# August 2

From doi:10.1371/journal.pone.0026400 on the PLoS website, downloaded Figures S2 and S3.

Copied Qall from Figure S2 into a text file, "parser test.txt", to test my parser.

Checking a few values at random.

Row N, column Q should be 2.511.

>>> x['N']['Q']

'2.511'

Row E, column S should be .728.

>>> x['E']['S']

'0.728'

Row S, column E should be .728.

>>> x['S']['E']

'0.728'

Here's the working code:

**def** parse**(**mat\_file**):**

found\_resns **=** False

**for** line **in** mat\_file**:**

# Ignore comments

**if** line**[**0**]** **==** '#'**:**

**continue**

# Find which resn corresponds to which column number

**if** **not** found\_resns**:**

found\_resns **=** True

col\_names **=** line**.**split**()**

# Check to make sure they're all there

**for** resn **in** **[**'C'**,** 'N'**,** 'H'**,** 'D'**,** 'S'**,** 'Q'**,** 'K'**,** 'M'**,** 'P'**,**

'T'**,** 'F'**,** 'A'**,** 'G'**,** 'I'**,** 'L'**,** 'R'**,** 'W'**,** 'E'**,**

'Y'**,** 'V'**]:**

**assert** resn **in** col\_names**,** "missing " **+** resn

# Make the matrix that will be returned

output **=** dict**([(**resn**,** **None)** **for** resn **in** col\_names**])**

**for** key**,** value **in** output**.**items**():**

output**[**key**]** **=** dict**([(**resn**,** **None)** **for** resn **in** col\_names**])**

**continue**

row **=** line**.**split**()**

row\_name **=** line**[**0**]**

**for** rate**,** col\_name **in** zip**(**row**[**1**:],** col\_names**):**

output**[**row\_name**][**col\_name**]** **=** rate

**return** output

"x" was generated with the following commands:

**>>>** execfile**(**r'C:\cygwin\home\alex\beta barrels\bbtm derivation\bbtm.py'**)**

**>>>** f **=** open**(**'parser test.txt'**,** 'r'**)**

**>>>** x **=** parse**(**f**)**

I used Mathematica to find . Surprisingly, it was pretty much uniform, that is, all the same number. When I changed 40 to a very small nuber, it was no longer uniform. It's as if 40 is an incredibly large evolutionary time, such that a position becomes equally likely to be any residue.

However, against this interpretation, if I put in a huge number like 100 000, all the numbers get very small (and still very uniform). The matrix does not appear to be a stochastic matrix, which makes me doubt it is really a transition probability matrix.

Also, I do not see why positions would become equally likely to be any residue after a very large time. Some states I would expect to be stronger attractors than others; I'd expect a non-uniform stationary state, in other words.

# August 7, 2012

The rate matrix **Q** has rows that add up to zero. Therefore, has rows that add up to 1.

Let be a vector that’s 20 1’s. Element i of is the sum of the ith row of **Q**.

, so:

Lemma:

By the lemma,

When I load **Q**all from “parser text.txt”, e^ **Q**all\*t gives about .998 for each row sum. But with t=1000, it’s down to about .86. 10,000, down to about .22. A million and it’s near zero. This is using Python with the “expm” function from scipy, with default settings. Scipy is version 0.11.0rc1, Python is version 2.73.

Why is there an error at all?

It could be errors in the approximation used by scipy. From scipy’s help:

>>> help(scipy.linalg.expm)

Help on function expm in module scipy.linalg.matfuncs:

expm(A, q=False)

Compute the matrix exponential using Pade approximation.

Parameters

----------

A : array, shape(M,M)

Matrix to be exponentiated

Returns

-------

expA : array, shape(M,M)

Matrix exponential of A

References

----------

N. J. Higham,

"The Scaling and Squaring Method for the Matrix Exponential Revisited",

SIAM. J. Matrix Anal. & Appl. 26, 1179 (2005).

Scipy implements matrix exponentiation wit a few different approximations. According to its help, expm2 uses an eigenvalue decomposition, and expm3 uses a taylor series. If the problem is in the apprixmation, then using two different approximations should give two different errors.

However, using the first two approximations with a power of 1000, the rowsums are the very close. With the taylor series method they seem to grow without bound.

It could be because the rows don’t really add up to 1.

>>> rowsum(r)

matrix([[ -2.00000000e-03],

[ 2.00000000e-03],

[ -1.00000000e-03],

[ 0.00000000e+00],

[ -1.00000000e-03],

[ -8.88178420e-16],

[ 1.77635684e-15],

[ 1.00000000e-03],

[ -1.00000000e-03],

[ -1.00000000e-03],

[ -4.44089210e-16],

[ -1.00000000e-03],

[ 1.00000000e-03],

[ -1.00000000e-03],

[ 1.00000000e-03],

[ -1.00000000e-03],

[ -1.00000000e-03],

[ 1.00000000e-03],

[ 1.00000000e-03],

[ 2.66453526e-15]])

So, when I was using mathematica and getting non-stochastic matrices with tiny elements, it’s probably sbecause I was taking powers that are too high.

# August 8, 2012

Using Python 2.7.2 today. Numpy version 1.6.1, Scipy version 0.11.0rc1.

I loaded **Q**all as "r", and created a new matrix, r\_fixed, by subtracting from each diagonal element the sum of the whole row. This way, the rowsums add up to zero.

**>>>** rowsums**(**r\_fixed**)**

matrix**([[** 0.00000000e+00**],**

**[** **-**7.10542736e-15**],**

**[** 0.00000000e+00**],**

**[** 0.00000000e+00**],**

**[** 0.00000000e+00**],**

**[** **-**8.88178420e-16**],**

**[** 1.77635684e-15**],**

**[** 0.00000000e+00**],**

**[** 0.00000000e+00**],**

**[** 1.66533454e-16**],**

**[** **-**4.44089210e-16**],**

**[** 7.10542736e-15**],**

**[** **-**1.77635684e-15**],**

**[** 7.10542736e-15**],**

**[** 0.00000000e+00**],**

**[** 0.00000000e+00**],**

**[** 0.00000000e+00**],**

**[** 0.00000000e+00**],**

**[** 2.84217094e-14**],**

**[** 2.66453526e-15**]])**

Up to t=100,000 at least, the rowsums of exp(r\_fixed \* t) are still "1". At t=106, they start being far enough from 1 for python to recognize them as " 0.99999999"

While the expm2 algorithm gives the same result, the expm3 algorithm, supposedly an implementation of the taylor series, still seems to increase without bound with t and with the number of terms in the series that are calculated.

So, the errors in the row sums were due to rounding errors in the original matrix. The effects of these rounding errors become more and more extreme as t goes up.

Here's the code I used to for today's tests above, and for yesterday's tests. It was modified as I went along, but I expect all the results to still be replicable with it.

**import** scipy**.**linalg

**import** itertools

**import** math

**import** os

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

os**.**chdir**(**r'C:\cygwin\home\alex\beta barrels\bbtm derivation'**)**

**def** parse**(**mat\_file**):**

# Take in an open file in the format in which matrices are presented

# in "Patterns of Amino Acid Substitutions..." Jimenez-Morales, Jie Liang

# and return a dictionary that can be used like q['A']['T'] to find

# the entry in row A, column T.

found\_resns **=** False

**for** line **in** mat\_file**:**

# Ignore comments

**if** line**[**0**]** **==** '#'**:**

**continue**

# Find which resn corresponds to which column number

**if** **not** found\_resns**:**

found\_resns **=** True

col\_names **=** line**.**split**()**

# Check to make sure they're all there

**for** resn **in** **[**'C'**,** 'N'**,** 'H'**,** 'D'**,** 'S'**,** 'Q'**,** 'K'**,** 'M'**,** 'P'**,**

'T'**,** 'F'**,** 'A'**,** 'G'**,** 'I'**,** 'L'**,** 'R'**,** 'W'**,** 'E'**,**

'Y'**,** 'V'**]:**

**try:**

**assert** resn **in** col\_names**,** "missing " **+** resn

**except** AssertionError**:**

**print(**line**)**

**raise**

# Make the matrix that will be returned

output **=** dict**([(**resn**,** **None)** **for** resn **in** col\_names**])**

**for** key**,** value **in** output**.**items**():**

output**[**key**]** **=** dict**([(**resn**,** **None)** **for** resn **in** col\_names**])**

**continue**

row **=** line**.**split**()**

row\_name **=** line**[**0**]**

**for** rate**,** col\_name **in** zip**(**row**[**1**:],** col\_names**):**

output**[**row\_name**][**col\_name**]** **=** float**(**rate**)**

**return** output

**def** to\_mat**(**m**,** order **=** 'None'**):**

# The parser returns a dictionary. This function turns one of

# those dictionaries into a matrix, with the elements in the

# specified order. "order" should be a list of one-letter

# residue names.

**if** order **is** **None:**

resns **=** m**.**keys**()**

**else:**

resns **=** order

mat\_as\_list **=** list**()**

**for** row\_name **in** resns**:**

mat\_as\_list**.**append**([**m**[**row\_name**][**col\_name**]** **for** col\_name **in** resns**])**

mat **=** scipy**.**matrix**(**mat\_as\_list**)**

**return** mat

**def** p**(**q**,**t**):**

mat **=** to\_mat**(**q**)**

p\_as\_mat **=** scipy**.**linalg**.**expm**(**t**\***mat**)**

output **=** dict**((**resn**,** **None)** **for** resn **in** resns**)**

**for** key **in** output**.**keys**():**

output**[**key**]** **=** dict**((**resn**,** **None)** **for** resn **in** resns**)**

numrows**,** numcols **=** p\_as\_mat**.**shape

**for** i**,** j **in** itertools**.**product**(**range**(**numrows**),** range**(**numcols**)):**

output**[**resns**[**i**]][**resns**[**j**]]** **=** p\_as\_mat**[**i**,**j**]**

**return** output

**def** rowsums**(**array**):**

# Return a vector such that element i of the vector is the sum of the

# elements in row i of a given matrix

**return** np**.**matrix**(**array**)** **\*** np**.**matrix**([[**1.**]** **for** i **in** range**(**array**.**shape**[**1**])])**

**def** test**(**power**,** subject**):**

# Return a vector of the sum of each row of e^(subject\*10^power)

# Used to test how these change as the factor by which the subject

# is multiplied increases

**return** rowsums**(**scipy**.**linalg**.**expm**(**subject**\***10**\*\***power**))**

**def** test2**(**power**,** subject**):**

# Same as test using scipy.linalg.expm2 instead of scipy.linalg.expm

# Same function, different algorithm

**return** rowsums**(**scipy**.**linalg**.**expm2**(**subject**\***10**\*\***power**))**

**def** test3**(**power**,** subject**,** q**=**20**):**

**return** rowsums**(**scipy**.**linalg**.**expm3**(**subject**\***10**\*\***power**,** q**=**q**))**

**with** open**(**'parser test.txt'**,** 'r'**)** **as** f**:**

r\_dict **=** parse**(**f**)**

published\_ordering**=** **[**'S'**,** 'T'**,** 'N'**,** 'Q'**,** 'D'**,** 'E'**,** 'R'**,** 'K'**,** 'H'**,** 'C'**,**

'P'**,** 'G'**,** 'W'**,** 'Y'**,** 'F'**,** 'V'**,** 'I'**,** 'L'**,** 'A'**,** 'M'**]**

r **=** to\_mat**(**r\_dict**,** order **=** published\_ordering**)**

# Make a new matrix whose rows actually sum to zero like they do in the

# unrounded matrix

r\_fixed **=** np**.**copy**(**r**)**

**for** rownum **in** range**(**20**):**

rowsum **=** 0

**for** colnum **in** range**(**20**):**

rowsum **+=** r\_fixed**[**rownum**,** colnum**]**

r\_fixed**[**rownum**,** rownum**]** **-=** rowsum

I made a folder "matrices" containing text files"bbtmall.txt", "bbtmin.txt", "bbtmout.txt", "qall.txt", "qin.txt", and "qout.txt", each containing the matrix in the filename. The contents of the text files were copied and pasted from the supplementary information of Jimenez-Morales and Liang "Patterns of amino acid substitutions..."

Errors attempting to use **Q**out. Turns out the matrix has two "D" rows, and no "K" row.  
These matrices appear to be symmetric:

**>>>** q**[**'L'**][**'S'**]**

0.585

**>>>** q**[**'S'**][**'L'**]**

0.585

**>>>** q**[**'R'**][**'G'**]**

0.293

**>>>** q**[**'G'**][**'R'**]**

0.293

The second D row appears to be identical to the K column:

**>>>** q**[**'S'**][**'K'**]**

0.141

**>>>** q**[**'D'**][**'S'**]**

0.141

**>>>** q**[**'R'**][**'K'**]**

0.546

**>>>** q**[**'D'**][**'R'**]**

0.546

And, the second D row is in the same spot that the K row is in qall.

It seems that the K row was mislabeled a D row in qout. I am remedying this in "qout.txt".

# August 9, 2012

Modified the above code slightly so that "r" is the qout matrix, loaded from "qout.txt". Wrote a script that would calculate the largest difference between an element of exp(r\*40) and .05. Then, had it do the same for r\_fixed, to rule out the possibility that the results are just due to the way the rows don't sum to zero.

My code, which uses the functions above:

**with** open**(**'matrices/qout.txt'**,** 'r'**)** **as** f**:**

r\_dict **=** parse**(**f**)**

published\_ordering**=** **[**'S'**,** 'T'**,** 'N'**,** 'Q'**,** 'D'**,** 'E'**,** 'R'**,** 'K'**,** 'H'**,** 'C'**,**

'P'**,** 'G'**,** 'W'**,** 'Y'**,** 'F'**,** 'V'**,** 'I'**,** 'L'**,** 'A'**,** 'M'**]**

r **=** to\_mat**(**r\_dict**,** order **=** published\_ordering**)**

# Make a new matrix whose rows actually sum to zero like they do in the

# unrounded matrix

r\_fixed **=** np**.**copy**(**r**)**

**for** rownum **in** range**(**20**):**

rowsum **=** 0

**for** colnum **in** range**(**20**):**

rowsum **+=** r\_fixed**[**rownum**,** colnum**]**

r\_fixed**[**rownum**,** rownum**]** **-=** rowsum

# Make a list of the elements of e^(r \* 40)

elem\_list **=** **[**scipy**.**linalg**.**expm**(**r**\***40**)[**i**,** j**]** \

**for** i**,** j **in** itertools**.**product**(**range**(**20**),** range**(**20**))]**

**print(**max**([**i **-** .05 **for** i **in** elem\_list**]))**

# Same thing for r\_fixed:

elem\_list **=** **[**scipy**.**linalg**.**expm**(**r\_fixed**\***40**)[**i**,** j**]** \

**for** i**,** j **in** itertools**.**product**(**range**(**20**),** range**(**20**))]**

**print(**max**([**i **-** .05 **for** i **in** elem\_list**]))**

And the output:

>>>

-0.000465113153873

3.76504383226e-14

So, the transition probability matrix approaches a steady state with a uniform distribution, it seems.

# August 15, 2012

Wrote this code to produce a scoring matrix from a transition probability matrix:

**def** scoring\_matrix**(**p**,** lam**,** pi**):**

'''p(A, G) becomes (1/lam) \* log(p(A,G)/pi(G))'''

b **=** dict**()**

**for** x **in** p**.**keys**():**

b**.**update**({**x**:** dict**((**x**,** **None)** **for** y **in** p**.**keys**())})**

**for** from\_**,** to **in** itertools**.**product**(**p**.**keys**(),** p**.**keys**()):**

b**[**from\_**][**to**]** **=** **(**1**/**lam**)** **\*** math**.**log**(**p**[**from\_**][**to**]** **/** pi**[**to**])**

I tested it on a matrix where the mapping to a scoring matrix was obvious (though it is not a valid transition probability matrix):

**>>>** test **=** dict**()**

**>>>** **for** i **in** **[**'A'**,** 'B'**]:**

test**.**update**({**i**:** dict**((**j**,** **None)** **for** j **in** **[**'A'**,** 'B'**])})**

**>>>** to\_mat**(**test**,** **[**'A'**,** 'B'**])**

matrix**([[None,** **None],**

**[None,** **None]],** dtype**=**object**)**

**>>>** test**[**'A'**][**'A'**]** **=** 10

**>>>** test**[**'A'**][**'B'**]** **=** 100

**>>>** test**[**'B'**][**'A'**]** **=** 1000

**>>>** test**[**'B'**][**'B'**]** **=** 10000

**>>>** to\_mat**(**test**,** **[**'A'**,** 'B'**])**

matrix**([[** 10**,** 100**],**

**[** 1000**,** 10000**]])**

**>>>** pi **=** **{**'A'**:** 10**,** 'B'**:** 1**}**

**>>>** lam **=** 1**/**math**.**log10**(**math**.**e**)**

**>>>** b **=** scoring\_matrix**(**test**,** lam**,** pi**)**

**>>>** to\_mat**(**b**,** **[**'A'**,** 'B'**])**

matrix**([[** 0.**,** 2.**],**

**[** 2.**,** 4.**]])**

**>>>** b

**{**'A'**:** **{**'A'**:** 0.0**,** 'B'**:** 1.9999999999999998**},** 'B'**:** **{**'A'**:** 1.9999999999999998**,** 'B'**:** 3.9999999999999996**}}**