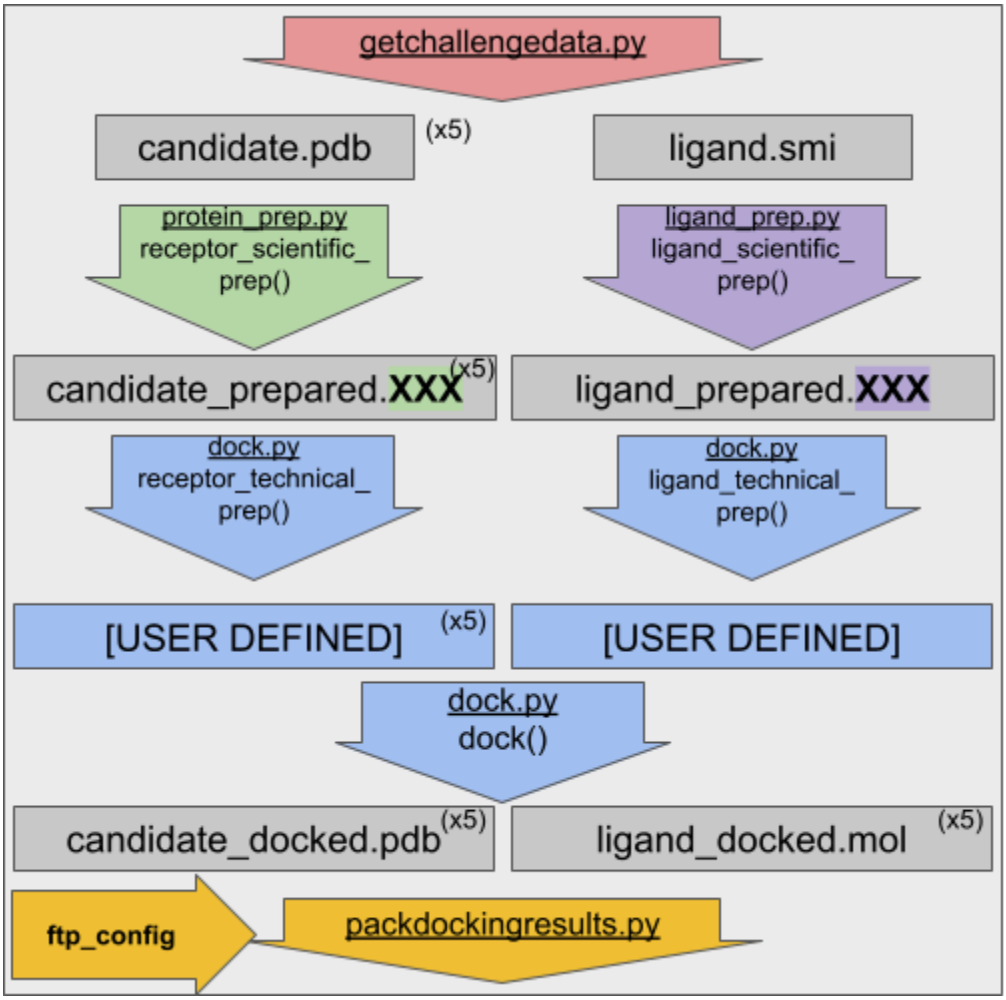
1) GitHub is a state-of-the-art code management platform. It offers easy versioning, approachable branching and forking, public bug reporting, and a clean interface to integration testing services. These features significantly lower the barrier to creating quality code and reproducible results.
2) Publicly sharing your workflow code on GitHub is an easy way to contribute to the scientific community. Whether you’re a professional in molecular docking or just visiting from another discipline, participation in CELPP is a great way to help improve computer-aided drug design. Public code enables other scientists to credit your contributions and carry forward your work.

“Scientific” vs. “Technical” molecule preparation

CELPP is about more than just docking. Starting from a receptor pdb file and ligand SMILES, many decisions about molecule preparation have to be made before docking can be run. These decisions include, for example, “how to generate 3D conformations of the ligand” and “whether to keep solvent molecules in the binding site”. Each of these decisions is an open scientific question. However, a hypothetical CELPP participant from a lab that focuses on ligand conformer generation may not have a strong background in binding site solvent retention (or any number of other molecule preparation decisions). For that reason, we have attempted to discretize the steps in CELPP to make it easier to jump in and make improvements to an existing workflow.

This first version of CELPPade represents our initial attempt at a flexible-yet-accessible workflow template. It splits molecule preparation into “scientific” and “technical” phases. “Scientific” preparation is loosely defined as “changing the information inherent in the structure”. Examples of scientific molecule preparation include decisions to keep or remove solvent, atomic charge assignment, protonation, atomic movements, and resolution of dual-occupancy residues.

“Scientific” preparation is contrasted with the subsequent “Technical” preparation stage, which should handle processing of a generic file type in preparation to run a specific docking program. Examples of technical preparation steps include conversion of structures to proprietary file formats and grid generation for specific docking programs. The purpose of separating these stages into different files is so that specialized researchers can easily insert their proposed improvements into a common workflow and compare its performance.



Here's an overview of the entire workflow. Downward-pointing arrows represent Python functions, and boxes represent data files. Users implement their workflow in 5 functions split across 3 scripts - protein_prep.py (green), ligand_prep.py (purple), and dock.py (blue) files. A participant's upload credentials (Box.com login information and SciCrunch ID) are saved in the yellow ftp_config file.

My simple AutoDock Vina workflow

I recommend that you start off by documenting the commands that will run your workflow on the command line for a single PDB file (protein.pdb) and ligand SMILES (ligand.smi). I'm going to write the commands for my receptor prep, ligand prep, and docking separately.

Requirements: Chimera, openbabel, RDKit (python package), and MGLTools

Ligand scientific preparation

```
# First, generate a single guess at a 3d structure  of the ligand
$ python rdkit_smiles_to_3d_sdf.py ligand.smi ligand.sdf

# Convert this structure to mol2 format
$ babel -isdf ligand.sdf -omol2 ligand.mol2

# Use Chimera's DockPrep to calculate charges
$ chimera --nogui --script "chimeraPrep.py ligand.mol2 prepared_ligand.mol2"

# And use an AutoDockTool to convert the mol2 to pdbqt format
$ $MGL_ROOT/bin/pythonsh $MGL_ROOT/MGLToolsPckgs/AutoDockTools/Utilities24/prepare_ligand4.py -l
prepared_ligand.mol2
```

Protein scientific preparation

```
# Remove all non-receptor information from the PDB file
$ grep ATOM protein.pdb > stripped_protein.pdb

# Generate charges and other small tweaks for receptor preparation
$ chimera --nogui --script "chimeraPrep.py stripped_protein.pdb prepared_protein.mol2"

# And use an AutoDockTool to convert the mol2 to pdbqt format
$ $MGL_ROOT/bin/pythonsh $MGL_ROOT/MGLToolsPckgs/AutoDockTools/Utilities24/prepare_receptor4.py -r
prepared_protein.mol2
```

Docking

```
## Perform vina docking
vina --receptor prepared_protein.pdbqt --ligand prepared_ligand.pdbqt --center_x 30.662 --center_y
-2.264 --center_z 34.558 --size_x 10 --size_y 10 --size_z 10 --seed 999

## Extract only the top pose from the 10 output structures
sed -e '/ENDMDL/,$d' prepared_ligand_out.pdbqt > top_pose.pdbqt
echo ENDMDL >> top_pose.pdbqt

## Convert the top ligand pose to mol
$MGL_ROOT/bin/pythonsh $MGL_ROOT/MGLToolsPckgs/AutoDockTools/Utilities24/pdbqt_to_pdb.py -f
top_pose.pdbqt -o top_pose.pdb
babel -ipdb top_pose.pdb -omol top_pose.mol
```

```
## Convert the protein to pdb (technically we didn't allow for protein flexibility here so we could
## just hand back stripped_protein.pdb, but I'll write this for the more general case)
$MGL_ROOT/bin/pythonsh $MGL_ROOT/MGLToolsPckgs/AutoDockTools/Utilities24/pdbqt_to_pdb.py -f
prepared_protein.pdbqt -o docked_protein.pdb
```

Helper scripts

I've referenced two python scripts in the process. Here they are:

rdkit_smiles_to_3d_sdf.py

```
## Generates a naive 3D sdf file from a
## ligand SMILES string.

import rdkit.Chem
import rdkit.Chem.AllChem
import sys

smiles = open(sys.argv[1]).read().strip()
mol = rdkit.Chem.MolFromSmiles(smiles)
molH = rdkit.Chem.AddHs(mol)
rdkit.Chem.AllChem.EmbedMolecule(molH)
rdkit.Chem.AllChem.UFFOptimizeMolecule(molH)

w = rdkit.Chem.SDWriter(sys.argv[2])
w.write(molH)
w.close()
```

chimeraPrep.py

```
## A commandline script to call
## Chimera's DockPrep function using all
## default settings.

import chimera
import sys
opened =
chimera.openModels.open(sys.argv[1])
mol = opened[0]

import DockPrep

DockPrep.prep([mol])
from WriteMol2 import writeMol2
with open(sys.argv[2],'wb') as of:
    writeMol2([mol], of)
```

Getting the template code and implementing the workflow

CELPPade is distributed using a system called "cookiecutter". Do not download CELPPade using the "git clone" command - use the cookiecutter command below instead.

First, install cookiecutter itself:
$ pip install cookiecutter

Then make your cookiecutter version of CELPPade
$ cookiecutter https://github.com/drugdata/cookiecutter-pycustomdock.git

Cookiecutter will prompt you to personalize your code by providing some information. The first several prompts (class and program/package name) are important to distinguish your code on

GitHub. For the questions after “package_description”, we recommend that you accept the default values (hit enter) unless you are an advanced user.

You will now have a new folder in the current directory. What’s inside? Here is how my implementation of our internal AutoDock Vina contestant started out:

```
internal_autodockvina_contestant/
├── ftp_config
├── full_week_run
│   └── full_week_run.sh
├── internal_autodockvina_contestant
│   ├── __init__.py
│   ├── internal_autodockvina_contestant_protein_prep.py
│   ├── internal_autodockvina_contestant_ligand_prep.py
│   └── internal_autodockvina_contestant_dock.py
├── README.rst
├── setup.py
├── test_data
│   ├── celpp_weekXX_XXXX.tar.gz
│   ├── mini_pdb
│   │   └── fc
│   │       └── pdb1fcz.ent
│   └── test.sh
└── tox.ini
```

Underlined files are intended to be modified by participants

Now I’ll discuss how to implement our commandline workflow into each of these template files.

ftp_config

Contains your identifying participant information. This file is read by the challenge data downloader (getchallengedata.py) and result uploader (packdockingresults.py) to access to your personal submission directory. Only registered participants can successfully use these scripts as they require Box.com and CELPP registration. However, unregistered participants can still manually access the weekly challengedata folder via web browser.

When you are ready to start downloading challenge packages and uploading your predictions, you should substitute the default values in this file with your Box.com username/password and 5-digit D3R contestant ID. If you requested multiple submission directories, the suffix for the folder you intend to submit to should be included in this path. Unless you are an advanced user, do not modify the “host” line.

If you later add your package to GitHub, make sure not to add this file, as it will contain your Box.com password in plain text!

Example ftp_config file

```
host dav.box.com
user <your box.com email address here>
pass <your box.com password here>
contestantid <your D3R-assigned 5-digit ID, eg. 12345>
```

If you requested multiple submission directories, you should have a separate ftp_config for each prediction workflow you will run. Each submission directory after the first will have an underscore and suffix (eg “12345_2”), and you should add this to the contestantid field in that workflow’s ftp_config.

internal_autodockvina_contestant_protein_prep.py

Contains the receptor_scientific_prep() function. If you do not overwrite this function, its default behavior is to pass the unchanged candidate pdb file to the subsequent technical prep functions. I’ll walk through how I put the original protein prep commands (orange) into the Python workflow.

internal_autodockvina_contestant_protein_prep.py (excluding __main__())
Grey text is unchanged from the template file
Orange text is verbatim from the commandline sequences above
Green text is newly written and discussed below

```
#!/usr/bin/env python


__author__ = 'j5wagner@ucsd.edu'

from d3r.celppade.custom_protein_prep import ProteinPrep
import os

chimera_prep_text = '''
import chimera
import sys
opened = chimera.openModels.open(sys.argv[1])
mol = opened[0]
import DockPrep
DockPrep.prep([mol])
from WriteMol2 import writeMol2
with open(sys.argv[2],'wb') as of:
```

#1#

```
    writeMol2([mol], of)
'''
class chimera_protprep(ProteinPrep):
    """Abstract class defining methods for a custom docking solution
    for CELPP
    """
    ProteinPrep.OUTPUT_PROTEIN_SUFFIX = '.pdbqt'                                    #2#
    def receptor_scientific_prep(self,
                                 protein_file,
                                 prepared_protein_file,
                                 targ_info_dict={}):
        """
        Protein 'scientific preparation' is the process of generating
        a dockable representation of the candidate protein from a
        single-chain PDB file.
        :param protein_file: PDB file containing candidate protein.
        :param prepared_protein_file: The result of preparation should have this file
name.
        :param targ_info_dict: A dictionary of information about this target and the
candidates chosen for docking.
        :returns: True if preparation was successful. False otherwise.
        """
        with open('chimeraPrep.py', 'wb') as of:                                    #3#
            of.write(chimera_prep_text)
        os.system('grep ATOM ' + protein_file + ' > stripped_protein.pdb')          #4#
        os.system('chimera --nogui --script "chimeraPrep.py stripped_protein.pdb    #5#
prepared_protein.mol2" 1> chimeraPrep.stdout 2> chimeraPrep.stderr')
        os.system('. /usr/local/mgltools/bin/mglenv.sh; $MGL_ROOT/bin/pythonsh
$MGL_ROOT/MGLToolsPckgs/AutoDockTools/Utilities24/prepare_receptor4.py -r
prepared_protein.mol2 1> prepare_receptor4.stdout 2> prepare_receptor4.stderr')  #6#
        os.system('cp prepared_protein.pdbqt ' + prepared_protein_file)
        return True
                                                                                    #7#
                                                                                    #8#
```

1. Because I wanted my short chimeraPrep.py script in the working directory, I've put all of its code into a string at the top of the file. Even though the we're writing a Python function, the chimeraPrep.py code has to be run in Chimera's Python interpreter, so I can't just run these commands inside of the scientific_prep function. Later, I'll write this short script to a file, and use a commandline call to run chimeraPrep.py through the Chimera interpreter.
2. CELPPade requires participants to define an output file format from their scientific preparation. For Vina, it's useful to produce a "pdbqt" file. Pre-defining the file type is necessary and allows the workflow to ensure that the preparation was successful. Based on this output suffix, the receptor_scientific_prep() function will be given a filename (input argument "prepared_protein_file") that it must produce in order for preparation to be

considered successful .This design choice improves the modularity of CELPPade-derived code, as another user who uses this scientific prep can safely expect a certain type of output.

3. This code writes the chimera_prep_text string to the file “chimeraPrep.py” in the current directory. Each receptor is prepared in a new directory, so the script has to be rewritten each time.
4. The input protein filename is defined by the “protein_file” argument to this function. I’ve put all of my shell commands into os.system() calls. This mimics human commandline input. I’m using os.system() because of its simplicity, but experienced Python users are better off using the shutil or subprocess modules.
5. This line runs the chimeraPrep.py script through the Chimera script interpreter. It is very useful to pipe stdout and stderr to files for debugging workflows.
6. This line runs the mol2-to-pdbqt conversion. My default environment does not include certain environmental variables such as $MGL_ROOT, so the first part of this command sources an environment script to set them.
7. This line takes the “prepared_protein.pdbqt” and copies it to have the correct filename. If this file does not exist when the function is done running, CELPPade will consider the scientific protein preparation to have failed.
8. The return value from this function can be used to self-report failure by returning False. This may be useful if the participant does not want this receptor to move forward to docking. In my case, I want every possible receptor to move forward, so this function returns True.

internal_autodockvina_contestant_ligand_prep.py

Contains the ligand_scientific_prep() function. This is where the workflow initially converts ligand SMILES to 3D representations. If you do not write this function, the subsequent technical step functions will simply be passed the unchanged ligand SMILES file.

internal_autodockvina_contestant_ligand_prep.py (excluding __main__())
Grey text is unchanged from the template file
Orange text is verbatim from the commandline sequences above
Green text is newly written and discussed below

```
#!/usr/bin/env python

__author__ = 'j5wagner@ucsd.edu'

from d3r.celppade.custom_ligand_prep import LigandPrep
import os
```

#1#

```
rdkit_smiles_to_3d_sdf_text = '''
import rdkit.Chem
import rdkit.Chem.AllChem
import sys
if not(len(sys.argv)) == 3:
    print "python smiles2Mol.py inputSmiles outputSdf"
    sys.exit()
smiles = open(sys.argv[1]).read().strip()
mol = rdkit.Chem.MolFromSmiles(smiles)
molH = rdkit.Chem.AddHs(mol)
rdkit.Chem.AllChem.EmbedMolecule(molH)
rdkit.Chem.AllChem.UFFOptimizeMolecule(molH)
w = rdkit.Chem.SDWriter(sys.argv[2])
w.write(molH)
w.close()
'''

chimera_prep_text = '''
import chimera
import sys
opened = chimera.openModels.open(sys.argv[1])
mol = opened[0]
import DockPrep
DockPrep.prep([mol])
from WriteMol2 import writeMol2
with open(sys.argv[2],'wb') as of:
    writeMol2([mol], of)
'''

class chimera_ligprep(LigandPrep):
    """Abstract class defining methods for a custom ligand docking solution
    for CELPP
    """                                                                    #2#
    LigandPrep.OUTPUT_LIG_SUFFIX = '.pdbqt'
    def ligand_scientific_prep(self,
                               lig_smi_file,
                               out_lig_file,
                               targ_info_dict={}):
        """
        Ligand 'scientific preparation' is the process of generating a
        dockable representation of the target ligand from its SMILES
        string.
        :param lig_smi_file: File containing SMILES for target ligand.
        :param out_lig_file: The result of preparation should have this file name.
        :param targ_info_dict: A dictionary of information about this target and the
candidates chosen for docking.
        :returns: True if preparation was successful. False otherwise.
        """
                                                                           #3#
```

```
        with open('rdkit_smiles_to_3d_sdf.py','wb') as of:
            of.write(rdkit_smiles_to_3d_sdf_text)
         with open('chimeraPrep.py','wb') as of:
            of.write(chimera_prep_text)                                          #4#

         os.system('python rdkit_smiles_to_3d_sdf.py ' + lig_smi_file +
                   ' ligand.sdf 1> rdkit_smiles_to_3d_sdf.stdout 2>' +
                   ' rdkit_smiles_to_3d_sdf.stderr')
         os.system('babel -isdf ligand.sdf -omol2 ligand.mol2'                   #5#
                   + ' 1> lig_sdf_to_mol2.stdout 2> lig_sdf_to_mol2.stderr')
         os.system('chimera --nogui --script "chimeraPrep.py ' +
                   'ligand.mol2 charged_ligand.mol2"' +                          #6#
                   ' 1> chimeraLigPrep.stdout 2> chimeraLigPrep.stdout')
         os.system('. /usr/local/mgltools/bin/mglenv.sh; pythonsh
$MGL_ROOT/MGLToolsPckgs/AutoDockTools/Utilities24/prepare_ligand4.py -l      #7#
charged_ligand.mol2 1> prepare_ligand4.stdout 2> prepare_ligand4.stderr')
         os.system('cp charged_ligand.pdbqt ' + out_lig_file)                    #8#

         return True
```

1. Similar to chimeraPrep.py, here we also need to generate 3D coordinates for a molecule. This is a very basic way to do so with RDKit. In my experience, RDKit has had some interesting behavior, so I insulate it from my ligand_scientific_prep() function by using an os.system() call, and am able to capture its output (point #4#)
2. Vina requires both protein and ligand to be in pdbqt format. It is a coincidence that receptor and ligand prep both produce pdbqt files in this example. There is no requirement that they be the same.
3. First, we write our specialty scripts to local files so we can call them in subsequent os.system() commands.
4. Again, the ligand in the real workflow will not be named “ligand.smi” like in our example. The “ligand_smi_file” argument passed to this function will have the real ligand filename. Note that in this and subsequent os.system() calls, the commands capture stdout and stderr to files. This piping makes debugging much easier.
5. The same as in receptor_scientific_prep() - chimeraPrep.py is an exotic script that calls Chimera’s DockPrep procedure, but requires a live Chimera session to do so.
6. Again, we require non-standard environment variables to be set to use the prepare_ligand4 script from AutoDockTools. The command before the semicolon in this string sources a shell script that sets up environment variables, and the command after inherits its environment. Note that this environment would be “forgotten” if we ran two separate os.system() commands - Any required environment variables must be set in the same os.system() command as the command that requires them.
7. This line simply copies the final prepared ligand file (charged_ligand.pdbqt) to have the appropriate output name, as given by the out_lig_file argument to the function. More experienced users are directed to the shutil Python module.

8. Again, this function gives the option to self-report failure. We don't have logic for failure cases in this simple workflow, so we just always return True. CELPPade will still check that the properly-named file was created and is of nonzero size.

internal_autodockvina_contestant_dock.py

Contains 3 functions - ligand_technical_prep(), receptor_technical_prep(), and dock().

internal_autodockvina_contestant_dock.py (excluding __main__())
Grey text is unchanged from the template file
Orange text is verbatim from the commandline sequences above
Green text is newly written and discussed below

```
#!/usr/bin/env python

__author__ = 'j5wagner@ucsd.edu'


from d3r.celppade.custom_dock import Dock

class autodockvina(Dock):
    """Abstract class defining methods for a custom docking solution
    for CELPP
    """
    Dock.SCI_PREPPED_LIG_SUFFIX = '_prepared.pdbqt'                                  #1#
    Dock.SCI_PREPPED_PROT_SUFFIX = '_prepared.pdbqt'


    def ligand_technical_prep(self, sci_prepped_lig, targ_info_dict = {}):          #2#
        """
        'Technical preparation' is the step immediate preceding
        docking. During this step, you may perform any file
        conversions or processing that are specific to your docking
        program. Implementation of this function is optional.
        :param sci_prepped_lig: Scientifically prepared ligand file
        :param targ_info_dict: A dictionary of information about this target and the
candidates chosen for docking.
        :returns: A list of result files to be copied into the
        subsequent docking folder. The base implementation merely
        returns the input string in a list (ie. [sci_prepped_lig])
        """
        return super(autodockvina,
                     self).ligand_technical_prep(sci_prepped_lig,
                                                 targ_info_dict = targ_info_dict)

    def receptor_technical_prep(self,
                                sci_prepped_receptor,                                #2#
                                pocket_center,
                                targ_info_dict = {}):
        """
        'Technical preparation' is the step immediately preceding
        docking. During this step, you may perform any file
```

```
        conversions or processing that are specific to your docking
        program. Implementation of this function is optional.
        :param sci_prepped_receptor: Scientifically prepared receptor file
        :param pocket_center: list of floats [x,y,z] of predicted pocket center
        :param targ_info_dict: A dictionary of information about this target and the
candidates chosen for docking.
        :returns: A list of result files to be copied into the
        subsequent docking folder. This implementation merely
        returns the input string in a list (ie [sci_prepped_receptor])
        """

        #return [sci_prepped_receptor, pocket_center]
        return super(autodockvina,
                     self).receptor_technical_prep(sci_prepped_receptor,
                                                   pocket_center,
                                                   targ_info_dict=targ_info_dict)




    def dock(self,
             tech_prepped_lig_list,
             tech_prepped_receptor_list,
             output_receptor_pdb,
             output_lig_mol,
             targ_info_dict={}):
        """
        This function is the only one which the contestant MUST
        implement.  The dock() step runs the actual docking
        algorithm. Its first two arguments are the return values from
        the technical preparation functions for the ligand and
        receptor. These arguments are lists of file names (strings),
        which can be assumed to be in the current directory.
        If prepare_ligand() and ligand_technical_prep() are not
        implemented by the contestant, tech_prepped_lig_list will
        contain a single string which names a SMILES file in the
        current directory.
        If receptor_scientific_prep() and receptor_technical_prep() are not
        implemented by the contestant, tech_prepped_receptor_list will
        contain a single string which names a PDB file in the current
        directory.
        The outputs from this step must be two files - a pdb with the
        filename specified in the output_receptor_pdb argument, and a
        mol with the filename specified in the output_ligand_mol
        argument.
        :param tech_prepped_lig_list: The list of file names resturned by
ligand_technical_prep. These have been copied into the current directory.
        :param tech_prepped_receptor_list: The list of file names resturned by
receptor_technical_prep. These have been copied into the current directory.
        :param output_receptor_pdb: The final receptor (after docking) must be
converted to pdb format and have exactly this file name.
        :param output_lig mol: The final ligand (after docking) must be converted to
mol format and have exactly this file name.
        :param targ_info_dict: A dictionary of information about this target and the
candidates chosen for docking.
        :returns: True if docking is successful, False otherwise. Unless overwritten,
this implementation always returns False
        """
        receptor_pdbqt = tech_prepped_receptor_list[0]
        ligand_pdbqt = tech_prepped_lig_list[0]
```

```
        pocket_center = targ_info_dict['pocket_center']                                  #3#

        vina_command = ('vina --receptor ' + receptor_pdbqt + '  --ligand  ' +           #4#
                        ligand_pdbqt + ' --center_x ' + str(pocket_center[0]) +
                        ' --center_y ' + str(pocket_center[1]) +                         #5#
                        ' --center_z ' + str(pocket_center[2]) +
                        ' --size_x 10 --size_y 10 --size_z 10 --seed 999 ' +
                        ' 1> vina.stdout 2> vina.stderr')
        print "Running: " + vina_command
        os.system(vina_command)

        out_dock_file = ligand_pdbqt.replace('.pdbqt','_out.pdbqt')

        os.system("sed -e '/ENDMDL/,$d' " + out_dock_file + " > top_pose.pdbqt")         #6#
        os.system("echo ENDMDL >> top_pose.pdbqt")
                                                                                         #7#
        os.system('. /usr/local/mgltools/bin/mglenv.sh; python
$MGL_ROOT/MGLToolsPckgs/AutoDockTools/Utilities24/pdbqt_to_pdb.py -f top_pose.pdbqt
-o top_pose.pdb')                                                                        #8#
        os.system("babel -ipdb top_pose.pdb -omol " + output_lig_mol)

        os.system('. /usr/local/mgltools/bin/mglenv.sh; python                           #8#
$MGL_ROOT/MGLToolsPckgs/AutoDockTools/Utilities24/pdbqt_to_pdb.py -f ' +
receptor_pdbqt + ' -o ' + output_receptor_pdb)                                           #8#
```

1. The dock.py file needs to know what kind of scientific prep to expect. Scientific prep steps might, in the future, produce many file types, so here we specify which one we're looking for. The matching file for each gets copied into the technical preparation directories.
2. Autodock Vina doesn't require any kind of "exotic" technical ligand or receptor preparation, so this code leaves the original template class in use. (*Aside: it could be argued that the mol2-to-pdbqt conversion in the scientific prep steps was "exotic", and that mol2 is the more "generic" format. By this logic, the mol2-to-pdbqt step could have been performed in the technical_prep() functions in this file, instead of in the scientific_prep() functions earlier*)
3. We expect that technical prep may produce multiple required files. For that reason, technical preparation functions return a python list of output filenames, all of which are copied into the docking directory. These lists of filenames are inputs to the dock() function. Since we don't implement any technical preparation above, the default tech prep function just passes in the scientifically prepared file for receptor and ligand.
4. You may have noticed that all of these functions receive an input called "targ_info_dict". This dictionary contains some useful information about the target structure and the candidates. Here we read the pocket center coordinates from the dictionary. The pocket center is calculated as the center of mass of the Largest Maximal Common Substructure (LMCSS) ligand, and an example of a value is below. Information about the crystallization pH,number of rotatable bonds, and other candidates may be useful for designing more complex docking procedures.

targ_info_dict for 1fcz


```
{
'query': ['1fcz'],

'ph': ['7'],

'pocket_center': [40.252, 15.196, 84.399],

'ligand': ['156'],

'inchi':['InChI=1S/C24H26O3/c1-23(2)13-14-24(3,4)20-15-18(10-11-19(20)23)21(25)12-7-16-5-8-17(9-6-16)22(26)27/h5-12,15H,13-14H2,1-4H3,(H,26,27)/b12-7+'],

'rotatable_bond': ['4'],

'LMCSS': [{'lig_name': ['156'], 'chain': ['A'], 'cand_id': ['1fcz'], 'mcss_size': ['29'], 'resolution': ['1.38'], 'size': ['29']}],

'hiResHolo': [{'cand_id': ['1fcy'], 'lig_name': ['564'], 'resolution': ['1.3'], 'chain': ['A']}],

'SMCSS': [{'lig_name': ['REA'], 'chain': ['A'], 'cand_id': ['2lbd'], 'mcss_size': ['14'], 'resolution': ['2.06'], 'size': ['28']}],

'hiTanimoto': [{'cand_id': ['1fcz'], 'lig_name': ['156'], 'resolution': ['1.38'], 'tanimoto_similarity': ['1.0'], 'chain': ['A']}], 'size': ['29']
}
```


5. The commandline Vina call requires the pocket center coordinates, which will change for each receptor. This line feeds in the appropriate coordinates for each docking candidate.
6. AutoDock Vina has a specific pattern for naming its docking output. Since our input name is different for each receptor, I apply the pattern to each input filename to generate the expected output filename.
7. We have to be a little bit hack-ish here to get the first pose from the docking output.
8. The workflow converts the top ligand pose and receptor to their final required filenames and formats.

Running your workflow

Installation

Your CELPPade package itself does not need to be installed to run. In fact, we recommend you don't install it (to keep your PATH namespace simple in case you copy this workflow later). However, you *do* need to install the "d3r" pypi package to run your CELPPade workflow. You can install it with the command

$ pip install d3r

Running locally on test data

The cookiecutter code comes with a test_data folder. Inside is a shell script, test.sh, which runs receptor prep, ligand prep, and docking on a single benchmark target, 1fcz. It is not necessary for the entire workflow to be implemented before running test.sh -- In fact, it is helpful to run test.sh as each CELPPade stage is being written to ensure correct output.

```
$ cd test_data
$ source test.sh
```

test.sh will simulate running a challenge week, but will only use local data, and will not upload results to your Box.com submission directory. Each script file (protein_prep.py, ligand_prep.py, and dock.py) will run in a separate directory (2-protein_prep, 3-ligand_prep, and 4-docking, respectively) to facilitate debugging. Note that, in the interest of robustness, each of these three stages will always run, independent of whether the previous stage failed for some/all targets.

If the workflow is successful, the 5-pack_docking_results folder should become populated after completion of the test.sh script. When my workflow was complete, it looked like this:

```
5-pack_docking_results
├── celpp_weekXX_XXXX_dockedresults_XXXXX
│   └── 1fcz
│       ├── hiResHolo-1fcz_1fcy_docked.mol
│       ├── hiResHolo-1fcz_1fcy_docked.pdb
│       ├── hiTanimoto-1fcz_1fcz_docked.mol
│       ├── hiTanimoto-1fcz_1fcz_docked.pdb
│       ├── LMCSS-1fcz_1fcz_docked.mol
```

```
│       ├── LMCSS-1fcz_1fcz_docked.pdb
│       ├── SMCSS-1fcz_2lbd_docked.mol
│       └── SMCSS-1fcz_2lbd_docked.pdb
├── celpp_weekXX_XXXX_dockedresults_XXXXX.tar.gz
└── final.log
```

If this had been a real challenge week run, the file `celpp_weekXX_XXXX_dockedresults_XXXXX.tar.gz` would have the X's replaced by the appropriate week, year, and contestant identifier, and would automatically be uploaded to my submission directory.

Running the weekly challenge

If the workflow runs successfully on the test data and the ftp_config file is correctly filled out, then you are ready to run a real challenge week. Real CELPP runs are performed in the full_week_run folder. The full_week_run.sh code is almost identical to the test.sh, except that it checks the CELPP challengedata folder on box to identify this week's tar file, downloads it, and then uploads your final result to your submission directory.

Most weeks have around 50 targets with 4-5 candidates each, meaning a total of 200-250 prep/docking jobs. On our machines, this takes 6+ hours.

When you are ready to run a weekly challenge, enter

```
$ cd full_week_run
$ source full_week_run.sh
```

The full_week_run.sh script will fail if the CELPP competition is not currently active. Recall that the competition runs Sunday at 12:01 AM to Tuesday at 3:00 PM (U.S. West Coast Time).

Outside of this window, users can manually download an old challengedata package and use that for more thorough testing. The CELPP evaluation server only downloads the most recent week's submission, so it's fine if your test run uploads results from a previous week (however these results won't be evaluated).

Setting up automatic weekly runs

You might have better things to do at 12:01 AM on Sunday morning than start a CELPP workflow. Look into schedulers that run on your operating system (eg. cron) to automate weekly runs. If the workflow requires a special Python environment, be sure that the scheduler runs the proper bin/activate script to enable it.

Uploading the package to GitHub

If you're interested in sharing your code on GitHub, instructions for uploading a cookiecutter-derived package can be found here:

https://cookiecutter-pypackage.readthedocs.io/en/latest/tutorial.html