Substitution Inferences using Mathematical Biology Approach (SIMBA)

by Milton Simbarashe Kambarami

in fulfilment of Master of Philosophy (MPhil) in Bioinformatics under Faculty of Medicine and Health Sciences, University of Zimbabwe.

Substitution Inference using Mathematical Biology Approach (SIMBA) is a package which uses a mathematical approach based on Nei and Gojobori, 1986 to calculate numbers of synonynmous and non-synonymous substitution and further modified to calculate selection pressure by comparing values of synonymous and non-synonymous substitution obtained.

It is worthnoting that different approaches usually reach close values especially when working with similar codons. The SIMBA approach like most evolutionary analysis approaches treats each codon as an independent unit undergoing selection without effects of neighbouring codons or homologous codons in other sequences. When working with too divergent sequences the obtained values might be biased, however because this is an approach to be useful when working with Spike genes of SARS CoV 2, the bias is minimised.

My motivation for developing this package is the unavailability of **specific** tools to analyse the Spike gene of Coronaviruses and the computational expensiveness of available tools when trying to generalise analysis of all sequences, convergent or divergent. SIMBA is specific for the spike gene analysis and uses a very fast lightweight mathematical approach to find substitution inferences. However, analysis of other genes can be done by tweaking certain parts of the code which the author can freely do upon request.

NB for the sake of shortness of code, I used the word codon as short for triplet code in this package.

A number of approaches used to ensure faster inferences include:

1. Use a fast Nei and Gojobori, 1986 mathematical approach
2. Assumption codons are independent evolutionary units.
3. Exclusion of weighting since its going to be used on very similar amino acids.

Spike Gene Cutter

So far there is no simple process of cutting and cleaning Spike gene sequences from SARS-CoV-2 whole genome sequences. However, the user is supposed to use alignment tool of choice after the spike gene has been extracted. User can cut their genes using this block of code.

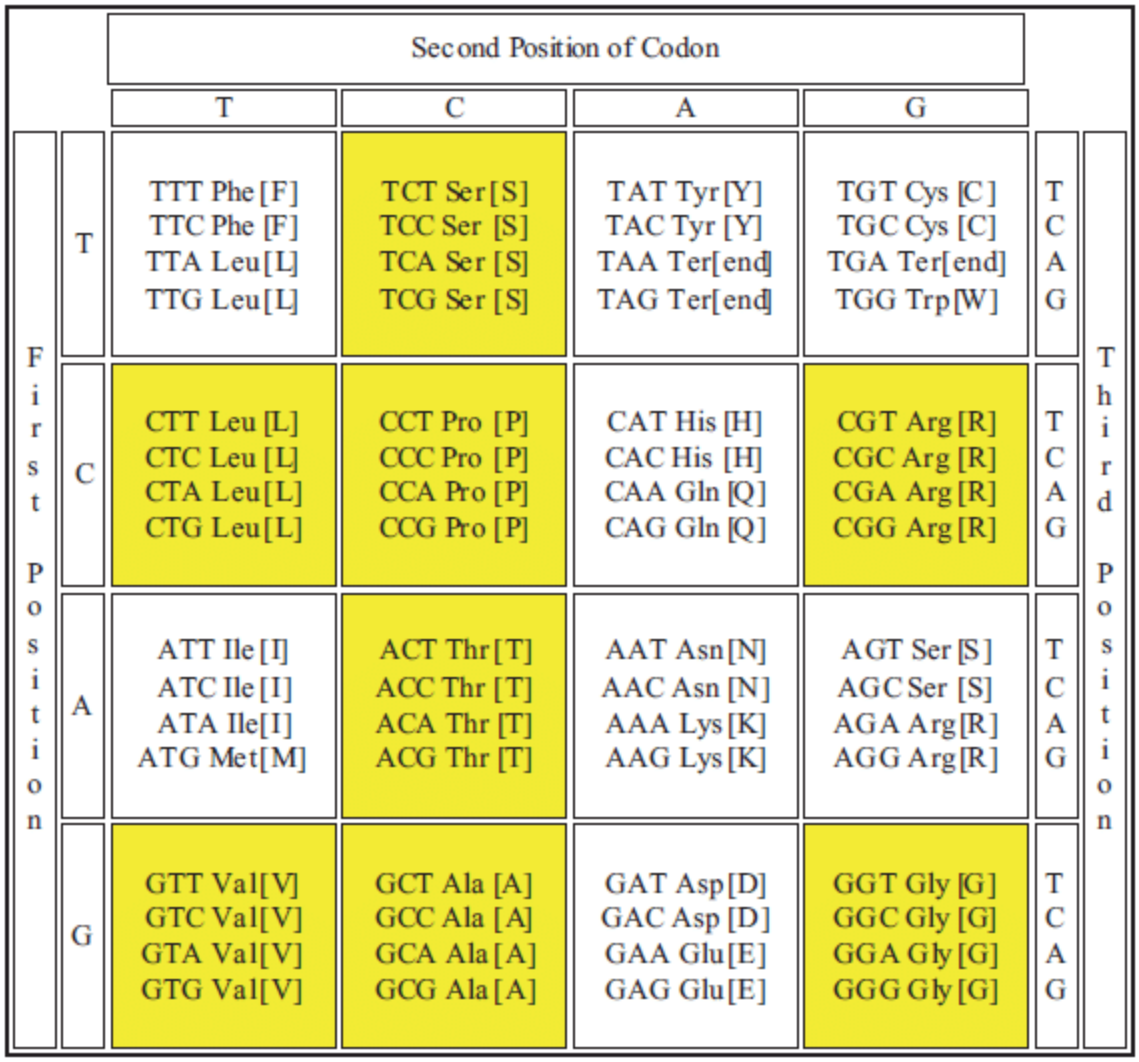
In [ ]:
```
def spike_gene_cutter_cleaner(input_file, file_format):

    # from Bio import SeqIO    # We use Biopython's SeqIO parser to
```

```
load our sequences
    import sys

    # The  next section loads the SARS-CoV-2 sequences from file.

    input_file = input("Enter full 'input' filename and filetype eg
input_file.fasta")
    file_format = input("Enter 'file format' eg fasta")

    # Extraction of the approximate S gene
    spike_genes = []        # All extracted spike genes will be stored
in this list

    for seq_record in SeqIO.parse(input_file, file_format):
        spike_cut = seq_record[21500:25500]
        spike_genes.append(spike_cut)


    spike_file = SeqIO.write(spike_genes,
'Spike_gene_file.fasta','fasta')

    # Next part removes spike genes with a number of Unidentified
nucleotides more than 2

    clean_spike_genes = [] # This list will store cleaned spike genes

    for spike_record in SeqIO.parse('Spike_gene_file.fasta','fasta'):
        # first its wiser to use a string object of the sequence for
counting additionally all letters should be ensured to be in upper case

        string_spike = (str(spike_record.seq)).upper()

        if string_spike.count('N') > 2:              #Unidentified
nucleotides more than 2
            if spike_record.seq not in clean_spike_genes:  # Removing
repeat sequences
                clean_spike_genes.append(spike_record)      #spike
record is appended instead of string_spike so it does not lose Sequence
object attributes
```

```
print('Please check current folder for the Spike gene sequences which
were extracted')


# In case the user prefers to see extracted sequences
print_sample = input('Do you want to view snips of extracted sequences:
Y/N ')

if print_sample.upper() == 'Y':
    for spike in SeqIO.parse('Spike_gene_file.fasta','fasta'):
            print(f'Length of approximate spike_gene = {len(spike)}')
            print(f'Representative spike sequence = {repr(spike.seq)}')
            print(f'Sequence id = {spike.id}')

elif print_sample.upper() == 'N':
        pass

else:
    print("Please enter 'Y' for 'Yes' and 'N' for 'No'")

# User can use alignment tool of choice and it is recommended to align
against the GISAID Ref Seq for more accurate
# downstream processes
```

The Triplet Code Table

Synonymous substitutions maybe used as a molecular clock for dating the evolutionary time of closely related species.



First Position	Second Position of Codon: T	Second Position of Codon: C	Second Position of Codon: A	Second Position of Codon: G	Third Position
T	TTT Phe [F] TTC Phe [F] TTA Leu [L] TTG Leu [L]	TCT Ser [S] TCC Ser [S] TCA Ser [S] TCG Ser [S]	TAT Tyr [Y] TAC Tyr [Y] TAA Ter [end] TAG Ter [end]	TGT Cys [C] TGC Cys [C] TGA Ter [end] TGG Trp [W]	T C A G
C	CTT Leu [L] CTC Leu [L] CTA Leu [L] CTG Leu [L]	CCT Pro [P] CCC Pro [P] CCA Pro [P] CCG Pro [P]	CAT His [H] CAC His [H] CAA Gln [Q] CAG Gln [Q]	CGT Arg [R] CGC Arg [R] CGA Arg [R] CGG Arg [R]	T C A G
A	ATT Ile [I] ATC Ile [I] ATA Ile [I] ATG Met [M]	ACT Thr [T] ACC Thr [T] ACA Thr [T] ACG Thr [T]	AAT Asn [N] AAC Asn [N] AAA Lys [K] AAG Lys [K]	AGT Ser [S] AGC Ser [S] AGA Arg [R] AGG Arg [R]	T C A G
G	GTT Val [V] GTC Val [V] GTA Val [V] GTG Val [V]	GCT Ala [A] GCC Ala [A] GCA Ala [A] GCG Ala [A]	GAT Asp [D] GAC Asp [D] GAA Glu [E] GAG Glu [E]	GGT Gly [G] GGC Gly [G] GGA Gly [G] GGG Gly [G]	T C A G

In [ ]:

```
from Bio import AlignIO

aligned_sequences_filename = input('Enter filename including format eg
aligned_spike.fasta: ')     #make sure the alignment file is in in the
same

#directory as SIMBA else use full filepath
aligned_sequences_format = input('Enter file format eg fasta: ')

aligned_sequences = AlignIO.read(aligned_sequences_filename,
aligned_sequences_format)

aligned_sequences = ()

def translator(codon):

    '''Triplet codes, (codon for simplicity) that code for specific
amino acids based on the DNA triplet code table'''
```

```
    for codon in aligned_sequences:
        # Gene sequence is a fasta file from a codon-aligned sequences,
loaded using AlignIO of Biopython.
        # Reminder that all loaded DNA sequences must be in upper case
and in string format
        if codon == 'TTT' or 'TTC':
            amino_acid = 'F'

        elif codon == 'TTA' or 'TTG' or 'CTT' or 'CTC'or 'CTA' or
'CTG':
            amino_acid = 'L'

        elif codon == 'ATT' or 'ATC' or 'ATA':
            amino_acid = 'I'

        elif codon == 'ATG':
            amino_acid = 'M'

        elif codon == 'GTT' or 'GTC' or 'GTA' or 'GTG':
            amino_acid = 'V'

        elif codon == 'TCT' or 'TCC' or 'TCA' or 'TCG' or 'AGT' or
'AGC':
            amino_acid = 'S'

        elif codon == 'CCT' or 'CCC' or 'CCA' or 'CCG':
            amino_acid = 'P'

        elif codon == 'ACT' or 'ACC' or 'ACA' or 'ACG':
            amino_acid = 'T'

        elif codon == 'GCT' or 'GCC' or 'GCA' or 'GCG':
            amino_acid = 'A'

        elif codon == 'TAT' or 'TAC':
            amino_acid = 'Y'

        elif codon == 'TAA' or 'TAG' or 'TGA':
            amino_acid = '*'

        elif codon == 'CAT' or 'CAC':
```

```
            amino_acid = 'H'

        elif codon == 'CAA' or 'CAG':
            amino_acid = 'Q'

        elif codon == 'AAA' or 'AAG':
            amino_acid = 'K'

        elif codon == 'GAA' or 'GAG':
            amino_acid = 'E'

        elif codon == 'TGT' or 'TGC':
            amino_acid = 'C'

        elif codon == 'TGG':
            amino_acid = 'W'

        elif codon == 'CGT' or 'CGC' or 'CGA' or 'CGG' or 'AGA' or
'AGG':
            amino_acid = 'R'

        elif codon == 'GGT' or 'GGC' or 'GGA' or 'GGG':
            amino_acid = 'G'

        elif 'N' IS IN codon:
            amino_acid = '?'

        else:
            print('Please check if all bases are A, C, T or G')
```

Sum of synonymous substitution in a codon at each *i* th site

Genetic code table indicates that all substitutions at the second nucleotide positions of codons result in amino acid whereas a fraction of the nucleotide changes at the first and third positions are synonymous.

Under the assumption of equal nucleoties frequencies and random substitution, this fraction is ~5 % for the first position and ~72 % for the thrid position.

> f_i = fraction of synonymous changes at the *i* th position of a given codon (i = 1,2,3)
>
> *s* = sum of synonymous substitution at each site

> *n* = sum of non-synonymous substitution at each site

The *n* and *s* for this codon are then given by:

$$s = \sum_{i=1}^{3} f_i \ \backslash \ \text{and} \backslash \ \ \text{n=(3-s)} \tag{1}$$

> using **Leu** as an example,

$f_1 = \frac{1}{3}\text{A} \rightarrow \text{G}$

using genetic code table, there is 1 in 3 chances that a change is from T $\rightarrow$ C.

f_2 = 0,

$f_3 = \frac{1}{3}(A \rightarrow G)$ thus,

$S = \frac{1}{3} + 0 + \frac{1}{3} = \frac{2}{3}$

$n = \frac{2}{3} + \frac{3}{3} + \frac{2}{3} = \frac{7}{3}$

Approach of method:

1. Load aligned sequences.
2. Function which assigns each codon a codon_site number (psi_j) in each sequence
3. Compare each base at *i* th position for each codon at ψ_j where ψ_j = codon at site `j` by comparing all codons at ψ_j with ψ_j = 0
4. Find sum of synonymous substitution fractions (s) at each ψ_j and append to a sum of synonymous sites list
5. Find sum of non-synonymous substitution fraction (n) at each psi_j.

In [ ]:
```
#1. Load aligned sequences

from Bio import AlignIO

aligned_sequences_filename = input('Enter filename including format eg
aligned_spike.fasta: ')     #make sure the alignment file is in in the
same

#directory as SIMBA else use full filepath
aligned_sequences_format = input('Enter file format eg fasta: ')

aligned_sequences = AlignIO.read(aligned_sequences_filename,
aligned_sequences_format)
```

In [ ]:
```
# 2. Function which assigns each codon a codon_site number (psi_j) in
each sequence
```

```
def codon_splitter(aligned_sequences):
    codoned_sequences = []                    #this list holds all
sequences but in their codon form, each sequence is a list of codons
    for seq in aligned_sequences:
        codon_list = []                       #this list holds each
sequence as an array of codons.
        for i in range(0, len(seq), 3):
            codon_list.append(seq[i:i+3])
        return codon_list
    codoned_sequences.append(codon_list)

    # the next part seperates lists of sequences in codon form inside
the codon_sequences list
    for index,item in enumerate(codoned_sequences):
        codoned_sequences[index] = codoned_sequences[index]
[0].split(",")
        return codoned_sequences
```

In [ ]:

```
#3. Compare each base at _i_ th position for each codon at codon_j
where codon_j = codon at site `j`
#4. Find sum of synonymous substitution fraction and append to a sum of
codon synonymous substitution fractions list


list_of_codon_synonymous_substitutions_fractions = []      #list to
collect sum of codon synonymous substitutions_fractions  per site
                                                           #code must
be added which k

def sum_of_codon_synonymous_substitutions(codoned_sequences, lower=0,
upper=2):
    psi_j = [ys + [x] for x, ys in zip(codoned_sequences)]       #this
code selects items(i.e. codons) with the same index in codoned
sequences list lists

    for psi_j in codoned_sequences:
        ref_codon = psi_j[0]

        for i,nucleotide in enumerate(zip(psi_j, ref_codon)):
#Lower represents the lower bound in our sum and upper is the upper
bound. For python index, the first value would be0
```

```
                                                    #i is the index of
a base in a given codon_site
            if i == 0:
                if nucleotide != ref_codon:
                    fraction_of_change1 = 0.05            #Fraction of
change denotes the chances of a codon for coding another amino acid
when there is a mutation eg
                else:
                    fraction_of_change1= 0
#changing the first codon nucleotide results in 5% chance of change in
amino acid coded.

            elif i == 1:
                if nucleotide != ref_codon:
                    fraction_of_change2 = 1
                else:
                    fraction_of_change2 = 0

            elif i == 2:
                if nucleotide != ref_codon:
                    fraction_of_change3 = 0.72
                else:
                    fraction_of_change3 = 0

            sum_of_synonymous_fractions = sum(fraction_of_change1 +
fraction_of_change2 + fraction_of_change3)              #gives use the
sum of fraction of synonymous changes at ith position.

list_of_codon_synonymous_substitutions_fractions.append(sum_of_synonymous


            return sum_of_synonymous_fractions
```

In [ ]:

```
#5. Find sum of non-synonymous substitution fraction (n) at each
$psi_j$

def
sum_of_non_synonymous_codon_substitutions(sum_of_synonymous_fractions):
    sum_of_non_synonymous_fractions = 3 - sum_of_synonymous_fractions
# to find sum of non-synonymous fraction we subtract the sum of
```

```
    return sum_of_non_synonymous_fractions
#synonymous fraction from 3
```

Mean of synonymous sites (S) and non_synonymous sites (N)

For a DNA sequence of **r** codons, the total number of synonymous and non-synonymous sites at ψ_j is therefore given by:

$$S = \sum_{j=1}^{r} S_i \text{ and} \backslash \quad \text{N} = (3\text{r -S})$$

$$\text{where } s_i = \text{value of s for the i-th codon}$$

$$\text{j} = \text{position number of codon j in DNA sequence with r codons}$$

When two sequences are compared, the averages of **S** and **N** are used.

Therefore:

$$S = \sigma \text{ (list of codon synonymous substitutions fractions)}$$

$$\text{where } \sigma = \text{mean/average at } \psi_j \text{ obtained in s}$$

$$S = \frac{\text{sum of elements in list of codon synonymous substitutions fractions}}{\text{number of elements in list of codon synonymous substitutions fractions}}$$

In [ ]:

```
import statistics

# mean_number_synonymous_substitution_list = []
# mean_number_non_synonymous_substitution_list = []
#these lists will be converted to dict to keep their `j` and added into
the dataframe for visualisation.



def
mean_number_synonymous_substitution_at_psi_j(list_of_codon_synonymous_su

    mean_number_synonymous_substitution =
int(statistics.mean((list_of_codon_synonymous_substitutions_fractions)))

    #
mean_number_synonymous_substitution_list.append(mean_number_synonymous_su



def
mean_number_non_synonymous_substitution_at_psi_j(list_of_codon_synonymous
```

```
    r = len(list_of_codon_synonymous_substitutions_fractions)
    mean_number_non_synonymous_substitution = ((3*r) -
(mean_number_synonymous_substitution))
    #
mean_number_non_synonymous_substitution_list.append(mean_number_non_synon
```

Computing nucleotide differences between a pair of homologous sequences

To compute the number of synonymous and non synonymous nucleotide differences between a pair of homologous sequences, we compare the two sequences codon, we compare the two sequences codon by codon and count the number of synoymous and non-synonymous nucleotide differences for each pair of codon compared.

When there is only one nucleotide difference, we can immediately decide whether the substitution is synonymous or non-synonymous.

> eg, if the codon pairs compared are **GTT** (*Val*) and **GTA** (*Val*), there is one synonymous difference.

We denote S_d and n_d the number of synonymous and non-synoynymous differences per codon, respecitively. In the prent case, $S_d = 1$ and $n_d = 0$.

For example, in the comparison of **TTT** and **GTA**, the two pathways are as follows:

> *Pathway I*:
>
> > **TTT** (*Phe*) $\rightarrow$ **GTT** (*Val*) $\rightarrow$ **GTA** (*Val*)
>
> *Pathway II*:
>
> > **TTT** (*Phe*) $\rightarrow$ **TTA** (*Leu*) $\rightarrow$ **GTA** (*Val*)

Pathway I involves one synonymous and one non-synonymous substitution whereas Pathway II involves two non-synonymous substitutions.

We assume that pathway I and II occur with equal probability.

The s_d and n_d then become 0.5 and 1.5 repsectively.

When there are three nucleotide differences between the codons compared, there are six different possible pathwaysbetween the codons [3(x-1)] and in each pathway there are 3 mutation steps [x] where x = nucleotide differences.

It is now clear that the total number of synonymous and non-synonymous differences can be obtained by summing up these values over all codons i.e.

$$S_d = \sum_{j=1}^{r} s_{dj}$$

$$N_d = \sum_{j=1}^{r} n_{dj}$$

where $s_{dj}\backslash$ and $n_{dj}\backslash$ are $s_d\backslash$ and n_d for the j-th codon respectively and r is the number of codons compared

In [ ]:

```
import statistics

synonymous_differences_list = []
non_synonymous_differences_list = []

def selection_differences_number(psi_j):
    psi_j = [ys + [x] for x, ys in zip(codoned_sequences)]       #this
code selects items(i.e. codons) with the same index in codoned
sequences list lists

    for psi_j in codoned_sequences:
        ref_codon = psi_j[0]

        for i,nucleotide in enumerate(zip(psi_j, ref_codon)):
            if i == 0:
                if nucleotide == ref_codon:
                    synonymous_difference = 0
                    non_synonymous_difference = 0

                elif nucleotide != ref_codon:
                    if translator(nucleotide) == translator(ref_codon):
                        synonymous_difference = 1

synonymous_differences_list.append(synonymous_difference)

                    elif translator(nucleotide) !=
translator(ref_codon):
                        non_synonymous_difference = 1

non_synonymous_differences_list.append(non_synonymous_difference)

            if i == 1:
                if nucleotide == ref_codon:
                    synonymous_difference = 0
```

```
                    non_synonymous_difference = 0

                elif nucleotide != ref_codon:
                    if translator(nucleotide) == translator(ref_codon):
                        synonymous_difference = 1

synonymous_differences_list.append(synonymous_difference)

                    elif translator(nucleotide) !=
translator(ref_codon):
                        non_synonymous_difference = 1

non_synonymous_differences_list.append(non_synonymous_difference)

            elif i == 2:
                if nucleotide == ref_codon:
                    synonymous_difference = 0
                    non_synonymous_difference = 0

                elif nucleotide != ref_codon:
                    if translator(nucleotide) == translator(ref_codon):
                        synonymous_difference = 1

synonymous_differences_list.append(synonymous_difference)

                    elif translator(nucleotide) !=
translator(ref_codon):
                        non_synonymous_difference = 1

non_synonymous_differences_list.append(non_synonymous_difference)

mean_number_of_synonymous_differences =
statistics.mean(int(synonymous_differences_list))
mean_number_of_non_synonymous_differences =
statistics.mean(int(non_synonymous_differences_list))
```

Estimating the proportion of synonymous and non-synonymous differences

We can then therefore, estimate the proportion of synonymous (p_s) and non-synonymous(p_n) differences by the following equations:

$$p_s = S_d/S \tag{2}$$
$$p_n = N_d/N \tag{3}$$

To estimate the number of synonymous substitution (d_s) and non-synonymous substitutions (d_N) per site, the following formula developed by *Jukes and Cantor (1969)* is used:

$$d = -\frac{3}{4}log_e(1 - \frac{4}{3}p) \tag{4}$$
$$\text{where p is either } p_S \text{ or } p_N \tag{5}$$

THis method gives only approximate estimates of d_S and d_N, and is very accurate for more similar sequences.

In [ ]:

```
# Calculation of p
def
proportion_of_synonymous_diff(mean_number_of_synonymous_differences,
mean_number_of_synonymous_substitution):
    proportion_of_synonymous_diff_val =
mean_number_of_synonymous_differences /
mean_number_of_synonymous_substitution
    return proportion_of_synonymous_diff_val

def
proportion_of_non_synonymous_diff(mean_number_of_non_synonymous_differenc
 mean_number_non_synonymous_substitution):
    proportion_of_non_synonymous_diff_val =
mean_number_of_non_synonymous_differences /
mean_number_non_synonymous_substitution
    return proportion_of_non_synonymous_diff_val

# Calculation of d
import math
def
number_of_synonymous_substitution_per_site(proportion_of_synonymous_diff)

    jukes_cantor_synonymous = 1 - ((4/3)*proportion_of_synonymous_diff)
    number_of_synonymous_substitution_per_site_val = (-3/4)*
(math.log(jukes_cantor_synonymous))
    return number_of_synonymous_substitution_per_site_val


def
number_of_non_synonymous_substitution_per_site(proportion_of_non_synonym
```

```
    jukes_cantor_non_synonymous = 1 -
((4/3)*proportion_of_non_synonymous_diff)
    number_of_non_synonymous_substitution_per_site_val = (-3/4)*
(math.log(jukes_cantor_non_synonymous))
    return number_of_non_synonymous_substitution_per_site_val
```

Substitution rate per site (ω)

Substitution rate per site,(ω_j) can be calculated by a ration of non-synonymous substitution : synonymous substitution

$$\omega_j = \frac{number\ of\ synonymous\ substitution\ per\ site}{number\ of\ non-synonymous\ substitution\ per\ site}$$

where ω_j = substitution rate per site

In [ ]:
```
omega_j =
number_of_synonymous_substitution_per_site_val/number_of_non_synonymous_s
```