Afpdb - Protein AI Design Use Cases

Example AI Protein Design Use Cases

One effective AI protein design strategy involves generating hypothetical binder backbone structures based on a target protein (e.g., using RFDiffusion), followed by using inverse folding AI models to create binder sequences (e.g., using ProteinMPNN). These designs can then be validated using structure prediction models like AlphaFold. In order to archieve successful designs, thousand of protein structures need to be manipulated within a project. **Afpdb** aims to increase the productivity for such large-scale AI protein design efforts.

In this section, we demonstrate a few real use cases on how **Afpdb** tools help in AI protein design processes.

```
from pathlib import Path
import os
pwd=Path(os.getcwd())
IN_COLAB=str(pwd)=="/content" # we are in Google Colab

if IN_COLAB:
    pwd=Path("/content/afpdb/tutorial")
    # remove local proxy setting
    os.environ["https_proxy"]=""
    os.environ["http_proxy"]=""
    os.environ["ftp_proxy"]=""
    # install afpdb
    if not os.path.isfile("INSTALL_AFPDB"):
        ! git clone git+https://github.com/data2code/afpdb.git && cd afpdb && pip install .
        ! touch INSTALL_AFPDB
    from IPython.display import Javascript
    display(Javascript('''google.colab.output.setIframeHeight(0, true, {maxHeight: 50000})'''))
    from IPython.display import HTML, display
    def set_css():
      display(HTML('''
        <style>
          pre {
              white-space: pre-wrap;
          }
        </style>
      '''))
    get_ipython().events.register('pre_run_cell', set_css)
else: # in a local jupyter notebook
    %reload_ext autoreload
    %autoreload 2
    # we assume afpdb has been preinstall

def install_pymol():
    try:
        import pymol2
    except Exception as e:
        if not IN_COLAB:
            print("Please install PyMOL first!")
        else:
            !pip install -q condacolab
            import condacolab
            condacolab.install()
            ! conda install conda-forge::pymol-open-source
            print("Colab does not have openssl 3.2.0, install it...")
            ! conda install conda-forge::openssl=3.2.0
            import pymol2

from afpdb.afpdb import Protein,util,RS,RL,ATS
import numpy as np
import pandas as pd
import re
# two example PDB files used in this tutorial
fn = pwd / "example_files/5cil.pdb"
fk = pwd / "example_files/fake.pdb"
```

Handle Missing Residues in AlphaFold Prediction

In evaluating protein structure prediction models such as AlphaFold, the true experimental structure `p_exp` may contain missing residues. We often replace missing residues with glycine for AlphaFold modeling. When align the predicted structure `p_af` against `p_exp`, we need to exclude those added G residues, as they do not exist in `p_exp`.

The example below first created a fake experimental `p_exp` with five missing residues, and a fake AF-predicted `p_af` with five extra Glycines. `p_exp` contains 106 residues with L15-19 missing. `p_af` contains 111 residues with five extra Gs.

From `p_exp`, `rsi_missing()` returns a residue selection object that can be used to point at the extra G residues in `p_af`. Methods `align` and `rmsd` can then be done with those extra residues excluded.`

```
p=Protein(fn)
miss_residues="L15-19"

# create an experimental structure for chain L with residues L15-19 missing
p_exp=p.extract( ~ p.rs(miss_residues) & "L")
print(p_exp.seq(), "\n")
print("Notice five residues XXXXX corresponds to position 15-19\n")

af_seq=p_exp.seq(gap="G")
print(af_seq, "\n")
print("We run AlphaFold prediction using af_seq, i.e., replacing missing residues with Gly ...\n")

# Create a fake AlphaFold predicted structure, by changing the missing residues to Glycine
p_af=p.extract("L") # extract chain L
rs=p_af.rs(miss_residues)
print(rs)
p_af.data.aatype[rs.data]=RS.i("G") # set residue type to Gly
# np.ix_ select all rows in rs.data and all columns in ~ATS("N,CA,C,O")).data
p_af.data.atom_mask[np.ix_(rs.data, (~ATS("N,CA,C,O")).data)]=0 # Gly does not have side chain atoms
print("Pretend p_af is the output of AlphaFold prediction\n")
print("Verify the missing resiudes are not missing in p_af, they are Gs:\n")
print(p_af.seq(), "\n")

print(f"# of residues: p_exp={len(p_exp)}, p_af={len(p_af)}")
print("AlphaFold predicted structure has 5 more residues, which needs to be excluded for alignment purpose.\n")
# we now need to align the common residues, excluding the inserted G residues as they do not exist in p_exp
# the above is equivalent to first find the missing residue indices, notice rsi_missing returns an array
# rsi=p_exp.rsi_missng()
# warning: do not use rsi on p_exp, as rsi contains non-existing residue indices!!!
# cast rsi into a RS object for object p_af
# rs_miss=RS(p_af, rsi)

# The shortcut is to find the inserted residues in p_af with respect to p_exp
rs_miss=p_af.rs_insertion(p_exp)

# now rs_miss is meaningful, as they point to the 5 Gs
print(rs_miss, rs_miss.seq(), "\n")

# now align the two structure and measure RMSD
print(p_exp.rmsd(p_af, rl_b=~rs_miss, ats="CA", align=True), "\n")
```

```
EIVLTQSPGTQSLSXXXXXTLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSLSTFGQGTKVEVKRTV

Notice five residues XXXXX corresponds to position 15-19

EIVLTQSPGTQSLSGGGGGTLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSLSTFGQGTKVEVKRTV

We run AlphaFold prediction using af_seq, i.e., replacing missing residues with Gly ...

L15-19
Pretend p_af is the output of AlphaFold prediction

Verify the missing resiudes are not missing in p_af, they are Gs:

EIVLTQSPGTQSLSGGGGGTLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSLSTFGQGTKVEVKRTV

# of residues: p_exp=106, p_af=111
AlphaFold predicted structure has 5 more residues, which needs to be excluded for alignment purpose.

L15-19 GGGGG

5.956938906837272e-15
```

Structure Prediction with ESMFold

We provide a fold() method, which uses Meta's free EMSFold web service to predict protein structures. Our method support multiple chains, as well as missing residues. The multi-chain support was based on the idea outlined in the "Merge & Split" section, i.e., by default we concatenate chains by 50 glycine residues (gap=50 can be modified) into one chain. Missing residues are replaced by glycines just as we did in the above AlphaFold example.

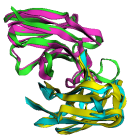
Please be aware that ESMFold generally produces less accurate results compared to AlphaFold. The web service limit the intermedidate sequence length to maximally 400 residues. Connection can fail after a few contiously predicted, as this public resource is protected from being over used. `fold()` is a quick way of occassionaly turning a sequence into a structure.

We here provide an example of predicting the antibody structure of 5CLI.

Let us try the Ab-Ag complex, which ESMFold failed to prediction.

```
os.environ['https_proxy']=''
p=Protein(fn)
exp=p.extract("H:L")
try:
    pred=Protein.fold(exp.seq())
    print(pred.rmsd(exp, ats="N,CA,C,O", align=True))
    Protein.merge([exp, pred]).show(color="chain")
except Exception as e:
    print(e)
    print("ESMFold service is not always stable. If you see error, try again!")
```

```
/Users/zhoyyi1/anaconda3/lib/python3.11/site-packages/urllib3/connectionpool.py:1100: InsecureRequestWarning: Unverified HTTPS request is being made to host 'api.esmatl
  warnings.warn(
2.455435560957574
```



```
p=Protein(fn)
exp=p.extract("H:L:P")
try:
    pred=Protein.fold(exp.seq())
    # algin by Ab only
    # predicted chains are A, B, C
    pred.rename_chains({"A":"H", "B":"L", "C":"P"})
    pred.align(exp, "H:L", "H:L", ats="N,CA,C,O")
    # print RMSD of antigens
    print(pred.rmsd(exp, "P", "P", ats="N,CA,C,O"))
    Protein.merge([exp, pred]).show(color="chain")
except Exception as e:
    print(e)
    print("ESMFold service is not always stable. If you see error, try again!")
```

```
/Users/zhoyyi1/anaconda3/lib/python3.11/site-packages/urllib3/connectionpool.py:1100: InsecureRequestWarning: Unverified HTTPS request is being made to host 'api.esmatl
  warnings.warn(
41.00484574766786
Rename chain: object 1, H to A
Rename chain: object 1, L to B
Rename chain: object 1, P to C
```

We can see that ESMFold predicts the antibody structure well, however, it got the antigen binding mode wrong. Instead of binding to the CDR loops, it places antigen on the other end. Ab-Ag complex prediction is an extremely difficult problem, the prediction accuracy was improved in the latest AlphaFold3 model.

Create Side Chains for de novo Designed Proteins

In the current AI-based protein design, RFDiffusion only generates a protein backbone. The output PDB file only contains coordinates for N, CA, C, O atoms. Often times, RFDiffusion works on an input template and was used to design only part of the structure (known as inpainting). For example, in an antibody design application, RFDiffusion may be only used to redesign the CDR loops, while leaving the framework residues and structures untouched. In the example below, we use RFDiffusion to design CDR H3 loops with the rest of the antibody sequence fixed. For convenience, let us refer to those fixed residues approximately as framework residues and the CDR H3 loop as CDR residues without creating confusion.

In the output PDB file generated by RFDiffusion, no sidechain atom exists, even for those framework residues. The generated CDR H3 backbone are represented by glycine residues. ProteinMPNN will preserve the identities of the input framework residues, while proposing CDR H3 residues to replace those glycines. In order to visualize the full-atom model of the design antibody-antigen complex structure, we can use AlphaFold to predict the structure from the ProteinMPNN sequences. However, AlphaFold prediction is time consuming and the predicted structure can be very different from the desired PDB structure due to the limited prediction accuracy on Ab-Ag complexes.

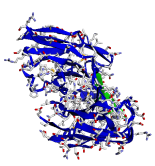
The method `thread_sequence()` helps us rapidly create a full-atom structure by threading the ProteinMPNN-generated sequence onto the RFDiffusion backbone-only structure. When the ProteinMPNN sequence is threaded onto the PDB backbone template, the side chain coordinates of those framework residues are now generated by PyMOL, which are sampled from their most frequent torsion angles and can be very different from their original known coordinates. For this reason, `thread_sequence()` takes an additional argument `side_chain_pdb`, which specifies the original PDB file that contains the side chain atoms for the framework residues (this is the PDB file used as the input for RFDiffusion). With this additonal input, the side chain coordinates for framework residues in the original PDB structure will be used instead of relying on PyMOL generation. Argument `seq2bfactor=True` maps the upper/lower-case of the input sequence onto b-factors (1.0/0.5), so that we can use b-factor to distinguish the redesigned residues (lower case) from those framework residues (upper case) preserved from the input template.

There are details inside the method that make use of several other **Afpdb** methods. For example:

- The predicted structure needs to be aligned with the `side_chain_pdb` first, before we are able to clone the coordinates of the side chain atoms. This is because the original PDB structure and the RFDiffusion-generated structure are not aligned by default.
- When RFDiffusion generates its output, chains are named A, B, C, etc. with the original chain names lost. As RFDiffusion can hallucinate new chains that do not exist in the original structure, it may not be able to preserve the original chain names even if it likes to. For this reason, we also need to provide a `chain_map` argument, so that the threading code knows how to map the wild-type chain names into the RFDiffusion chain names in order to align them correctly.
- Side chain coordinate cloning for the framework residues were done by manipulating the backend NumPy arrays.

```
# we first mimic a backbone-only structure, by changing all residues to Gly
q=Protein(pwd / "example_files/5cil_rfdiffuse_H3.pdb")
# rs_missing_atoms returns all residues where their sidechain atoms are missing
no_sc=q.rs_missing_atoms()
print("All non-Gly residues have missing side-chain atoms:", util.unique(no_sc.seq()), "\n")
seq_MPNN={
        'A':'EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSLSTFGQGTKVEVKRTV',
        'B':'NWFDITNWLWYIK',
        'C':'VQLVQSGAEVKRPGSSVTVSCKASGGSFSTYALSWVRQAPGRGLEWMGGVIPLLTITNYAPRFQGRITITADRSTSTAYLELNSLRPEDTAVYYCARhlvrtvgsgsnpemgdvvWGQGTLVTVSS'
       }
# we thread seq_MPNN onto RFDiffusion output structure q
# output a new PDB file: test.pdb
# copy sidechains of framework residues from template fn
# map the A/B/C chain names in the RFDiffusion structure to the L/P/H chains in the template fn
# seq2bfactor=True means we assign b-factor of value 0.5 for lower-cased seq_MPNN residues (those are CDR H3 residues redesigned)
# framework residues (upper case) have b-factor of value 1.0.
install_pymol()
q.thread_sequence(seq_MPNN, 'test.pdb', seq2bfactor=True, side_chain_pdb=fn, chain_map={"H":"C", "L":"A", "P":"B"})
q=Protein("test.pdb")
print("\nResidues with missing atoms:", q.rs_missing_atoms(), "\n")
q.show(color="b", show_sidechains=True)
```

```
All non-Gly residues have missing side-chain atoms: ['T', 'E', 'R', 'N', 'K', 'Y', 'F', 'P', 'V', 'Q', 'S', 'D', 'W', 'A', 'I', 'M', 'L', 'C']

RMSD after backbone alignment: 0.1905358537395586
Generated new input PDB:  /var/folders/fk/g7pd5s1n0gndf1_wwbqcftw00000gn/T/_THREADyaccukg2.pdb
MUTATE PyMOL> Old  C 98 GLY >>> New HIS
PyMOL>refresh_wizard
Selected!
 ExecutiveRMSPairs: RMSD =    0.007 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 9 rotamers loaded.
 Rotamer 3/9, strain=9.52
MUTATE PyMOL> Old  C 99 GLY >>> New LEU
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.007 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 9 rotamers loaded.
 Rotamer 6/9, strain=10.42
MUTATE PyMOL> Old  C 100 GLY >>> New VAL
Selected!PyMOL>refresh_wizard

 ExecutiveRMSPairs: RMSD =    0.006 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 3 rotamers loaded.
 Rotamer 1/3, strain=27.81
MUTATE PyMOL> Old  C 101 GLY >>> New ARG
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.006 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 81 rotamers loaded.
 Rotamer 39/81, strain=37.17
MUTATE PyMOL> Old  C 102 GLY >>> New THR
Selected!PyMOL>refresh_wizard

 ExecutiveRMSPairs: RMSD =    0.007 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 3 rotamers loaded.
 Rotamer 3/3, strain=40.63
MUTATE PyMOL> Old  C 103 GLY >>> New VAL
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.006 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 3 rotamers loaded.
 Rotamer 3/3, strain=47.80
MUTATE PyMOL> Old  C 105 GLY >>> New SER
Selected!PyMOL>refresh_wizard

 ExecutiveRMSPairs: RMSD =    0.006 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 3 rotamers loaded.
 Rotamer 1/3, strain=122.20
MUTATE PyMOL> Old  C 107 GLY >>> New SER
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.006 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 3 rotamers loaded.
 Rotamer 1/3, strain=78.86
MUTATE PyMOL> Old  C 108 GLY >>> New ASN
Selected!PyMOL>refresh_wizard

 ExecutiveRMSPairs: RMSD =    0.006 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 18 rotamers loaded.
 Rotamer 13/18, strain=92.73
MUTATE PyMOL> Old  C 109 GLY >>> New PRO
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.008 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 2 rotamers loaded.
 Rotamer 1/2, strain=74.34
MUTATE PyMOL> Old  C 110 GLY >>> New GLU
Selected!PyMOL>refresh_wizard

 ExecutiveRMSPairs: RMSD =    0.007 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 27 rotamers loaded.
 Rotamer 17/27, strain=45.93
MUTATE PyMOL> Old  C 111 GLY >>> New MET
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.007 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 27 rotamers loaded.
 Rotamer 5/27, strain=27.73
MUTATE PyMOL> Old  C 113 GLY >>> New ASP
Selected!PyMOL>refresh_wizard

 ExecutiveRMSPairs: RMSD =    0.007 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 9 rotamers loaded.
 Rotamer 7/9, strain=44.32
MUTATE PyMOL> Old  C 114 GLY >>> New VAL
Selected!
PyMOL>refresh_wizard
 ExecutiveRMSPairs: RMSD =    0.006 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 3 rotamers loaded.
 Rotamer 3/3, strain=20.24
MUTATE PyMOL> Old  C 115 GLY >>> New VAL
Selected!PyMOL>refresh_wizard

 ExecutiveRMSPairs: RMSD =    0.006 (3 to 3 atoms)
 Mutagenesis: no phi/psi, using backbone-independent rotamers.
 Mutagenesis: 3 rotamers loaded.
 Rotamer 3/3, strain=15.03
test.pdb
{'A': 'EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDFAVYYCQQYGQSLSTFGQGTKVEVKRTV', 'B': 'NWFDITNWLWYIK', 'C': 'VQLVQSGAEVKRPGSSVTV
###JSON STARTS
{"output_pdb": "test.pdb", "ok": true, "output_equal_target": true, "input": {"A": "EIVLTQSPGTQSLSPGERATLSCRASQSVGNNKLAWYQQRPGQAPRLLIYGASSRPSGVADRFSGSGSGTDFTLTISRLEPEDF
###JSON END

Residues with missing atoms:
```



In the RFDiffusion output file 5cil_rfdiffuse_H3.pdb, all 250 residues have missing atoms due to their missing side chains. After threading using the wild-type PDB 5cil.pdb as the template, we copy the side chain coordinates for all upper-case residues except the lower-case residues in the H3 CDR loop. The side chain atoms for the H3 loop are generated by PyMOL by replacing Glycine with corresponding ProteinMPNN-generated

residues. At the end, the final output test.pdb contains all atoms, therefore, no residue has missing atoms. The framework residues are colored in blue and the redesigned CDR H3 loop is colored in green.

Binding Score for EvoPro

EvoPro is a de novo protein design framework that iteratively evolutes protein binder sequences (https://doi.org/10.1073/pnas.2307371120). EvoPro starts from a pool of binder sequences; it uses AlphaFold to predict complex structures and then ranks the candidates by their EvoPro scores. A new canidate pool is generated with genetic algorithm based on a few top-scoring sequences. This process iterates until sequences of good scores are found.

We here demonstrate how EvoPro score, as described in the publication can be implemented straightforwardly with Afpdb. The score is the sum of three scores. The first is the placement confidence score that is the total number of interface residue pairs, weighted by AlphaFold PAE scores. The second is the fold confidence score based on AlphaFold's pLDDT scores. The third is the conformal stability score based on the RMSD between the binder structures as a monomer and as the binder of the complex.

The implementation is shown below:

```
# We score 5cli
# Ab-Ag predicted complex structure
p_complex = Protein(pwd / "example_files/5cil_AF.pdb")
p_complex.rename_chains({"A":"L", "B":"H", "C":"P"})
# load the PAE score generated by AF, PAE is the pairwise uncertainty in inter-residue distance
import json
pae = np.array(json.loads(util.read_string(pwd / "example_files/5cil_AF.json"))['pae'])
# AF uses b-factors to store pLDDT
plddt=p_complex.b_factors()
# Antibody predicted as a monomer
p_ab = Protein(pwd / "example_files/5cil_ab_AF.pdb")
p_ab.rename_chains({"A":"L", "B":"H"})

# Compute placement confidence
# identify all contact residue pairs
rs_ab, rs_ag, dist = p_complex.rs_around("P", rs_within="H:L", dist=4)
# extract PAE for contact residue pairs
# each row in dist DataFrame is one residue pair, they are weighted by PAE
# if PAE is small, we are more confident that the contact pair is real
pae_xy=pae[dist.resi_a.values, dist.resi_b.values]
pae_yx=pae[dist.resi_b.values, dist.resi_a.values]
# add negative, b/c the lower the score, the better
placement_score = -np.sum(((35-pae_xy) + (35-pae_yx))/70)

# Compute fold confidence
# pLDDT score is [0-100], scale it by ten
fold_score = -np.mean(plddt)/10

# Compute conformational stability score
# The argument is if the binder does not change its conformation after binding,
# it is more likely to bind
stability_score = p_ab.rmsd(p_complex, "H:L", "H:L", align=True)*5

# the lower the better
overall_fitness = placement_score + fold_score + stability_score

print(f"Fitness: {overall_fitness:.2f}")
print(f"Placement: {placement_score:.2f}")
print(f"Fold: {fold_score:.2f}")
print(f"Stability: {stability_score:.2f}")
```

```
Fitness: -39.96
Placement: -31.51
Fold: -9.73
Stability: 1.28
```

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit