User manual for pyvolve v1.0

Stephanie J. Spielman

Contents

1 Introduction

Pyvolve (pronouced "pie-volve") is an open-source python module for simulating genetic data along a phylogeny according to Markov models of sequence evolution, according to standard methods [14]. The module is available for download on github (and see here for API documentation). Note that pyvolve has several dependencies, including BioPython, NumPy, and SciPy. These modules must be properly installed and in your python path for pyvolve to work properly. Please file any and all bug reports on the github repository Issues section.

Pyvolve is written such that it can be seamlessly integrated into your python pipelines without having to interface with external software platforms. However, please note that for extremely large (e.g. >1000 taxa) and/or extremely heterogenous simulations (e.g. where each site evolves according to a unique evolutionary model), pyvolve may be quite slow and thus may take several minutes to run. Faster sequence simulators you may find useful include (but are certainly not limited to!) Indelible [1] and indel-Seq-Gen [11].

Pyvolve supports a variety of evolutionary models, including the following:

- Nucleotide Models

  - Generalized time-reversible model [12] and all nested variants
- Amino-acid exchangeability models
  - JTT [5], WAG [13], and LG [9]
- Codon models
  - Mechanistic (dN/dS) models (MG-style [10] and GY-style [2])
  - Empirical codon model [8]
- Mutation-selection models
  - Halpern-Bruno model [3], implemented for codons and nucleotides

Note that it is also possible to specify custom matrices (detailed in section 8.2 below). Both site-wise and temporal (branch) heterogeneity are supported. A detailed and highly-recommended overview of Markov process evolutionary models, for DNA, protein, and codons, is available in the book *Computational Molecular Evolution*, by Ziheng Yang [14].

2 Basic Usage

Similar to other simulation platforms, pyvolve evolves sequences in groups of **partitions**. Each partition has an associated size and model (or set of models, if branch heterogeneity is desired). All partitions will evolve according to the same phylogeny; if you wish to have each partition evolve according to a distinct phylogeny, I recommend performing several simulations and then merging the resulting alignments in the post-processing stage.

The general framework for a simple simulation is given below. In order to simulate sequences, you must define the phylogeny along which sequences evolve as well as any evolutionary model(s) you'd like to use. Each evolutionary model has associated parameters which you can customize, as detailed in Section 4.

```
######### General pyvolve framework #########
#############################################

# Import the pyvolve module
import pyvolve

# Read in phylogeny along which pyvolve should simulate
my_tree = pyvolve.read_tree(file = "file_with_tree_for_simulating.tre")

# Define and construct evolutionary models
my_model = pyvolve.Model(<model_type>, <custom_model_parameters>)
my_model.construct_model()

# Define partitions
my_partition = pyvolve.Partition(models = my_model, size = 100)

# Evolve partitions with the callable Evolver() class
pyvolve.Evolver(tree = my_tree, partitions = my_partition)()
```

By default, sequences will be output to a fasta-formatted file called "simulated_alignment.fasta". Two additional tab-delimited files, called "site_rates.txt" and "site_rates_info.txt" are also output. These files provide useful information when heterogeneity (either site or branch) is implemented. The former file indicates to

which partition and rate category (if no rate heterogeneity specified, these values will all be 1) each site belongs, and the latter file provides more specific information about each rate category, in particular its associated partition, probability, and value.

To change the output file names for any of those files, provide the arguments **`ratefile`** ("site_rates.txt"), **`infofile`** ("site_rates_info.txt"), and/or **`seqfile`** ("simulated_alignment.fasta") when initializing an **`Evolver`** instance:

```
# Provide custom file names when initializing the Evolver instance
myevolver = pyvolve.Evolver(tree = my_tree, partitions = my_partition, ratefile = "
    custom_ratefile.txt", infofile = "custom_infofile.txt", seqfile = "custom_seqfile.fasta" )
myevolver() #evolve
```

To suppress the creation of any of these files, define the argument(s) as either **`None`** or **`False`**:

```
# Only output a sequence file (suppress the ratefile and infofile)
myevolver = pyvolve.Evolver(tree = my_tree, partitions = my_partition, ratefile = None, infofile
    = None)
myevolver() #evolve
```

The sequence file format can also be changed with the argument **`seqfmt`**. Note that pyvolve uses Biopython to write sequence files, so consult the Biopython AlignIO module documentation (or this nice wiki) for available formats.

```
# Save the sequence file as seqs.phy, in phylip format
myevolver = pyvolve.Evolver(tree = my_tree, partitions = my_partition, seqfile = "seqs.phy",
    seqfmt = "phylip")
myevolver() #evolve
```

<model_type> is the type of model matrix. <custom_model_parameters> is a *dictionary* of parameters for your chosen model. See below for available model types and associated parameter keys.

3 Defining phylogenies

Phylogenies may be specified either as a newick tree string or by providing the name of a file that contains the newick tree. To provide a phylogeny, use the **`read_tree`** function.

```
# Read phylogeny from file with the argument "file"
phylogeny = read_tree(file = "/path/to/tree/file.tre")

# Read phylogeny from string with the argument "tree"
phylogeny = read_tree(tree = "(t4:0.785,(t3:0.380,(t2:0.806,(t5:0.612,t1:0.660):0.762):0.921):0.
    207);")
```

To implement temporal (branch) heterogeneity, in which different branches on the phylogeny evolve according to different models, you will need to specify *model flags* at particular nodes in the newick tree. Model flags must be in the format **`_flagname_`** (i.e. with both a leading and a trailing underscore), and they should be placed after branch lengths or nodes (not after taxon names!). Note that model flags may be repeated throughout the tree, but the model associated with each model flag will always be the same. Once a model flag has been placed at a given node, all of that node's children will inherit that model. If a new model is specified in a child node, however, then this model will be applied downstream.

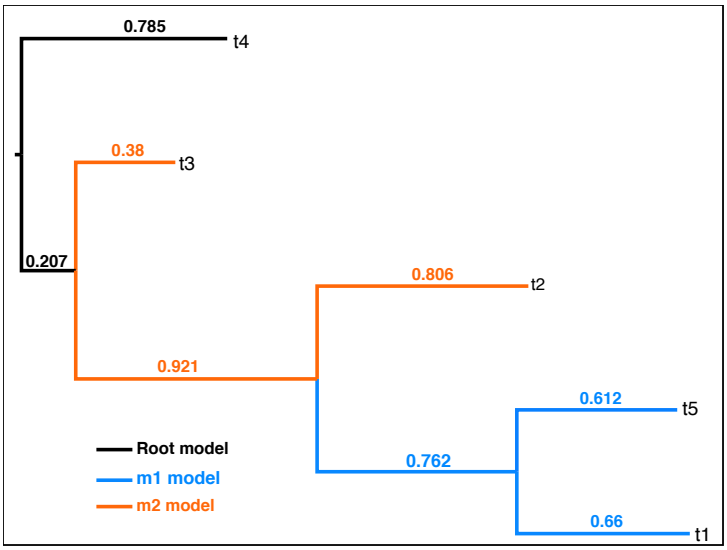

Figure 1: The newick tree with model flags given by `"(t4:0.785,(t3:0.380,(t2:0.806,(t5:0.612,t1:0.660):0.762_m1_):0.921)_m2_:0.207);"` indicates the model assignments shown.

For example, a tree specified as `(t4:0.785,(t3:0.380,(t2:0.806,(t5:0.612,t1:0.660):0.762_m1_):0.921)_m2_:0.207);` will be interpreted as in Figure 1. All branches on the tree which are not assigned a model flag will evolve according to the root model. For more details on evolving sequences with branch heterogeneity, see Section 6.

4 Defining Evolutionary Models

The evolutionary models built into pyvolve are outlined in the Introduction. All models used in simulation must be defined using the **`Model()`** class, the basic usage of which is detailed here.

4.1 Nucleotide Models

Nucleotide rate matrix elements, for the substitution from nucleotide i to j, are generally given by

$$q_{ij} = \mu_{ij}\pi_j \tag{1}$$

where μ_{ij} describes the rate of change from nucleotide i to j, and π_j represents the equilibrium frequency of the target nucleotide j. Note that mutation rates are symmetric, e.g. $\mu_{ij} = \mu_{ji}$.

By default, nucleotide pyvolve models use equal mutation rates and equal equilibrium frequencies (corresponding to the Jukes-Cantor model [6]). A basic model can be constructed with,

```
# Simple nucleotide model
nuc_model = pyvolve.Model("nucleotide")
nuc_model.construct()
```

To customize a nucleotide model, include a custom-parameters dictionary as a second argument to **`Model()`** with optional keys **`"mu"`** for custom mutation rates and **`"state_freqs"`** for custom equilibrium frequencies

(see Section 7 for details on frequency customization).

```
# Define mutation rates in a dictionary with keys giving the nucleotide pair
# Below, the rate from A to C is 0.5, and similarly C to A is 0.5
custom_mu = {'AC':0.5, 'AG':0.25, 'AT':1.23, 'CG':0.55, 'CT':1.22, 'GT':0.47}

# Define custom frequencies, in order A C G T. This can be a list or numpy array.
freqs = [0.1, 0.45, 0.3, 0.15]

# Construct nucleotide model with custom mutation rates and frequencies.
nuc_model = pyvolve.Model( "nucleotide", {'mu':custom_mu, 'state_freqs':freqs} )
nuc_model.construct()
```

Note that any undefined mutation rates will be set to 1. Further, mutation rates are symmetric; if you provide a rate for $A \rightarrow T$, it will automatically be applied as the rate $T \rightarrow A$.

As an alternate to **"mu"**, you can provide the key **"kappa"**, which corresponds to the transition:transversion ratio (e.g. for an HKY85 model [4]), in the custom-parameters dictionary. When kappa is specified, tranversion rates are set to 1, and transition rates are set to the provided value.

```
# Construct nucleotide model with transition-to-transversion bias, and default frequencies
nuc_model = pyvolve.Model( "nucleotide", {"kappa":2.75, "state_freqs":freqs} )
nuc_model.construct()
```

4.2 Amino-acid models

Amino-acid exchangeability matrix elements, for the substitution from amino acod i to j, are given by

$$q_{ij} = r_{ij}\pi_j \tag{2}$$

where r_{ij} is a symmetric matrix which describes the probability of changing from amino acid i to j, and π_j is the equilibrium frequency of the target amino acid j. The r_{ij} matrix corresponds to an empirically determined model, such as WAG [13] or LG [9].

By default, pyvolve assign equal equilibrium frequencies. A basic amino-acid model can be constructed with,

```
# Simple amino-acid model
aa_model = pyvolve.Model("WAG") # Here, WAG can be one of JTT, WAG, LG (case-insensitive)
aa_model.construct()
```

To customize an amino-acid model, specify the custom-parameters dictionary with the key **"state_freqs"** for custom equilibrium frequencies (see Section 7 for details on frequency customization). Note that amino-acid frequencies must be in the order A, C, D, E, ... Y. Further, to specify the *model's* default equilibrium frequencies, use the pyvolve EmpiricalModelFrequencies class:

```
# Define default WAG state frequencies
f = EmpiricalModelFrequencies("WAG") # model name is case-insensitive
freqs = f.construct_frequencies()

# Construct amino-acid model with WAG frequencies
aa_model = pyvolve.Model( "WAG", {"state_freqs":freqs} )
aa_model.construct()
```

4.3 Mechanistic (dN/dS) codon models

GY-style [2] matrix elements, for the substitution from codon i to j, are generally given by

$$q_{ij} = \begin{cases} \mu_{o_i t_j} \pi_j \alpha & \text{synonymous change} \\ \mu_{o_i t_j} \pi_j \beta & \text{nonsynonymous change} \\ 0 & \text{multiple nucleotide changes} \end{cases}, \tag{3}$$

where $\mu_{o_i t_j}$ is the mutation rate (e.g. for a change AAA to AAC, the corresponding mutation rate would be A $\rightarrow$ C), π_j is the frequency of the target *codon* j, α is the rate of synonymous change, and β is the rate of nonsynonymous change. In this framework, β/α corresponds to dN/dS.

MG-style [10] matrix elements, for the substitution from codon i to j, are generally given by

$$q_{ij} = \begin{cases} \mu_{o_i t_j} \pi_{t_j} \alpha & \text{synonymous change} \\ \mu_{o_i t_j} \pi_{t_j} \beta & \text{nonsynonymous change} \\ 0 & \text{multiple nucleotide changes} \end{cases}, \tag{4}$$

where $\mu_{o_i t_j}$ is the mutation rate, π_{t_j} is the frequency of the target *nucleotide* t_j (e.g. for a change AAA to AAC, the target nucleotide would be C), α is the rate of synonymous change, and β is the rate of nonsynonymous change. In this framework, β/α corresponds to dN/dS. Further, mutation rates are symmetric.

Codon models *require* that you specify a dN/dS rate ratio as a parameter in the **`params`** dictionary. There are several options for specifying this value:

- Specify a single parameter, **`"omega"`**. This option sets the synonymous rate to 1.
- Specify a single parameter, **`"beta"`**. This option sets the synonymous rate to 1.
- Specify a two parameters, **`"alpha"`** and **`"beta"`**. This option sets the synonymous rate to α and the nonsynonymous rate to β. Further, mutation rates are symmetric.

By default, pyvolve assigns equal mutation rates and equal equilibrium frequencies. Basic mechanistic codon models can be constructed with,

```
# Simple GY-style model (specify as GY94)
gy_model = pyvolve.Model("GY94", {'omega': 0.5})
gy_model.construct()

# Simple MG-style model (specify as MG94)
mg_model = pyvolve.Model("MG94", {'alpha': 1.04, 'beta': 0.67})
mg_model.construct()

# Specifying "codon" results in a *GY-style* model
codon_model = pyvolve.Model("codon", {'beta': 1.25})
codon_model.construct()
```

To customize a mechanistic codon model, include the optional keys **`"mu"`** for custom mutation rates and **`"state_freqs"`** for custom equilibrium frequencies (see Section 7 for details on frequency customization) in the custom-parameters dictionary. Note that codon frequencies must ordered alphabetically (AAA, AAC, AAG, ..., TTG, TTT) *without* stop codons.

```
# Define mutation rates in a dictionary with keys giving the nucleotide pair
# Below, the rate from A to C is 0.5, and similarly C to A is 0.5
custom_mu = {'AC':0.5, 'AG':0.25, 'AT':1.23, 'CG':0.55, 'CT':1.22, 'GT':0.47}
```

```
4
5  # Construct codon model with custom mutation rates
6  codon_model = pyvolve.Model( "codon", {'mu':custom_mu, 'omega':0.55} )
7  codon_model.construct()
```

Note that any undefined mutation rates will be set to 1. Further, mutation rates are symmetric; if you provide a rate for $A \rightarrow T$, it will automatically be applied as the rate $T \rightarrow A$.

As an alternate to **"mu"**, you can provide the key **"kappa"**, which corresponds to the transition:transversion ratio (e.g. for an HKY85 model [4]), in the custom-parameters dictionary. When kappa is specified, tranversion rates are set to 1, and transition rates are set to the provided value.

```
1  # Construct codon model with transition-to-transversion bias, and default frequencies
2  codon_model = pyvolve.Model( "codon", {"kappa":2.75, "alpha":0.89, "beta":0.95} )
3  codon_model.construct()
```

4.4 Mutation-selection models

Mutation-selection (MutSel) model [3] matrix elements, for the substitution from codon (or nucleotide) i to j, are generally given by

$$q_{ij} = \begin{cases} \mu_{ij} \frac{S_{ij}}{1-1/S_{ij}} & \text{single nucleotide change} \\ 0 & \text{multiple nucleotide changes} \end{cases}, \tag{5}$$

where μ_{ij} is the mutation rate, and where S_{ij} is the scaled selection coefficient. The scaled selection coefficient indicates the fitness difference between the target and source state, e.g. $fitness_j - fitness_i$. Mutation rates in MutSel models are *not* constrained to be symmetric (e.g. μ_{ij} need not be equal to μ_{ji}).

MutSel models are implemented both for codons and nucleotides, and they may be specified either with equilibrium frequencies or with fitness values. Note that equilibrium frequencies must sum to 1, but fitness values are not constrained in any way. (Note that the relationship between equilibrium frequencies and fitness values is given in refs. citepHB98,SpielmanWilke2015). pyvolve automatically determines whether you are evolving nucleotides or codons based on the provided vector of equilibrium frequencies or fitness values; a length of 4 indicates nucleotides, and a length of 61 indicates codons. Note that, if you are constructing a codon MutSel model based on *fitness* values, you can alternatively specify a vector of 20 fitness values, indicating amino-acid fitnesses (in the order A,C, D, E, ... Y). These fitness values will be directly assigned to codons, such that all synonymous codons will have the same fitness.

Basic nucleotide MutSel models can be constructed with,

```
1  import numpy as np
2
3  # Simple nucleotide MutSel model constructed from frequencies, with default (equal) mutation
       rates
4  nuc_freqs = [0.1, 0.4, 0.3, 0.2]
5  mutsel_nuc_model_freqs = pyvolve.Model("MutSel", {'state_freqs': nuc_freqs})
6  mutsel_nuc_model_freqs.construct()
7
8  # Simple nucleotide MutSel model constructed from fitness values, with default (equal) mutation
       rates
9  nuc_fitness = [1.5, 0.88, -4.2, 1.3]
10 mutsel_nuc_model_fits = pyvolve.Model("MutSel", {'fitness': nuc_fitness})
11 mutsel_nuc_model_fits.construct()
```

Basic codon MutSel models can be constructed with,

```
import numpy as np

# Simple codon MutSel model constructed from frequencies, with default (equal) mutation rates
codon_freqs = np.repeat(1./61, 61) # constructs a vector of equal frequencies, as an example
mutsel_codon_model_freqs = pyvolve.Model("MutSel", {'freqs': codon_freqs})
mutsel_codon_model_freqs.construct()

# Simple codon MutSel model constructed from codon fitness values, with default (equal) mutation rates
codon_fitness = np.random.normal(size = 61) # constructs a vector of normally distributed codon fitness values, as an example
mutsel_codon_model_fits = pyvolve.Model("MutSel", {'freqs': codon_fitness})
mutsel_codon_model_fits.construct()

# Simple codon MutSel model constructed from *amino-acid* fitness values, with default (equal) mutation rates
aa_fitness = np.random.normal(size = 20) # constructs a vector of normally distributed amino-acid fitness values, as an example
mutsel_codon_model_fits2 = pyvolve.Model("MutSel", {'freqs': aa_fitness})
mutsel_codon_model_fits2.construct()
```

As usual, for both nucleotide and codon MutSel models, mutation rates can additionally be customized with the **"mu"** key in the **params** dictionary. Note that mutation rates in MutSel models do not need to be symmetric, but if you specify a rate for $A \rightarrow C$ and no rate for $C \rightarrow A$, then pyvolve will assume symmetry and assign $C \rightarrow A$ the same rate as $A \rightarrow C$. Again, the parameter **"kappa"** may instead be specified in the custom-parameters dictionary.

4.5 Empirical codon models

DO I EVEN WANT TO RETAIN THIS?

$$q_{ij} = \begin{cases} s^*_{ij}\pi_j\kappa(i,j)\alpha & \text{synonymous change} \\ s^*_{ij}\pi_j\kappa(i,j)\beta & \text{nonsynonymous change} \end{cases}, \tag{6}$$

Further, rest and unrest. The $\kappa(i,j)$ parameter is defined in their paper...

5 Site-wise heterogeneity

5.1 Nucleotide and amino-acid models

5.2 Codon models

5.3 Mutation-selection models

6 Temporal heterogeneity

7 Building a vector of equilibrium frequencies

By default, pyvolve assumes equal equilibrium frequencies (e.g. 0.25 for nucleotides, 0.05, for amino-acids, $1/61$ for codons). These conditions are not, however, very realistic. You can/should specify custom equilibrium frequencies for your simulations. pyvolve provides a convenient module to help you with this step, with several classes:

- EqualFrequencies (**default**)
  Sets frequencies as equal
- RandomFrequencies
  Computes (semi-)random frequencies
- CustomFrequencies
  Computes frequencies from a user-provided dictionary of frequencies
- ReadFrequencies
  Computes frequencies from a sequence or alignment file.
- EmpiricalModelFrequencies
  Sets frequencies to default values for a given *empirical* model

Basic usage of these classes:

```
# Define frequency object
f = EqualFrequencies("nuc") # or "amino" or "codon", depending on your simulation
frequencies = f.construct_frequencies() # returns a vector of equilibrium frequencies
```

Sometimes, it is useful to construct frequencies in a given alphabet and convert it to another. Such functionality is primarily useful when using the ReadFrequencies module, for instance when codon frequencies are desired from an amino-acid sequence file. This can be achieved with:

```
# Define frequency object
f = ReadFrequencies("amino", file = "my_aminoacid_file.fasta")
frequencies = f.construct_frequencies(type = "codon") # returns a vector of *codon* equilibrium
    frequencies
```

8 Special Features

8.1 Matrix scaling options

By convention, rate matrices are scaled such that the mean substitution rate is 1:

$$-\sum_{i=1} \pi_i q_{ii} = 1 \tag{7}$$

[2, 16]. Using this regime, branch lengths explicitly indicate the expected number of substitutions per unit (nucleotide, amino acid, or codon). By default, pyvolve will scale rate matrices according to equation 7, as this approach remains conventional in the field.

Unfortunately, this scaling approach can lead to some unexpected results. In particular, when multiple mechanistic codon (dN/dS) models are used, thus allowing for variable dN/dS values across sites, multiple rate matrices must be used – one matrix per dN/dS value. This scaling approach, then, would cause sites with $dN/dS = 0.05$ to experience the same average number of substitutions as sites with $dN/dS = 2.5$. From a biological perspective, this result is undesirable, as sites with low dN/dS values should evolve more slowly than sites with high dN/dS values.

To overcome this issue, pyvolve provides an option to scale matrices such that the mean *neutral* substitution rate is 1. For dN/dS codon models, this approach scales the matrix such that the mean number of substitutions when $dN/dS = 1$ is 1. For mutation-selection models (both nucleotide and codon), this approach scales the matrix such that the mean number substitution when all states have equal fitness is 1. This scaling option will have no effect on nucleotide or amino-acid models.

To invoke the neutral scaling, provide a third argument "neutral" when initializing a model:

```
# Construct a codon model with neutral-scaling
m = Model("GY94", {"omega":0.5}, "neutral") # Indicate neutral scaling with a third argument, "
    neutral"
m.construct_model()
```

We urge caution when using this scaling approach, as most phylogenetic inference softwares and modeling frameworks (including HyPhy [7] and PAML [15]), scale matrices according to equation 7, and thus inferences on data simulated with neutral scaling may be confounded.

8.2 Using custom rate matrices

This is my first python example:

```
from pyvolve import *

# Read in a newick tree
t = read_tree(file = "myfile.tre")

# Construct state frequency vector. Optional!
f = EqualFrequencies("amino")
freqs = f.construct_frequencies(type = "codon")

# Build the evolutionary model
m = Model("GY94", {'state_freqs':freqs, 'omega':1.5, kappa:3.4}
```

```
m.construct_model()

# Initialize partitions
p = Partition(models = m, size = 100)

# Evolve, and call.
Evolver(partitions = p, tree = t, seqfile = "sequences.phy", seqfmt = "phylip")()
```

References

[1] W Fletcher and Z Yang. INDELible: A Flexible Simulator of Biological Sequence Evolution. *Mol Biol Evol*, 26(8):1879–1888, 2009.

[2] N Goldman and Z Yang. A codon-based model of nucleotide substitution for protein-coding DNA sequences. *Mol Biol Evol*, 11:725–736, 1994.

[3] AL Halpern and WJ Bruno. Evolutionary distances for protein-coding sequences: modeling site-specific residue frequencies. *Mol Biol Evol*, 15:910–917, 1998.

[4] M Hasegawa, H Kishino, and T Yano. Dating of human-ape splitting by a molecular clock of mitochondrial DNA. *J Mol Evol*, 22(2):160–174, 1985.

[5] DT Jones, WR Taylor, and JM Thornton. The rapid generation of mutation data matrices from protein sequences. *CABIOS*, 8:275–282, 1992.

[6] TH Jukes and CR Cantor. Evolution of protein molecules. In HN Munro, editor, *Mammalian protein metabolism*. Academic Press, New York, 1969.

[7] Sergei L. Kosakovsky Pond, Simon D. W. Frost, and Spencer V. Muse. HyPhy: hypothesis testing using phylogenies. *Bioinformatics*, 12:676–679, 2005.

[8] C. Kosiol, I. Holmes, and N. Goldman. An empirical codon model for protein sequence evolution. *Mol Biol Evol*, 24:1464 – 1479, 2007.

[9] SQ Le and O Gascuel. An improved general amino acid replacement matrix. *Mol Biol Evol*, 25:1307–1320, 2008.

[10] SV Muse and BS Gaut. A likelihood approach for comparing synonymous and nonsynonymous nucleotide substitution rates, with application to the chloroplast genome. *Mol Biol Evol*, 11:715–724, 1994.

[11] Cory L Strope, Stephen D Scott, and Etsuko N Moriyama. indel-Seq-Gen: a new protein family simulator incorporating domains, motifs, and indels. *Mol Biol Evol*, 24(3):640–649, 2007.

[12] S Tavare. Lines of descent and genealogical processes, and their applications in population genetics models. *Theor Popul Biol*, 26:119–164, 1984.

[13] Simon Whelan and Nick Goldman. A general empirical model of protein evolution derived from multiple protein families using a maximum likelihood approach. *Mol Biol Evol*, 18:691–699, 2001.

[14] Z. Yang. *Computational Molecular Evolution*. Oxford University Press, 2006.

[15] Z. Yang. PAML 4: Phylogenetic analysis by maximum likelihood. *Molecular Biology and Evolution*, 24:1586–1591, 2007.

[16] Ziheng Yang. Maximum likelihood phylogenetic estimation from DNA sequences with variable rates over sites: approximate methods. *J Mol Evol*, 39:306 – 314, 1994.