# 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#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])

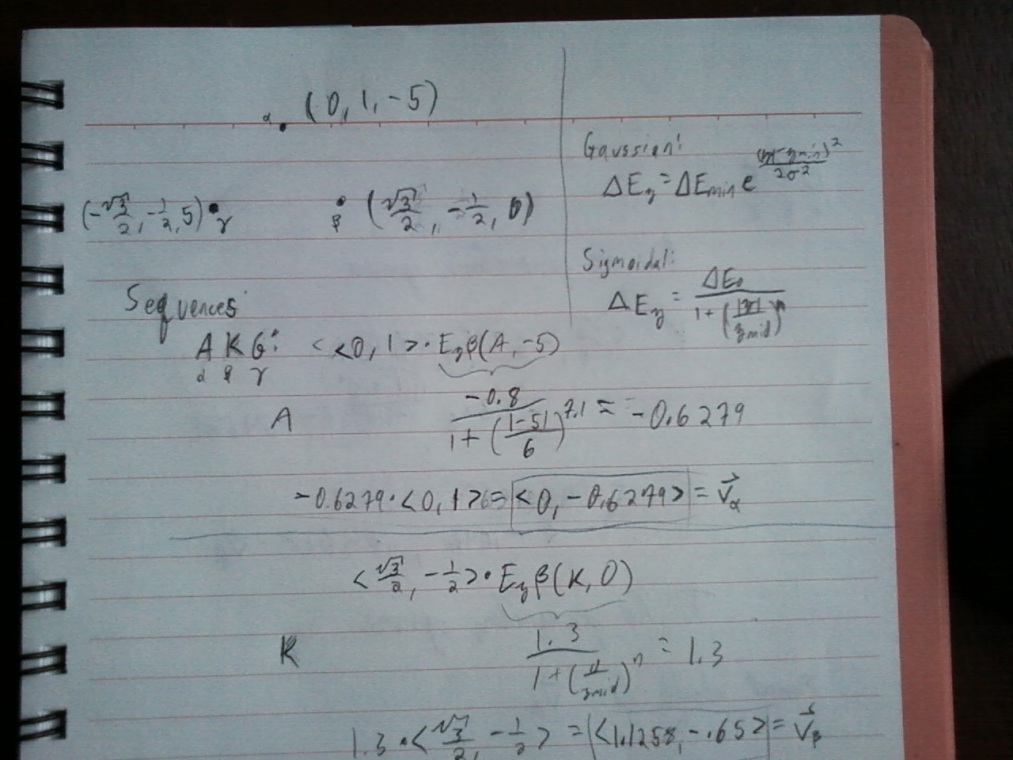
# February 25, 2013

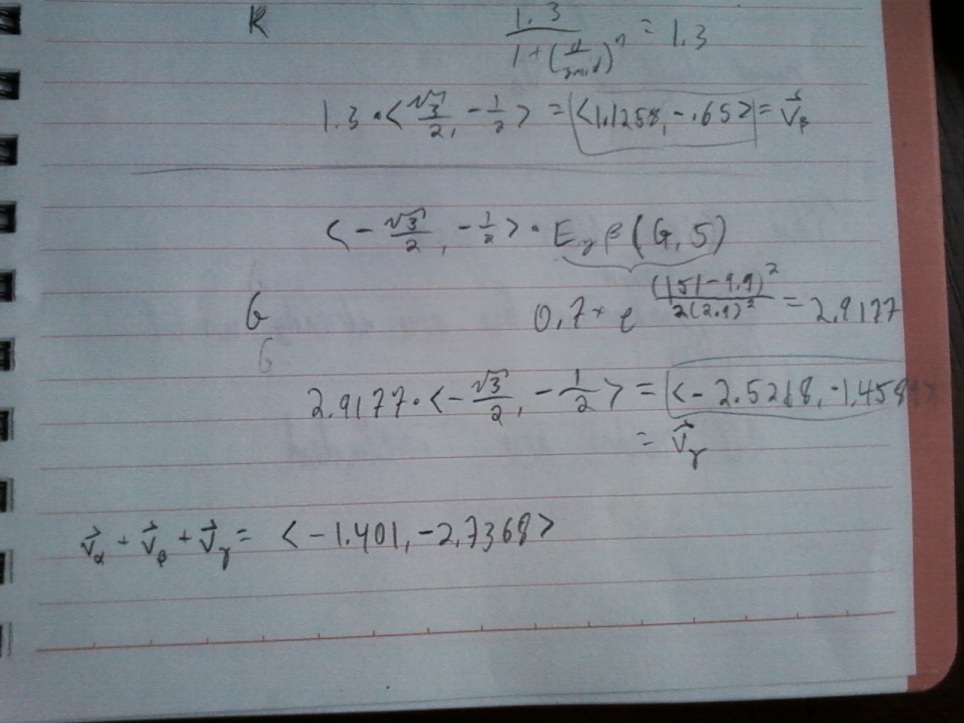
Testing the Ezβ moment against hand calculations.

## Equilateral triangle of radius 1, two sequences

Looked at from above (that is, projected onto the xy plane), C-alphas are at the corners of an equilateral triangle centered on the origin with one corner on (0,1). Starting at the y axis and progressing clockwise, the z-coordinates are -5, 0 and 5. The two sequences (again, starting at the y axis and progressing clockwise) are AKG and CYT. Checking that the computer reproduces these calculations will not check that position vectors are being normalized properly, and it does not check that selections are made properly.

### Hand calculations for first sequence





I doublechecked these calculations in IPython 0.13 using Python 2.7.2. Here's the unabridged log:

Importing necessary modules

**import** numpy **as** np

**from** math **import** **\***

get\_ipython**().**system**(**u'dir /on '**)**

Initially screwed up defining the position vectors, so that "north" was pointing straight west:

north **=** np**.**array**([**cos**(**pi**),** sin**(**pi**),** **-**5**])**

southeast **=** np**.**array**([**cos**(**pi **-** 1**\*(**2**\***pi**/**3**)),** sin**(**pi **-** 1**\*(**2**\***pi**/**3**)),** 0**])**

southwest **=** np**.**array**([**cos**(**pi **-** 2**\*(**2**\***pi**/**3**)),** sin**(**pi **-** 2**\*(**2**\***pi**/**3**)),** 5**])**

north **-** np**.**array**([**0**,** 1**,** **-**5**])**

#[Out]# array([-1., -1., 0.])

Correctly defined the position vectors (north is position α, southeast is position β, southwest is position γ)

north **=** np**.**array**([**cos**(**pi**/**2**),** sin**(**pi**/**2**),** **-**5**])**

southeast **=** np**.**array**([**cos**(**pi**/**2 **-** 1**\*(**2**\***pi**/**3**)),** sin**(**pi**/**2 **-** 1**\*(**2**\***pi**/**3**)),** 0**])**

southwest **=** np**.**array**([**cos**(**pi**/**2 **-** 2**\*(**2**\***pi**/**3**)),** sin**(**pi**/**2 **-** 2**\*(**2**\***pi**/**3**)),** 5**])**

Checked that they were equal to the vectors used in handwriting:

north **-** np**.**array**([**0**,** 1**,** **-**5**])**

#[Out]# array([ 6.12323400e-17, 0.00000000e+00, 0.00000000e+00])

**from** \_\_future\_\_ **import** division

southeast **-** np**.**array**([**sqrt**(**3**)/**2**,** **-**1**/**2**,** 0**])**

#[Out]# array([ 1.11022302e-16, 1.66533454e-16, 0.00000000e+00])

southwest **-** np**.**array**([-**sqrt**(**3**)/**2**,** **-**1**/**2**,** 0**])**

#[Out]# array([ 1.11022302e-16, -3.33066907e-16, 5.00000000e+00])

Coped functions to calculate Ezβ from my handwritten notes:

**def** sigmoidal**(**e0**,** zmid**,** n**,** z**):**

**return** e0**/(**1**+(**abs**(**z**)/**zmid**)\*\***n**)**

**def** gaussian**(**emin**,** zmin**,** sd**,** z**):**

**return** emin **\*** exp**(((**abs**(**z**)-**zmin**)\*\***2**)/(**2**\***sd**\*\***2**))**

Copied parameters from the Ezβ paper:

alanine\_params **=** **[-**.8**,** 6**,** 7.1**]**

lysine\_params **=** **[**1.3**,** 14**,** 3.7**]**

glycine\_params **=** **[**.7**,** 9.9**,** 2.9**]**

A few errors while trying to calculate vα:

valpha **=** sigmoidal**(**alanine\_params**\*,** **-**5**)** **\*** north

valpha **=** sigmoidal**(\***alanine\_params**,** **-**5**)** **\*** north

valpha **=** sigmoidal**(\***alanine\_params**,** z**=-**5**)** **\*** north

valpha

#[Out]# array([ -3.84492571e-17, -6.27924021e-01, 3.13962010e+00])

Calculated ~~vβ~~ vα and found that it matched my handwritten notes:

valpha **=** sigmoidal**(\***alanine\_params**,** z**=-**5**)** **\*** north**[:**2**]**

valpha

#[Out]# array([ -3.84492571e-17, -6.27924021e-01])

Calculated vβ and found that it matched my handwritten notes:

vbeta **=** sigmoidal**(\***lysine\_params**,** z**=**0**)** **\*** southeast**[:**2**]**

vbeta

#[Out]# array([ 1.12583302, -0.65 ])

An error

vgamma **=** gaussian**(\***glycine\_params**,** z**=**5**)**

vgamma

#[Out]# 2.9176904364436074

Calculated vγ and found that it matched my handwritten notes

vgamma **=** gaussian**(\***glycine\_params**,** z**=**5**)** **\*** southwest**[:**2**]**

vgamma

#[Out]# array([-2.52679404, -1.45884522])

Summed them up to an Ezβ moment and found that it matched my handwritten notes

valpha **+** vbeta **+** vgamma

#[Out]# array([-1.40096101, -2.73676924])

---

Inserted after beginning to test the moments generated by the code

There was a mistake in my hand calculations. When I calculated the Ezβ of glycine, I left the negative sign out of the exponent.

My calculation:

In [74]: .7\*math.exp((abs(5)-9.9)\*\*2/(2\*2.9\*\*2))

Out[74]: 2.9176904364436074

Correct calculation:

In [75]: .7\*math.exp(-(abs(5)-9.9)\*\*2/(2\*2.9\*\*2))

Out[75]: 0.1679410515521531

So, v\_gamma should really be <-- no, its <-.1455, -.084>. The calculation that led to this was:

In [76]: -.1680\*np.array([-2.5268, -1.4589])

Out[76]: array([ 0.4245024, 0.2450952])

I don't even remember where those numbers came from.

Also, that should really be .1679 not .1680, I rounded the wrong direction, but *that's* no big deal.

~~Adding that to the v\_alpha and v\_beta that I had calculated before, I get~~

~~In [77]: np.array([ -3.84492571e-17, -6.27924021e-01]) + np.array([ 1.12583302,~~

~~-0.65 ]) + np.array([.4245, .2451])~~

~~Out[77]: array([ 1.55033302, -1.03282402])~~

~~That's my predicted moment: <1.550, -1.033>~~

In [82]: valpha = np.array([ -3.84492571e-17, -6.27924021e-01])

In [83]: vbeta = np.array([ 1.12583302, -0.65 ])

In [84]: new\_vgamma = .1679 \* np.array([-math.sqrt(3)/2, -.5])

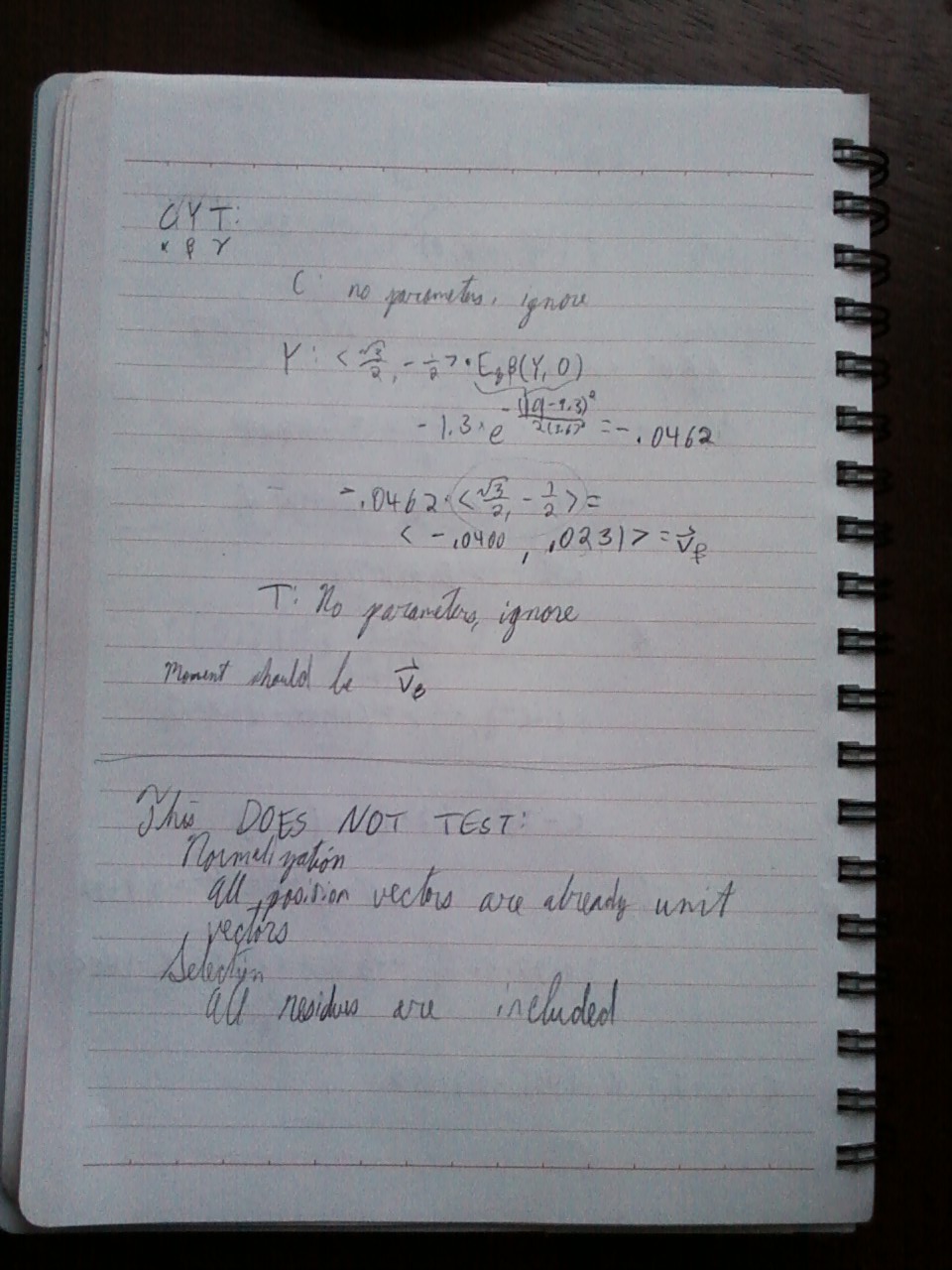
In [85]: valpha + vbeta + new\_vgamma

Out[85]: array([ 0.98042735, -1.36187402])

*That's* my predicted moment. <0.9804, -1.36187>.

---

### Hand calculations for second sequence



### Constructing the triangle in PyMOL

#### The pseudoatom command and the format of its parameters

Creating an atom in PyMOL can be accomplished with the pseudoatom command. The usage of this command, from the PyMOL wiki:

pseudoatom object [, selection [, name [, resn [, resi [, chain

[, segi [, elem [, vdw [, hetatm [, b [, q [, color [, label

[, pos [, state [, mode [, quiet ]]]]]]]]]]]]]]]]]

I will need to set the name so that it will be treated as a Cα, the resn so it will be treated as the right kind of residue, and the position so it will be in the right place. I will also add a chain identifier and a residue number (resi).

To get an idea of the format of these parameters in PyMOL, I downloaded the PDB structure 1A0S using "fetch 1A0S", clicked a random residue, and examined these properties:

You clicked **/**1a0s**//**P**/**THR**`**135**/**CB

Selector: selection "sele" defined with 7 atoms.

PyMOL**>**iterate\_state 1**,** sele**,** **print** **(**name**,** resn**,** **(**x**,**y**,**z**),** chain**,** resi**)**

**(**'N'**,** 'THR'**,** **(-**53.90399932861328**,** 1.819000005722046**,** 4.491000175476074**),** 'P'**,** '135'**)**

**(**'CA'**,** 'THR'**,** **(-**53.619998931884766**,** 2.4549999237060547**,** 3.2290000915527344**),** 'P'**,** '135'**)**

**(**'C'**,** 'THR'**,** **(-**52.520999908447266**,** 1.5889999866485596**,** 2.6080000400543213**),** 'P'**,** '135'**)**

**(**'O'**,** 'THR'**,** **(-**51.67599868774414**,** 1.0390000343322754**,** 3.3269999027252197**),** 'P'**,** '135'**)**

**(**'CB'**,** 'THR'**,** **(-**53.159000396728516**,** 3.9030001163482666**,** 3.510999917984009**),** 'P'**,** '135'**)**

**(**'CG2'**,** 'THR'**,** **(-**51.957000732421875**,** 4.283999919891357**,** 2.7209999561309814**),** 'P'**,** '135'**)**

**(**'OG1'**,** 'THR'**,** **(-**54.24800109863281**,** 4.797999858856201**,** 3.265000104904175**),** 'P'**,** '135'**)**

IterateState**:** iterated over 7 atom coordinate states**.**

It seems that Cα's are denoted with name='CA', residue names are all-caps three-letter codes, chain ID's are letters, and residue identifiers are strings with integer characters.

#### Creating the PDB file

Using PyMOL 1.5.0.4 executable build, which contains Python with sys.version equal to "2.7 (r27:82500, Nov 12 2010, 13:12:44) [MSC v.1600 64 bit (AMD64)]"

I created the three atoms with the pseudoatom command, with the second sequence CYT, using the following commands in PyMOL:

PyMOL**>**rein

PyMOL**>**pseudoatom triangle**,** name**=**CA**,** resn**=**CYS**,** pos**=(**0**,**1**,-**5**),** chain**=**A**,** resi**=**1

ObjMol**:** created triangle**/**PSDO**/**A**/**CYS**`**1**/**CA

PyMOL**>**pseudoatom triangle**,** name**=**CA**,** resn**=**TYR**,** pos**=(**math**.**sqrt**(**3**)/**2**,** **-**.5**,** 0**),** chain**=**A**,** resi**=**2

ObjMol**:** created triangle**/**PSDO**/**A**/**TYR**`**2**/**CA

PyMOL**>**pseudoatom triangle**,** name**=**CA**,** resn**=**THR**,** pos**=(-**math**.**sqrt**(**3**)/**2**,** **-**.5**,** 5**),** chain**=**A**,** resi**=**3

ObjMol**:** created triangle**/**PSDO**/**A**/**THR**`**3**/**CA

However, when loading this into Biopython I got a warning:

c:\python27\lib\site-packages\Bio\PDB\Atom.py:98: PDBConstructionWarning: Used e

lement 'C' for Atom (name=CA) with given element 'PS'

warnings.warn(msg, PDBConstructionWarning)

This seems alright, since I never access "element" and in any case "C" is the right element to use.

Also, DSSP does not give any output for this structure, since there is no backbone. For this reason, I am using a modified version of the code with 3 lines that use DSSP data commented out. After testing and debugging I will copy the compete moment-generating code, with this modification, into this log.

A more serious problem is after saving the triangle as a PDB file, and reopening it in PyMOL, I found that the x coordinates were about .32, rather than sqrt(3)/2 ≈ .86. The y and z coordinates were as expected. However, if I simply repeat the commands in Python, using PyMOL's cmd module, the coordinates of the atoms are as entered. When you use a PyMOL command, all the arguments are strings. It's only by using the cmd module directly that you can enter non-string arguments. So, since pseudoatom takes a tuple of reals as an argument, it must have some way of converting the string representation into the real thing. It probably doesn't work perfectly, explaining why math.sqrt(3)/2 somehow becomes .32: the command is actually getting a *string* that has the letters "math.sqrt(3)/2" in it, and not properly converting it to a number. Using the pseudoatom command directly, rather than through PyMOL's command parser, fixes this problem.

So, I recreated the triangle by invoking the cmd module directly, and this time I also set the elements equal to 'C':

PyMOL**>**cmd**.**pseudoatom**(**'triangle'**,** name**=**'CA'**,** resn**=**'CYS'**,** pos**=(**0**,**1**,-**5**),** chain**=**'A'**,** resi**=**'1'**,** elem**=**'C'**)**

PyMOL**>**cmd**.**pseudoatom**(**'triangle'**,** name**=**'CA'**,** resn**=**'TYR'**,** pos**=(**math**.**sqrt**(**3**)/**2**,** **-**.5**,** 0**),** chain**=**'A'**,** resi**=**'2'**,** elem**=**'C'**)**

PyMOL**>**cmd**.**pseudoatom**(**'triangle'**,** name**=**'CA'**,** resn**=**'THR'**,** pos**=(-**math**.**sqrt**(**3**)/**2**,** **-**.5**,** 5**),** chain**=**'A'**,** resi**=**'3'**,** elem**=**'C'**)**

And confirmed that the coordinates came out right:

PyMOL**>**iterate\_state 1**,** triangle**,** **print((**x**,**y**,**z**))**

**(**0.0**,** 1.0**,** **-**5.0**)**

**(**0.8660253882408142**,** **-**0.5**,** 0.0**)**

**(-**0.8660253882408142**,** **-**0.5**,** 5.0**)**

IterateState**:** iterated over 3 atom coordinate states**.**

### Creating the multiple sequence alignment

The sequences in this artificial Triangle family are AKG and CYT. I will create a multiple sequence alignment in Clustal format with these two sequences, named "alternate" and "structure", respectively, since the second one matches the residue names of the PDB structure of the triangle. Here's the abridged IPython session, using IPython 0.13, with Python with sys.version equal to '2.7.2 (default, Jun 12 2011, 15:08:59) [MSC v.1500 32 bit (Intel)]' and Biopython with Bio.\_\_version\_\_ equal to '1.57'.

In [92]: reset

Once deleted, variables cannot be recovered. Proceed (y/[n])? y

In [93]: from Bio.Seq import Seq

In [94]: alt\_seq = Seq('AKG')

In [95]: stru\_seq = Seq('CYT')

In [97]: from Bio.AlignIO import SeqRecord

In [98]: alt\_rec = SeqRecord(alt\_seq, id='alternate')

In [99]: stru\_rec = SeqRecord(stru\_seq, id='structure')

In [100]: from Bio.AlignIO import MultipleSeqAlignment

In [102]: msa = MultipleSeqAlignment([alt\_rec, stru\_rec])

In [103]: msa[0]

Out[103]: SeqRecord(seq=Seq('AKG', Alphabet()), id='alternate', name='<unknown n

ame>', description='<unknown description>', dbxrefs=[])

In [104]: msa[1]

Out[104]: SeqRecord(seq=Seq('CYT', Alphabet()), id='structure', name='<unknown n

ame>', description='<unknown description>', dbxrefs=[])

In [105]: import Bio.AlignIO

In [106]: Bio.AlignIO.write(msa, 'triangle.clu', 'clustal')

Out[106]: 1

### Calculating the moments

I will not keep a log of successive rounds of testing and debugging. I will only log errors that I think are interesting and worth remembering, so that they can be avoided. Then, when the code is completely debugged, I will log a victory test in which I demonstrate that everything is working as expected.

For this I am using IPython 0.13, with Python with sys.version equal to '2.7.2 (default, Jun 12 2011, 15:08:59) [MSC v.1500 32 bit (Intel)]' and Biopython with Bio.\_\_version\_\_ equal to '1.57'.

The geometric median of the triangle is slightly off:

In [29]: tri\_sele.geomed

Out[29]: array([ 0.00000000e+00, -1.46688072e-05])

This is not because it does not converge within the maximum number of iterations:

In [30]: geomed([dos.ca\_coord[:2] for dos in tri\_sele], quiet=False)

23 iterations

Out[30]: array([ 0.00000000e+00, -1.46688072e-05])

Rather, it seems to be because of rounding in the coordinates:

In [31]: tri\_sele[1]

Out[31]: <\_\_main\_\_.ResidueDossier at 0x4bceed0>

In [32]: tri\_sele[1].ca\_coord

Out[32]: array([ 0.866, -0.5 , 0. ], dtype=float32)

Not a big deal, and I don't know what I could do about it anyway.

I made some changes to the code used to generate the above geometric median results, but I didn't change anything that would affect those results.

The hand-calculated moments for the structural sequence (CYK) and the alternate sequence (AKG), after some correction that took place in the process of debugging the moment code, were <-.0400, .023> and <0.9804, -1.36187>, respectively (see above). The actual calculated moments, after debugging, matched:

In [108]: tri\_sele.moment('structure')

Out[108]: array([-0.04002375, 0.02310772])

In [109]: tri\_sele.moment('alternate')

Out[109]: array([ 0.98039181, -1.36189455])

The debugged code on which this was run:

**from** \_\_future\_\_ **import** division

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

**import** warnings

**from** Bio**.**PDB **import** PDBParser

**import** Bio**.**AlignIO

**import** DSSP\_win

**import** zenergy

**import** csv

**import** numpy **as** np

#from pymol import cmd

**def** draw\_vector**(**name**,** vector**,** center**):**

"""Create a distance with the length and direction of a given vector,

and then hide its label.

The center should have the same xy projection as the center you gave to

the moment function, but you might want to change the z value to make it

more visible.

ARGUMENTS:

(name, vector\_, center)

RETURNS:

Nothing."""

cmd**.**pseudoatom**(**'ps1'**,** pos **=** tuple**(**center**))**

cmd**.**pseudoatom**(**'ps2'**,** pos **=** tuple**(**center **+** vector**))**

cmd**.**distance**(**name**,** 'ps1'**,** 'ps2'**)**

cmd**.**hide**(**'labels'**,** name**)**

cmd**.**delete**(**'ps1 ps2'**)**

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

'''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 \

**\*** np**.**dot**(**pos\_mat**,** 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

**def** map\_res\_to\_pos**(**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

**class** **Family(**object**):**

'''A class that compiles information about a protein structure and

related sequences. This information is meant to be sufficient to filter

the residues and calculate an ez-beta moment from those that remain.

Attributes:

stru\_name: Name of the structure

stru\_path: Path of the PDB format structure file

stru: the structure, as a Biopython entity

msa: a dictionary mapping sequence identifiers to rows in a multiple

sequence alignment

template\_seq: the row of the MSA containing the sequence of the

structure

res\_to\_pos: a dictionary mapping residues from structures to their

column number in the MSA (asssuming the first column is numbered 0)

dssp: a Biopython DSSP object with a DSSP for the structure

calc: an Ez-beta calculator'''

**def** \_\_init\_\_**(**self**,** stru\_name**,** stru\_path**,** msa\_path**,** template\_name**,**

param\_path**):**

'''Requires a name for the structure (your choice), a path to a

PDB format structure file, a path to a multiple sequence alignment

containing a row with exactly the same sequence as the structure,

the sequence identifier of this row, and a path to a CSV file

of Ez-beta parameters (see zenergy.Calculator for how to make

these files)'''

self**.**stru\_name **=** stru\_name

self**.**stru\_path **=** stru\_path

**with** warnings**.**catch\_warnings**():**

warnings**.**simplefilter**(**'ignore'**)**

self**.**stru **=** PDBParser**().**get\_structure**(**stru\_name**,** stru\_path**)**

# When Daniel created the aligned structures, he removed heteroatoms

# (though the procedure he used seems to have removed anything

# without the residue identifer of one of the 20 standard amino

# acids, leading to main chain selenomethionines being removed from

# the 1FEP structure). However, he also added a sort of box of

# water atoms (perhaps as a visual aid, so you can tell how the

# coordinate system is defined?)

# Therefore, remove all waters

waters **=** **[**i **for** i **in** self**.**stru**.**get\_residues**()** \

**if** i**.**get\_resname**()** **==** 'HOH'**]**

**for** chain **in** self**.**stru**.**get\_chains**():**

**for** water **in** waters**:**

**try:**

chain**.**detach\_child**(**water**.**get\_id**())**

# Maybe it's not in this chain

**except** KeyError**:**

**pass**

msa **=** Bio**.**AlignIO**.**read**(**open**(**msa\_path**),** 'clustal'**)**

self**.**msa **=** dict**((**seq**.**id**,** seq**)** **for** seq **in** msa**)**

self**.**template\_seq **=** self**.**msa**[**template\_name**]**

self**.**res\_to\_pos **=** map\_res\_to\_pos**(**self**.**stru**.**get\_residues**(),**

self**.**template\_seq**)**

self**.**dssp **=** DSSP\_win**.**DSSP**(**self**.**stru**.**child\_dict**[**0**],** stru\_path**)**

params **=** csv**.**reader**(**open**(**param\_path**,** 'rb'**))**

self**.**calc **=** zenergy**.**Calculator**(**params**)**

**class** **ResidueDossier(**object**):**

'''A class that compiles the information about a residue that I expect

to use directly in moment calculations, so I don't have to remember

how to get each piece of information from the Structure object.

Attributes:

ca\_coord: c alpha coordinate in three dimensions

rel\_acc: relative accessibility from DSSP

Method:

ez\_b(self, seq\_id): calculate Ez-beta using a particular sequence

in the family

Two more attributes for if you need other information for some reason:

residue: the Biopython residue object

family: the Family object that this residue came from'''

**def** \_\_init\_\_**(**self**,** residue**,** family**):**

self**.**residue **=** residue

self**.**family **=** family

self**.**ca\_coord **=** residue**.**child\_dict**[**'CA'**].**get\_coord**()**

# chain\_id = residue.get\_full\_id()[2]

# extended\_resi = residue.get\_full\_id()[3]

# self.rel\_acc = family.dssp[(chain\_id, extended\_resi)][3]

self**.**resi **=** residue**.**get\_id**()[**1**]**

**def** ez\_b**(**self**,** seq\_id**):**

'''Calulate the Ez-Beta insertion energy of this residue, assuming

the residue type that it has in the sequence with the given id.

Will raise a NoParameters exception for a gap, or for a residue

for which there are no parameters (either because they are too

rare to determine their depth-dependent frequency trends, like

cysteine, or because they apparently do not have depth-dependent

frequency trends, like threonine).'''

column **=** self**.**family**.**res\_to\_pos**[**self**.**residue**]**

resn **=** self**.**family**.**msa**[**seq\_id**][**column**]**

z **=** self**.**ca\_coord**[**2**]**

**return** self**.**family**.**calc**.**calculate**(**resn**,** z**)**

**class** **Selection(**list**):**

**def** \_\_init\_\_**(**self**,** family**,** inclusion\_condition**):**

filing\_cabinet **=** **[**ResidueDossier**(**res**,** family**)**\

**for** res **in** family**.**stru**.**get\_residues**()]**

list**.**\_\_init\_\_**(**self**,** **(**dos **for** dos **in** filing\_cabinet \

**if** inclusion\_condition**(**dos**)))**

self**.**family **=** family

# Calculate the geometric median and save it right now,

# so that it doesn't have to be recalculated

# every time the moment is calculated with a different sequence

xy\_coords **=** **[**dos**.**ca\_coord**[:**2**]** **for** dos **in** self**]**

self**.**geomed **=** geomed**(**xy\_coords**)**

**def** show**(**self**):**

cmd**.**reinitialize**()**

cmd**.**load**(**self**.**family**.**stru\_path**,** self**.**family**.**stru\_name**)**

cmd**.**color**(**'purple'**,** '\*'**)**

**for** dos **in** self**:**

cmd**.**color**(**'green'**,** 'resi ' **+** str**(**dos**.**resi**))**

**def** moment**(**self**,** seq\_id**):**

running\_total **=** np**.**zeros**(**2**)**

**for** dos **in** self**:**

# Calculate ez-beta; skip this residue if this position is a gap

# or if there is no information on this kind of residue

**try:**

ez\_b **=** dos**.**ez\_b**(**seq\_id**)**

**except** zenergy**.**NoParameters**:**

**continue**

# Calculate vector that points from the geometric median of the

# structure, to this residue's c-alpha, with magnitude

# equal to it's ez-beta

# Then, add it to the running total

relative\_position **=** dos**.**ca\_coord**[:**2**]** **-** self**.**geomed

relative\_direction **=** relative\_position\

**/** np**.**linalg**.**norm**(**relative\_position**)**

running\_total **+=** ez\_b **\*** relative\_direction

**return** running\_total

**def** show\_moment**(**self**,** seq\_id**,** normalize**=**False**,** name **=** **None,** z**=**0**):**

**if** name **is** **None:**

name **=** seq\_id

mx**,** my **=** self**.**moment**(**seq\_id**)**

**if** **not** normalize**:**

moment **=** np**.**array**([**mx**,** my**,** 0**])**

**if** normalize**:**

moment **=** **(**mx**\*\***2 **+** my**\*\***2 **+** z**\*\***2**)\*\*(-**.5**)** **\*** np**.**array**([**mx**,**my**,**z**])**

gx**,** gy **=** self**.**geomed

geomed **=** np**.**array**([**gx**,** gy**,** z**])**

draw\_vector**(**name**,** moment**,** geomed**)**

# Tests

real\_family **=** Family**(**'1A0S'**,** 'structures/aligned\_1A0S.pdb'**,**

'gonnet aligned/1A0S with cluster73.clu'**,** 'template\_1A0S'**,**

'published params.csv'**)**

triangle **=** Family**(**'triangle'**,** 'triangle.pdb'**,** 'triangle.clu'**,** 'structure'**,**

'published params.csv'**)**

tri\_sele **=** Selection**(**triangle**,** **lambda** x**:** True**)**

So unless I made the same mistake in the hand-calculations, there are probably no mistakes in the code to calculate the moments. Unless they're part of normalizing the position vectors, making selections, or calculating the properties used to make selections.