# February 15, 2013

Copied directory "automated\_sequence\_alignment\gonnet aligned" to directory "family moments". See "automated\_sequence\_alignment/automated sequence alignment log.docx" for the story of that directory's creation. In short each file is a multiple sequence alignment containing an HHOMP cluster and a protein of known structure.

## Amount of Data Available

I took a look at the length of these alignments. Here's an excerpt from the Python 2.7 session:

>>> alignments = [(i, Bio.AlignIO.read(i, 'clustal')) **for** i **in** glob.glob('\*.clu')]

>>> **for** name, alignment **in** sorted(alignments, key=lambda x: len(x[1]))[::-1]:

**print**('{}: {}'.format(name, len(alignment)))

3EFM with cluster18.clu: 532

**1E54 with cluster28.clu: 320**

1QFG with 22.1.4.clu: 225

2VQI with nn.2.2.clu: 207

3BS0 with cluster71.clu: 195

3JTY with nn.9.1.clu: 173

1T16 with 14.1.1.clu: 163

2QDZ with cluster53.clu: 153

2F1V with 8.2.1.clu: 142

1QD6 with 12.6.1.clu: 86

1FEP with cluster8.clu: 82

1QJP with cluster75.clu: 79

**2J1N with cluster99.clu: 78**

1P4T with cluster144.clu: 77

**2O4V with 16.4.2.clu: 76**

3CSL with 22.4.6.clu: 68

2GUF with 22.4.5.clu: 62

1UYN with 12.1.6.clu: 58

**1A0S with cluster73.clu: 52**

2ERV with 8.4.1.clu: 45

1TLY with cluster108.clu: 37

2WJR with nn.36.1.clu: 36

1QJ8 with 8.3.1.clu: 32

1KMO with 22.2.4.clu: 32

3EMN with nn.54.1.clu: 26

1I78 with 10.1.1.clu: 16

1THQ with 8.5.1.clu: 15

1K24 with 10.2.1.clu: 3

I bolded some, maybe all of the trimers, whatever stood out at me.

I think that for every transmembrane residue, there's about 3 other transmembrane residues that share x and y coordinates with it. Out of this group of four, on average two of them will face outward. But if you multiply those two by only 15 sequences, that's already a decent statistical sample size.f

So, consider each (x,y) position along the rim of the barrel to be a sample. You get some sense of the average energy of this sample, and then you compare them to get a sense of where the interface might be.

There are about 23 non-trimeric porins and 4 trimeric porins with 15 or more sequences. Unless there's a really obvious difference it's going to be hard to draw any conclusions.

## Notes on calculating Ezβ

The code for calculating Ezβ is in "modules/zenergy.py". This module defines the "calculator" class, which has a single method, "calculate", that is a function of a residue type and a depth. See the docstrings of the class and the method for more information.

This module requires a parameter file. One that I have is "family moments/published params.csv". I don't seem to have any logs describing how I created it. However, I have compared it to Daniel, Vik and I's 2012 Protein Science paper and it's the parameters from that.

## Learning to use DSSP

Previously I've used DSSP taken from spreadsheets that Daniel made. I'd like to get away from using Daniel's data since I'm not always sure how to read it, I don't know where it came from, it makes my methods harder for others to understand since can't talk directly about where the information comes from, and I'd rather learn how to generate it myself.

I downloaded a the PDB structure file of 1A0S as " family moments\trying out dssp/1A0S.pdb". I then downloaded the corresponding DSSP file from the DSSP database, and generated a DSSP file myself, using the following code (excerpt from an iPython log, Python 2.7):

**import** subprocess

x = subprocess.output(['dssp', '1A0S.pdb'])

x = subprocess.check\_output(['dssp', '1A0S.pdb'])

x = subprocess.check\_output(['dssp', '1A0S.pdb', '1a0s home calculated.dssp'])

x

#[Out]# ''

urlretrieve('ftp://ftp.cmbi.ru.nl/pub/molbio/data/dssp/1a0s.dssp', filename="1a0s from dssp database.dssp")

urllib.urlretrieve('ftp://ftp.cmbi.ru.nl/pub/molbio/data/dssp/1a0s.dssp', filename="1a0s from dssp database.dssp")

#[Out]# ('1a0s from dssp database.dssp', <mimetools.Message instance at 0x036DD878>)

As you can see, not all of that was purposeful: it took me a few tries to generate the DSSP file.

I compared the two files, and there are only minor differences, numbers off by 1 or 2. At least, I think that's minor, but I didn't look closely at what the numbers mean. Apparently the version in the database was generated with a newer version of DSSP than I use.

# February 18, 2013

## Origin of PDB Structures

I copied all the PDB files from "pymol/structures" to "family moments/structures". These are files that Daniel gave me a long time ago. They are a single chain extracted from a PDB entry. Most β barrel structures are only one chain. However, for oligomers, this meant getting only one monomer. However, for 1QD6, a helix from each monomer is actually a separate chain. So, the 1QD6 structure is a monomer that is missing a helix. Importantly, they were all aligned, somehow, to be perpendicular to the xy plane.

## Fixing Bio.PDB.DSSP

Bio.PDB.DSSP provides an easy way to access DSSP data for a given structure, but is not, as written, usable on Windows. Since the DSSP program returns the information in a text file, this class has to first run DSSP and then parse the information. The way it does it is by creating a temporary file. However, it get an IOError: Permission Denied when it tries to read it. This is explained by the tempfile documentation (from http://docs.python.org/2/library/tempfile.html):

tempfile.**NamedTemporaryFile**([*mode='w+b'*[, *bufsize=-1*[, *suffix=''*[, *prefix='tmp'*[, *dir=None*[, *delete=True*]]]]]])

This function operates exactly as **[TemporaryFile()](http://docs.python.org/2/library/tempfile.html" \l "tempfile.TemporaryFile" \o "tempfile.TemporaryFile)** does, except that the file is guaranteed to have a visible name in the file system (on Unix, the directory entry is not unlinked). That name can be retrieved from the **name** attribute of the file object. **Whether the name can be used to open the file a second time, while the named temporary file is still open, varies across platforms (it can be so used on Unix; it cannot on Windows NT or later).** If *delete* is true (the default), the file is deleted as soon as it is closed.

I verified that this is the case on the computer I'm working on:

In [107]: f = tempfile.NamedTemporaryFile()

In [110]: g = open(f.name, 'r')

---------------------------------------------------------------------------

IOError Traceback (most recent call last)

<ipython-input-110-500a98f3ec02> **in** <module>()

----> 1 g = open(f.name, 'r')

IOError: [Errno 13] Permission denied: 'c:\\users\\nandal~1\\appdata\\local\\tem

p\\tmpioz7so'

At the cost of complicating the code, the functionality can be restored on Windows. Simply create a temporary file that is not deleted when closed. Close it before reading it, and then delete when done with os.remove. I tested that this would work:

In [112]: f = tempfile.NamedTemporaryFile(delete=False)

In [113]: f.write('lalalala')

In [114]: f.close()

In [116]: open(f.name, 'r').readlines()

Out[116]: ['lalalala']

In [117]: os.remove(f.name)

In [118]: open(f.name, 'r').readlines()

---------------------------------------------------------------------------

IOError Traceback (most recent call last)

<ipython-input-118-56c72550bf29> **in** <module>()

----> 1 open(f.name, 'r').readlines()

IOError: [Errno 2] No such file **or** directory: 'c:\\users\\nandal~1\\appdata\\loc

al\\temp\\tmpdvcllu'

I downloaded the Bio.PDB.DSSP module from biopython's repository on Github. I made the changes and saved it as "modules/DSSP\_win.py".

## Using Bio.PDB.DSSP

Suppose that "stru" is a structure taken from a PDB file using Biopython's PDBParser. A DSSP object is not created using "stru": rather, it is created using the *model* that "stru" contains. In addition, it requires the name of the file from which "stru" was created:

stru

#[Out]# <Structure id=3PRN>

stru.child\_dict

#[Out]# {0: <Model id=0>}

stru\_dssp = DSSP\_win.DSSP(stru.child\_dict[0], 'structures/aligned\_3PRN.pdb')

stru\_dssp

#[Out]# <DSSP\_win.DSSP instance at 0x049D17D8>

The DSSP object behaves like a dictionary. Rather than having residues as keys, it maps a duple of the chain ID and residue ID to information about the residue:

stru\_dssp.keys()[:10]

#[Out]# [('A', (' ', 1, ' ')),

#[Out]# ('A', (' ', 2, ' ')),

#[Out]# ('A', (' ', 3, ' ')),

#[Out]# ('A', (' ', 4, ' ')),

#[Out]# ('A', (' ', 5, ' ')),

#[Out]# ('A', (' ', 6, ' ')),

#[Out]# ('A', (' ', 7, ' ')),

#[Out]# ('A', (' ', 8, ' ')),

#[Out]# ('A', (' ', 9, ' ')),

#[Out]# ('A', (' ', 10, ' '))]

stru\_dssp[stru\_dssp.keys()[0]]

#[Out]# (<Residue MET het= resseq=1 icode= >,

#[Out]# '-',

#[Out]# 147,

#[Out]# 0.7819148936170213,

#[Out]# 360.0,

#[Out]# 140.5)

There does not seem to be any documentation on what these numbers represent. However, there is this line from the \_\_init\_\_ function of the DSSP object:

dssp\_map[key] = ((res, ss, acc, rel\_acc, phi, psi))

This suggests that the DSSP object maps to the residue itself, the secondary structure, the accessibility, the relative accessibility, the phi angle, and the psi angle.

## map\_resides\_to\_alignment\_positions

**from** sundries **import** one\_letter

**def** map\_residues\_to\_alignment\_positions(residues, sequence):

'''Given a list of residues from a Biopython structure and a sequence

    of the same protein from a sequence alignment,

    return a mapping from the residues to their positions in the sequence.

    Unfortunately this function depends critically on a flaw in my

    method of deriving moments. PDB structures often have residues missing,

    but in its current implementation this function will fail unless

    those residues are missing from the sequence as well. Due to a mistake

    generating the alignments, the resiudes ARE, in fact, missing from the

    sequences, so it works. But the alignments may be of lower quality

    due to that mistake.'''

# From the sequence, make an ordered list of pairs (index, resname)

# excluding gapped positions

numbered\_resnames = [pair **for** pair **in** enumerate(sequence) \

**if** pair[1] != '-']

# Make a mapping from residues to indices of the sequence

res\_to\_index = dict((res, pair[0]) \

**for** res, pair **in** zip(residues, numbered\_resnames))

# Check that the residues actually match

**for** res, index **in** res\_to\_index.items():

**if** one\_letter[res.get\_resname()] != sequence[index]:

**raise** ValueError("List of residues doesn't match sequence")

**return** res\_to\_index

Test of this function:

# Navigate to structures directory

get\_ipython().magic(u'cd structures/')

# Open ScrY structure

stru = Bio.PDB.PDBParser().get\_structure('aligned\_1A0S.pdb', '1A0S')

stru = Bio.PDB.PDBParser().get\_structure('1A0S', 'aligned\_1A0S.pdb')

warnings.simplefilter('ignore')

stru = Bio.PDB.PDBParser().get\_structure('1A0S', 'aligned\_1A0S.pdb')

# Navigate to alignments directory

get\_ipython().magic(u'cd ..')

get\_ipython().magic(u"cd 'gonnet aligned/'")

align = Bio.AlignIO.read('1A0S with cluster73.clu', 'clustal')

align[0].id

#[Out]# 'gi|3914230|sp|Q567'

align[1].id

#[Out]# 'gi|15601781|ref|NP'

align[2].id

#[Out]# 'gi|84394457|ref|ZP'

[seq **for** seq **in** align **if** 'template' **in** seq.id]

#[Out]# [SeqRecord(seq=Seq('------------------------------------------------------...---', SingleLetterAlphabet()), id='template\_1A0S', name='<unknown name>', description='template\_1A0S', dbxrefs=[])]

sequence = Out[245][0]

sequence

#[Out]# SeqRecord(seq=Seq('------------------------------------------------------...---', SingleLetterAlphabet()), id='template\_1A0S', name='<unknown name>', description='template\_1A0S', dbxrefs=[])

get\_ipython().magic(u'cd ..')

get\_ipython().magic(u'run family\_moments.py')

x = map\_residues\_to\_alignment\_positions(stru.get\_residues(), sequence)

x.items()[:10]

#[Out]# [(<Residue ILE het= resseq=235 icode= >, 328),

#[Out]# (<Residue GLY het= resseq=293 icode= >, 432),

#[Out]# (<Residue GLY het= resseq=207 icode= >, 300),

#[Out]# (<Residue VAL het= resseq=413 icode= >, 598),

#[Out]# (<Residue PRO het= resseq=171 icode= >, 261),

#[Out]# (<Residue ARG het= resseq=190 icode= >, 282),

#[Out]# (<Residue ILE het= resseq=198 icode= >, 290),

#[Out]# (<Residue THR het= resseq=208 icode= >, 301),

#[Out]# (<Residue ASP het= resseq=236 icode= >, 329),

#[Out]# (<Residue LEU het= resseq=182 icode= >, 274)]

sequence[328]

#[Out]# 'I'

sequence[432]

#[Out]# 'G'

sequence[300]

#[Out]# 'G'

sequence[598]

#[Out]# 'V'

# February 20, 2013

Wrote a function to calculate the geometric median:

**import** numpy as np

**def** geomed(starlist, quiet=False):

'''Calculate the geometric median aka Fermat point using Weiszfeld's

    algorithm. You can learn more about this from the Wikipedia article

    "Geometric Median"'''

# Throughout this function's comments, the metaphor is maintained

# that this function moves a spaceship from the origin to the

# geometric median of the surrounding stars by a series of smaller steps

# Start with the spaceship at the origin

ship = np.zeros(len(starlist[0]))

**for** i **in** range(1000):

# Create a matrix with the star positions as columns

pos\_mat = np.array(starlist).transpose()

# Create a vector where the nth entry is the reciprocal of the

# distance between the ship and the nth star

inverse\_distances = np.array([np.linalg.norm(star - ship)\*\*-1 \

**for** star **in** starlist])

# Find the next spot dictated by Weiszfeld's algorithm

destination = inverse\_distances.sum()\*\*-1 \

\* pos\_mat.dot(inverse\_distances)

# Break the loop if we're moving a billionth of an angstrom at a

# time

**if** np.linalg.norm(ship - destination) < 1e-12:

**break**

# Move the ship to the next spot

ship = destination

**if** **not** quiet:

**print**(str(i) + ' iterations')

**return** ship

I did a few basic tests with an equilateral triangle:

In [14]: triangle = [np.array([0,1]), np.array([sqrt(3)/2, -1/2]), np.array([-sq

rt(3)/2, -1/2])]

In [15]: triangle

Out[15]:

[array([0, 1]),

array([ 0.8660254, -0.5 ]),

array([-0.8660254, -0.5 ])]

In [20]: geomed(triangle, quiet=False)

0 iterations

Out[20]: array([ 0., 0.])

In [23]: geomed([i + np.array([1., 2.]) **for** i **in** triangle])

39 iterations

Out[23]: array([ 1., 2.])

Then I did some more serious testing with the 1A0S structure.

Running a script that creates "full\_sele", a selection that I've been using for testing containing ResidueDossier's generated from 1A0S  
In [26]: run family\_moments.py

!!! HEADER-card missing !!!

!!! COMPOUND-card missing !!!

!!! SOURCE-card missing !!!

!!! AUTHOR-card missing !!!

Retrieve the coordinates of each carbon alpha in the protein

In [29]: three\_d\_coords = [dos.ca\_coord **for** dos **in** full\_sele.dossiers]

In [30]: three\_d\_coords[:5]

Out[30]:

[array([-18.06399918, 9.22000027, -19.68199921], dtype=float32),

array([-20.24500084, 8.05900002, -16.78100014], dtype=float32),

array([-20.15500069, 5.66200018, -13.86699963], dtype=float32),

array([-17.23600006, 3.54699993, -12.71800041], dtype=float32),

array([-16.96100044, 0.92699999, -10.02299976], dtype=float32)]

Convert to 2D coordinates. Specifically, the projection on the yz plane. This was a mistake, I meant to do xy like I would when I'm actually calculating the moments, but it doesn't matter.

In [31]: coords = [triplet[1:] **for** triplet **in** three\_d\_coords]

In [32]: coords[:5]

Out[32]:

[array([ 9.22000027, -19.68199921], dtype=float32),

array([ 8.05900002, -16.78100014], dtype=float32),

array([ 5.66200018, -13.86699963], dtype=float32),

array([ 3.54699993, -12.71800041], dtype=float32),

array([ 0.92699999, -10.02299976], dtype=float32)]

Calculate the geometric median (metaphorically, place the spaceship)

In [35]: geomed(coords)

44 iterations

Out[35]: array([ 1.17956553, 4.78973498])

In [38]: ship = Out[35]

In [39]: ship

Out[39]: array([ 1.17956553, 4.78973498])

A first attempt at testing whether it really is the geometric median, but I accidentally did the calculation that comes out to zero for the geometric **mean**

In [40]: distances = [star - ship **for** star **in** coords]

In [41]: sum(distances)

Out[41]: array([ 133.95444499, 37.509462 ])

Actually testing if it's the geometric median - summing unit vectors pointing from the ship to each star (that is, from the proposed geometric median to each carbon alpha)

In [43]: directions = [np.linalg.norm(star-ship)\*\*-1 \* (star-ship) **for** star **in** c

oords]

In [44]: directions[:3]

Out[44]:

[array([ 0.31214355, -0.95003495]),

array([ 0.30384605, -0.95272114]),

array([ 0.23361039, -0.97233029])]

In [45]: directions[-3:]

Out[45]:

[array([-0.67488628, -0.73792175]),

array([-0.46477646, -0.88542806]),

array([-0.29181676, -0.95647424])]

In [46]: sum(directions)

Out[46]: array([ -2.45803933e-11, 8.75988171e-12])

Trying it on the full 3D coordinates - I have no plans ever to use this function on 3D coordinates, but my understanding of the function told me that this would work, so I wanted to try it

In [47]: three\_d\_ship = geomed(three\_d\_coords)

32 iterations

In [48]: three\_d\_ship

Out[48]: array([ 0.69068641, 1.28564143, 5.14336523])

In [50]: three\_d\_directions = [np.linalg.norm(star-three\_d\_ship)\*\*-1 \* (star-thr

ee\_d\_ship) **for** star **in** three\_d\_coords]

In [51]: sum(three\_d\_directions)

Out[51]: array([ -2.24931185e-12, 2.08670581e-11, -8.20565838e-13])