Pyvolve User Manual

Stephanie J. Spielman, PhD

Email: spielman@rowan.com

Contents

1 Introduction

Pyvolve is an open-source Python module for simulating genetic data along a phylogeny using Markov models of sequence evolution, according to standard methods [?]. Pyvolve is freely available under a FreeBSD license and is hosted on github: http://sjspielman.org/pyvolve/. Pyvolve has several dependencies, including BioPython, NumPy, and SciPy. These modules must be properly installed and in your Python path. 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 (>10,000 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 or more to run. Faster sequence simulators you may find useful are detailed in ref. [?], which gives an overview of various sequence simulation softwares (from 2012).

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

- Nucleotide Models
  - Generalized time-reversible model [?] and all nested variants
- Amino-acid exchangeability models
  - JTT [?], WAG [?], LG [?], AB [?], mtMam [?], mtREV24 [?], and DAYHOFF [?]
- Codon models
  - Mechanistic (dN/dS) models (MG-style [?] and GY-style [?])
  - Empirical codon model [?]
- Mutation-selection models
  - Halpern-Bruno model [?], implemented for codons and nucleotides

Both site- and branch- (temporal) heterogeneity are supported. A detailed and highly-recommended overview of Markov process evolutionary models, for DNA, amino acids, and codons, is available in the book *Computational Molecular Evolution*, by Ziheng Yang [?].

Although Pyvolve does not simulate insertions and deletions (indels), Pyvovle does include several novel options not available (to my knowledge) in other sequence simulation softwares. These options, detailed in Section **??**, include custom rate-matrix specification, novel matrix-scaling approaches, and branch length perturbations.

2 Installation

Pyvolve may be downloaded and installed using `pip` or `easy_install`. Source code is available from https://github.com/sjspielman/pyvolve/releases.

3 Citation

If you use Pyvolve, or code derived from Pyvolve, please cite us:

Spielman, SJ and Wilke, CO. 2015. Pyvolve: A flexible Python module for simulating sequences along phylogenies. **PLOS ONE**. 10(9): e0139047.

4 Basic Usage

Similar to other simulation platforms, Pyvolve evolves sequences in groups of **partitions** (see, for instance, the Indelible simulation platform [?]). Each partition has an associated size and model (or set of models, if branch heterogeneity is desired). Note that all partitions will evolve according to the same phylogeny[1].

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, and assign model(s) to partition(s). Each evolutionary model has associated parameters which you can customize, as detailed in Section 6.

```
######### 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 evolutionary model(s) with the Model class
my_model = pyvolve.Model(<model_type>, <custom_model_parameters>)

# Define partition(s) with the Partition class
my_partition = pyvolve.Partition(models = my_model, size = 100)

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

Each of these steps is explained below, in detail with several examples. For additional information, consult the API documentation, at http://sjspielman.org/pyvolve. Further, all functions and classes in Pyvolve have highly descriptive docstrings, which can be accessed with Python's `help()` function.

5 Defining phylogenies

Phylogenies must be specified as newick strings (see this wikipedia page for details) *with branch lengths*. Pyvolve reads phylogenies using the function `read_tree`, either from a provided file name or directly from a string:

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

# Read phylogeny from string with the keyword argument "tree"
phylogeny = pyvolve.read_tree(tree = "(t4:0.785,(t3:0.380,(t2:0.806,(t5:0.612,t1:0.660
    ):0.762):0.921):0.207);")
```

[1]If you wish to have different partitions evolve according to distinct phylogenies, I recommend performing several simulations and then merging the resulting alignments in the post-processing stage.

To implement branch (temporal) 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, as detailed in Section 11.

Further, to assess that a phylogeny has been parsed properly (or to determine the automatically-assigned names of internal nodes), use the `print_tree` function:

```
# Read phylogeny from string
phylogeny = pyvolve.read_tree(tree = "(t4:0.785,(t3:0.380,(t2:0.806,(t5:0.612,t1:0.660
    ):0.762):0.921):0.207);")

# Print the parsed phylogeny
pyvolve.print_tree(phylogeny)
## Output from the above statement: node_name  branch_length  model_flag
'''
>>>     t4 0.785 None
>>>     internalNode3 0.207 None
>>>         t3 0.38 None
>>>         internalNode2 0.921 None
>>>              t2 0.806 None
>>>              internalNode1 0.762 None
>>>                  t5 0.612 None
>>>                  t1 0.66 None
'''
```

In the above output, tabs represent nested hierarchies in the phylogeny. Each line shows the node name (either a tip name, "root", or an internal node), the branch length leading to that node, and the model flag associated with that node. This final value will be `None` if model flags are not provided in the phylogeny. Again, note that model flags are only required in cases of branch heterogeneity (see Section 11).

It is also possible to provide a phylogeny with named internal nodes. Any internal nodes without provided names will be automatically assigned.

```
# Read phylogeny with some internal node names (myname1, myname2)
phylogeny = pyvolve.read_tree(tree = "(t4:0.785,(t3:0.380,(t2:0.806,(t5:0.612,t1:0.660
    )myname1:0.762)myname2:0.921):0.207);")

# Print the parsed phylogeny
pyvolve.print_tree(phylogeny)
## Output from the above statement: node_name  branch_length  model_flag
'''
>>>     t4 0.785 None
>>>     internalNode1 0.207 None
>>>         t3 0.38 None
>>>         myname2 0.921 None
>>>              t2 0.806 None
>>>              myname1 0.762 None
>>>                  t5 0.612 None
>>>                  t1 0.66 None
'''
```

You can also rescale, as desired, branch lengths on your input phylogeny using the keyword argument `scale_tree` in the `read_tree` function. This argument takes a numeric value and multiplies **all** branch lengths in your tree by this scalar:

```
    t = pyvolve.read_tree(file = "name_of_file.tre", scale_tree = 2.5) # Multiply all
        branch lengths by 2.5
```

This argument is useful for changing the overall tree length, hence increasing or decreasing the expected number of substitutions over the course of simulation.

6 Defining Evolutionary Models

The evolutionary models built into Pyvolve are outlined in Table 1 of this manual. Pyvolve uses `Model` objects to store evolutionary models:

```
# Basic framework for defining a Model object (second argument optional)
my_model = Model(<model_type>, <custom_model_parameters_dictionary>)
```

A single argument, `<model_type>`, is required when defining a `Model` object. Available model types are shown in Table 1. Each model type has various associated parameters, which can be customized via the second optional argument to `Model`, written above as `<custom_model_parameters_dictionary>`. This argument should be a dictionary of parameters to customize, and each modeling framework has particular keys which can be included in this dictionary. Available model types and associated customizable parameters are shown in Table 1 and detailed in the subsections below.

Note that there are additional optional keyword arguments which may be passed to `Model`, including arguments pertaining to site-rate heterogeneity (see Section 10).

Modeling framework	Pyvolve model type(s)	Optional parameters (`"key"`)
Nucleotide models	`"nucleotide"`	• Equilibrium frequencies (`"state_freqs"`) • Mutation rates (`"mu"` or `"kappa"`)
Empirical amino-acid models	`"JTT"`, `"WAG"`, `"LG"`, `"AB"`, `"DAYHOFF"`, `"MTMAM"`, or `"MTREV24"`	• Equilibrium frequencies (`"state_freqs"`)
Mechanistic (dN/dS) codon models	`"GY"`, `"MG"`, or `"codon"`	• Equilibrium frequencies (`"state_freqs"`) • Mutation rates (`"mu"` or `"kappa"`) • dN/dS (`"alpha"`, `"beta"`, and/or `"omega"`)
Mutation-selection models	`"MutSel"`	• Equilibrium frequencies (`"state_freqs"`) OR fitness values (`"fitness"`) • Mutation rates (`"mu"` or `"kappa"`)
Empirical codon model (ECM)	`"ECMrest"`, `"ECMunrest"`, or `"ECM"`	• Equilibrium frequencies (`"state_freqs"`) • Transition-tranversion bias(es) (`"k_ti"` and/or `"k_tv"`) • $dN/dS^{\dagger}$ (`"alpha"`, `"beta"`, and/or `"omega"`)

Table 1. Accepted model types in Pyvolve with associated customizable parameters. Names given in the column "Pyvolve model type(s)" should be specified as the first argument to `Model` as strings (case-insensitive). Customizable parameters indicated in the column "Optional parameters" should be specified as keys in the custom model-parameters dictionary, the second argument using when defining a `Model` object.
†Note that the interpretation of this dN/dS value is different from the usual interpretation.

Subsections below explain each modeling framework in detail, with examples of parameter customizations.

6.1 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 (i.e. mutation rate), 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 models have equal equilibrium frequencies and equal mutation rates. A basic model can be constructed with,

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

To customize a nucleotide model, provide a custom-parameters dictionary with optional keys `"state_freqs"` for custom equilibrium frequencies and `"mu"` for custom mutation rates (see Section 6.7 for details on frequency customization and Section 6.8 for details on mutation rate customization).

```python
# 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} )
```

As nucleotide model mutation rates are symmetric, if you provide a rate for $A \to T$ (key `"AT"`), it will automatically be applied as the rate for $T \to A$. Any unspecified mutation rate pairs will have a value of 1.

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

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

6.2 Amino-acid models

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

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

where r_{ij} is a symmetric matrix that 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 [?] or LG [?].

By default, Pyvolve assigns the *default model* equilibrium frequencies for empirical models. These frequencies correspond to those published with each respective model's original paper. 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, DAYHOFF, MTMAM
    , MTREV24 (case-insensitive)
```

To customize an amino-acid model, provide a custom-parameters dictionary with the key `"state_freqs"` for custom equilibrium frequencies (see Section 6.7 for details on frequency customization). Note that amino-acid frequencies must be in the order A, C, D, E, ... Y.

6.3 Mechanistic (dN/dS) codon models

GY-style [?] 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 $\to$ C), π_j is the frequency of the target *codon* j, α is the rate of synonymous change (dS), and β is the rate of nonsynonymous change (dN).

MG-style [?] 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 (dS), and β is the rate of nonsynonymous change (dN).

Both GY-style and MG-style codon models use symmetric mutation rates. Codon models *require* that you provide a dN/dS rate ratio as a parameter in the custom-parameters dictionary. There are several ways to specify 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.

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

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

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

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

To customize a mechanistic codon model, provide a custom-parameters dictionary with optional keys `"state_freqs"` for custom equilibrium frequencies and `"mu"` for custom mutation rates (see Section 6.7 for details on frequency customization and Section 6.8 for details on mutation rate customization). 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}

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

Mechanistic codon model mutation rates are symmetric; if you provide a rate for $A \to T$ (key `"AT"`), it will automatically be applied as the rate for $T \to A$. Any unspecified mutation rate pairs will have a value of 1.

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

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

Importantly, by default, mechanistic codon models are scaled so that the mean substitution rate per unit time is 1. Here, "mean substitution rate" includes *both* synonymous and nonsynonymous substitutions in its calculations. An alternative scaling scheme (note, which the author **strongly prefers**) would be to scale the matrix such that the mean *neutral* substitution rate per unit time is 1. To specify this approach, include the argument `neutral_scaling = True` when defining a Model:

```python
# Construct codon model with neutral scaling
codon_model = pyvolve.Model( "codon", {"omega":0.5}, neutral_scaling = True )
```

6.4 Mutation-selection models

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

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

where μ_{ij} is the mutation rate, and S_{ij} is the scaled selection coefficient. The scaled selection coefficient indicates the fitness difference between the target and source state, e.g. $fitness_j - fitness_i$. MutSel mutation rates are *not* constrained to be symmetric (e.g. μ_{ij} can be different from 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. (The relationship between equilibrium frequencies and fitness values for MutSel models is detailed in refs. [? ?]). 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,

```python
# Simple nucleotide MutSel model constructed from frequencies, with default (equal)
    mutation rates
nuc_freqs = [0.1, 0.4, 0.3, 0.2]
mutsel_nuc_model_freqs = pyvolve.Model("MutSel", {"state_freqs": nuc_freqs})

# Simple nucleotide MutSel model constructed from fitness values, with default (equal)
     mutation rates
nuc_fitness = [1.5, 0.88, -4.2, 1.3]
mutsel_nuc_model_fits = pyvolve.Model("MutSel", {"fitness": nuc_fitness})
```

Basic codon MutSel models can be constructed with,

```python
import numpy as np # imported for convenient example frequency/fitness generation

# Simple codon MutSel model constructed from frequencies, with default (equal)
    mutation rates
```

```python
codon_freqs = np.repeat(1./61, 61) # constructs a vector of equal frequencies, as an example
mutsel_codon_model_freqs = pyvolve.Model("MutSel", {"state_freqs": codon_freqs})

# 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", {"fitness": codon_fitness})

# 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", {"fitness": aa_fitness})
```

Mutation rates can be customized with either the `"mu"` or the `"kappa"` key in the custom-parameters dictionary. Note that mutation rates in MutSel models do not need to be symmetric. However, if you a rate for $A \to C$ (key `"AC"`) and no rate for $C \to A$ (key `"CA"`), then Pyvolve will assume symmetry and assign $C \to A$ the same rate as $A \to C$. If *neither* pair is provided (e.g. both "AC" and "CA" are not defined), then both will be given a rate of 1.

6.5 Empirical codon model

Matrix elements of the empirical codon model (ECM) [?] are given by,

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

where s_{ij} is the symmetric, empirical matrix indicating the probability of changing from codon i to j, π_j is the equilibrium frequency of the target codon j, $\kappa(i,j)$ is a mutational parameter indicating transition and/or tranversion bias, and α and β represent dS and dN, respectively. Importantly, because this model is empirically-derived, the parameters $\kappa(i,j)$, α, and β as used in ECM each represent the transition-tranversion bias, synonymous rate, and nonsynonymous rate, respectively, *relative* to the average level present in the PANDIT database [?], from which this model was constructed [2]. The parameter $\kappa(i,j)$ is described in depth in ref. [?], specifically in the second half of the Results section *Application of the ECM*.

Importantly, there are two versions of this model: **restricted** and **unrestricted**. The restricted model restricts instantaneous change to single-nucleotide only, whereas the unrestricted model also allows for double- and triple-nucleotide changes. Pyvolve refers to these models, respectively, as ECMrest and ECMunrest.

By default, Pyvolve assumes that $\kappa(i,j)$, α, and β all equal 1, and Pyvolve uses the *default empirical model* equilibrium frequencies. These frequencies correspond to those published in the original paper publishing ECM.

Basic ECM can be constructed by specifying either `"ECMrest"` or `"ECMunrest"` (case-insensitive) when defining a `Model` object,

[2]Personally, I would not recommend using any of these parameters when simulating (although they have been fully implemented in Pyvolve), as their interpretation is neither straight-forward nor particularly biological.

```python
# Simple restricted ECM
ecm_model = pyvolve.Model("ECMrest")

# Simple unrestricted ECM
ecm_model = pyvolve.Model("ECMunrest")

# Specifying "ECM" results in a *restricted ECM* model
ecm_model = pyvolve.Model("ECM")
```

As with mechanistic codon models, the dS and dN parameters can be specified with custom model parameter dictionary keys α, β, and/or ω (but again, these parameters do not correspond to dN/dS in the traditional sense!):

```python
# Restricted ECM with dN/dS parameter of 0.75
ecm_model = pyvolve.Model("ECMrest", {"omega":0.75})
```

The $\kappa(i, j)$ parameter is specified using the keys `"k_ti"` for transition bias and `"k_tv"`, for transversion bias. Specifically, `"k_ti"` corresponds to *ts*, and `"k_tv"` corresponds to *tv* in equations 9-11 in ref. [**?**]. Thus, each of these parameters can be specified as either 0, 1, 2, or 3 (the Pyvolve default is 1).

Finally, equilibrium frequencies can be customized with the `"state_freqs"` key in the custom model parameters dictionary (see Section 6.7 for details on frequency customization).

6.6 Specifying custom rate matrices

Rather than using a built-in modeling framework, you can specify a custom rate matrix. This rate matrix must be square and all rows in this matrix must sum to 0. Pyvolve will perform limited sanity checks on your matrix to ensure that these conditions are met, but beyond this, Pyvolve takes your matrix at face-value. In particular, Pyvolve will not scale the matrix in any manner.

Importantly, if you have *separate* state frequencies which have not been incorporated into your rate matrix already, the supplied matrix must be symmetric. If you do not supply state frequencies explicitly, Pyvolve will automatically determine them directly from your provided matrix.

When providing a custom matrix, you also have the option to provide a custom *code*, or custom states which are evolved. In this way, you can evolve characters of any kind according to any specified transition matrix. If you do not provide a custom code, Pyvolve checks to make sure that your matrix has dimensions of either 4×4, 20×20, or 61×61 (for nucleotide, amino-acid, or codon evolution, respectively). Otherwise, Pyvolve will check that your provided code and matrix are compatible (in terms of dimensions). Providing a custom code is, therefore, an attractive option for specifying arbitrary models of character evolution.

To specify a custom rate matrix, provide the argument `"custom"` as the first argument when defining a `Model` object, and provide your matrix in the custom-parameters dictionary using the key `matrix`. Any custom matrix specified should be either a 2D numpy array or a python list of lists. Below is an example of specifying a custom nucleotide rate matrix:

```python
import numpy as np # import to construct matrix

# Define a 4x4 custom rate matrix
custom_matrix= np.array([[-1.0, 0.33, 0.33, 0.34],
[0.25, -1.0, 0.25, 0.50],
[0.10, 0.80, -1.0, 0.10],
[0.34, 0.33, 0.33, -1.0]] )

# Construct a model using the custom rate-matrix
custom_model = pyvolve.Model("custom", {"matrix":custom_matrix})
```

Pyvolve automatically assumes that any 4×4 matrix indicates nucleotide evolution. As stated above, Pyvolve will extract equilibrium frequencies from this matrix and check that they are acceptable. This frequency vector will be automatically saved to a file called "custom_matrix_frequencies.txt", and these values will be used to generate the root sequence during simulation.

To provide a custom code, include the additional key `"code"`in your dictionary. Note that this key would be ignored for any built-in model.

```
import numpy as np # import to construct matrix

# Define a 3x3 custom rate matrix
custom_matrix= np.array([[-0.50, 0.30, 0.20],
[0.25, -0.50, 0.25],
[0.40, 0.10, 0.50]] )

custom_code = ["0", "1", "2"]
# Construct a model using the custom rate-matrix and the custom code
custom_model = pyvolve.Model("custom", {"matrix":custom_matrix, "code":custom_code})
```

The resulting data simulated using the above model will contain characters 0, 1, and 2. Although the above example shows a 3×3 matrix, it is certainly possible to specify custom matrices and codes for the "standard" dimensions of 4, 20, and 61.

6.7 Specifying equilibrium frequencies

Equilibrium frequencies can be specified for a given `Model` object with the key `"state_freqs"` in the custom-parameters dictionary. This key's associated value should be a list (or numpy array) of frequencies, summing to 1. The values in this list should be ordered alphabetically. For nucleotides, the list should be ordered ACGT. For amino-acids, the list should be ordered alphabetically, with regards to single-letter amino-acids abbreviations: ACDEFGHIKLMNPQRSTVWY. Finally, for codons, the list should be ordered AAA, AAC, AAG, AAT, ACA, ... TTT, *excluding* stop codons.

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, so I strongly recommend that you specify custom equilibrium frequencies for your simulations. Pyvolve provides a convenient class, called `StateFrequencies`, to help you with this step, with several child classes:

- **`EqualFrequencies`** (default)
  - Computes equal frequencies
- **`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`**[3]
  - Sets frequencies to default values for a given *empirical* model

All of these classes should be used with the following setup (the below code uses EqualFrequencies as a representative example):

```
# Define frequency object
f = pyvolve.EqualFrequencies("nucleotide") # or "amino_acid" or "codon", depending on
    your simulation
frequencies = f.compute_frequencies() # returns a vector of equilibrium frequencies
```

The constructed vector of frequencies (named "frequencies" in the example above) can then be provided to the custom model parameters dictionary with the key `"state_freqs"`. In addition, to conveniently save this vector of frequencies to a file, use the argument `savefile = <name_of_file>` when calling `.construct_frequencies()`:

```
# Define frequency object
f = pyvolve.EqualFrequencies("nucleotide")
frequencies = f.compute_frequencies(savefile = "my_frequency_file.txt") # returns a
    vector of equilibrium frequencies and saves them to file
```

6.7.1 EqualFrequencies class

Pyvolve uses this class to construct the default equilibrium frequencies. Usage should be relatively straightforward, according to the example above.

6.7.2 RandomFrequencies class

This class is used to compute "semi-random" equilibrium frequencies. The resulting frequency distributions are not entirely random, but rather are virtually flat distributions with some amount of noise.

6.7.3 CustomFrequencies class

With this class, you can provide a dictionary of frequencies, using the argument `freq_dict`, from which a vector of frequencies is constructed. The keys for this dictionary are the nucleotides, amino-acids (single letter abbreviations!), or codons, and the values should be the frequencies. Any states not included in this dictionary will be assigned a 0 frequency, so be sure the values in this dictionary sum to 1.

In the example below, `CustomFrequencies` is used to create a vector of amino-acid frequencies in which aspartate and glutamate each have a frequency of 0.25, and tryptophan has a frequency of 0.5. All other amino acids will have a frequency of 0.

```
# Define CustomFrequencies object
f = pyvolve.CustomFrequencies("amino_acid", freq_dict = {"D":0.25, "E":0.25, "W":0.5})
frequencies = f.compute_frequencies()
```

[3]Note that this is not actually a child class of `StateFrequencies`, but its behavior is virtually identical.

6.7.4 ReadFrequencies class

The `ReadFrequencies` class can be used to compute equilibrium frequencies from a file of sequences and/or multiple sequence alignment. Frequencies can be computed either using all data in the file, or, if the file contains an alignment, using specified alignment column(s). Note that Pyvolve will ignore all ambiguous characters present in this sequence file.

When specifying a file, use the argument `file`, and to specify the file format (e.g. "fasta" or "phylip"), use the argument `format`. Pyvolve uses BioPython to read the sequence file, so consult the BioPython AlignIO module documentation (or this nice wiki) for available formats. Pyvolve assumes a default file format of FASTA, so the `format` argument is not needed when the file is FASTA.

```
# Build frequencies using *all* data in the provided file
f = pyvolve.ReadFrequencies("amino_acid", file = "a_file_of_sequences.fasta")
frequencies = f.compute_frequencies()
```

To read frequencies from a specific column in a multiple sequence alignment, use the argument `columns`, which should be a list (*indexed from 1*) of integers giving the column(s) which should be considered in frequency calculations.

```
# Build frequencies using alignment columns 1 through 5 (inclusive)
f = pyvolve.ReadFrequencies("amino_acid", file = "alignment_file.fasta", columns =
    range(1,6))
frequencies = f.compute_frequencies()

# Build frequencies using only phylip-formatted alignment column 15
f = pyvolve.CustomFrequencies("amino_acid", file = "alignment_file.phy", format = "
    phylip", columns = 15)
frequencies = f.compute_frequencies()
```

6.7.5 EmpiricalModelFrequencies class

The `EmpiricalModelFrequencies` class will return the default vector of equilibrium frequencies for a given empirical model [amino-acid models and the codon model ECM, restricted and unrestricted versions (see ref. [?] for details)]. These default frequencies correspond to the frequencies originally published with each respective empirical model. Provide `EmpiricalModelFrequencies` with the name of the desired empirical model to obtain these frequencies:

```
# Obtain frequencies for the WAG model
f = pyvolve.EmpiricalModelFrequencies("WAG")
frequencies = f.compute_frequencies()

# For the ECM models, use the argument "ECMrest" for restricted, and "ECMunrest" for
    unrestricted
f = pyvolve.EmpiricalModelFrequencies("ECMrest") # restricted ECM frequencies
frequencies = f.compute_frequencies()
```

Note that Pyvolve uses these empirical frequencies as the default frequencies, if none are provided, for each respective empirical model!

6.7.6 Restricting frequencies to certain states

When using the classes `EqualFrequencies` and `RandomFrequencies`, it is possible to specify that only certain states be considered during calculations using the `restrict` argument, when defining the object. This argument takes a list of states (nucleotides, amino-acids, or codons) which should have non-zero frequencies. All states not included in this list will have a frequency of zero. Thus, by specifying this argument, frequencies will be distributed *only* among the indicated states.

The following example will return a vector of amino-acid frequencies evenly divided among the five specified amino-acids; therefore, each amino acid in the `restrict` list will have a frequency of 0.2.

```
# Compute equal frequencies among 5 specified amino acids
f = pyvolve.EqualFrequencies("amino_acid", restrict = ["A", "G", "V", "E", "F"])
frequencies = f.compute_frequencies()
```

Note that specifying this argument will have no effect on the `CustomFrequencies`, `ReadFrequencies`, or `EmpiricalModelFrequencies` classes.

6.7.7 Converting frequencies between alphabets

When defining a StateFrequencies object, you always have to indicate the alphabet (nucleotide, amino acid, or codon) in which frequency calculations should be performed. However, it is possible to have the `.construct_frequencies()` method return frequencies in a different alphabet, using the argument `type`. This argument takes a string specifying the desired type of frequencies returned (either "nucleotide", "amino_acid", or "codon").

This functionality is probably most useful when used with the ReadFrequencies class; for example, you might want to obtain amino-acid frequencies from multiple sequence alignment of codons:

```
# Define frequency object
f = pyvolve.ReadFrequencies("codon", file = "my_codon_alignment.fasta")
frequencies = f.compute_frequencies(type = "amino_acid")
```

As another example, you might want to obtain amino-acid frequencies which correspond to equal codon frequencies of $1/61$ each:

```
f = pyvolve.EqualFrequencies("codon")
frequencies = f.compute_frequencies(type = "amino_acid") # returns a vector of amino-
    acid frequencies that correspond to equal codon frequencies
```

Alternatively, you can also go the other way (amino acids to codons):

```
f = pyvolve.EqualFrequencies("amino_acid")
frequencies = f.compute_frequencies(type = "codon")
```

When converting amino acid to codon frequencies, Pyvolve assumes that there is *no codon bias* and assigns each synonymous codon the same frequency.

6.8 Specifying mutation rates

Nucleotide, mechanistic codon (dN/dS), and mutation-selection (MutSel) models all use nucleotide mutation rates as parameters. By default, mutation rates are equal for all nucleotide changes (e.g. the Jukes Cantor model [**?**]). These default settings can be customized, in the custom model parameters dictionary, in one of two ways:

1. Using the key `"mu"` to define custom rates for any/all nucleotide changes
2. Using the key `"kappa"` to specify a transition-to-transversion bias ratio (e.g. the HKY85 mutation model. [**?**])

The value associated with the `"mu"` key should itself be a dictionary of mutation rates, with keys "AC", "AG", "AT", etc, such that, for example, the key "AC" represents the mutation rate from A to C. Importantly, nucleotide and codon models use symmetric mutation rates; therefore, if a rate for "AC" is defined, the same value will automatically be applied to the change C to A. Thus, there are a total of 6 nucleotide mutation rates you can provide for a custom nucleotide and/or mechanistic codon model. Note that any rates not specified will be set to 1.

Alternatively, MutSel models do not constrain mutation rates to be symmetric, and thus, for instance, the "AC" rate may be different from the "CA" rate. Thus, there are a total of 12 nucleotide mutation rates you can provide for a custom MutSel model. Again, if a rate for "AC" but not "CA" is defined, then the "AC" rate will be automatically applied to "CA". Any unspecified nucleotide rate pairs will be set to 1.

```
# Example using customized mutation rates to construct a nucleotide model
custom_mutation_rates = {"AC":1.5, "AG":0.5, "AT":1.75, "CG":0.6, "CT":1.25, "GT":1.88
    }
my_model = pyvolve.Model("nucleotide", {"mu": custom_mutation_rates})
```

If, instead, the key `"kappa"` is specified, then the mutation rate for all transitions (e.g. purine to purine or pyrimidine to pyrimidine) will be set to the specified value, and the mutation rate for all transversions (e.g. purine to pyrimidine or vice versa) will be set to 1. This scheme corresponds to the HKY85 [**?**] mutation model.

```
# Example using customized kappa to construct a nucleotide model
my_model = pyvolve.Model("nucleotide", {"kappa": 3.5})
```

7 Mutation rates vs. branch lengths

In the context of Markov models implemented in Pyvolve, mutation rates **do not** have the same interpretation as they would in a population genetics framework. Here, mutation rates indicate the **relative probabilities** of mutating between different nucleotides. Mutation rates do not correspond the underlying rate of change across the tree – this quantity is represented instead by **branch lengths**. The main purpose of different mutation rates is to set up biases in mutation, for example if you want A$\rightarrow$T to occur at a 5-times higher rate than C$\rightarrow$T. To increase/decrease the rate of overall change, you'll want to change branch lengths. Note that branch lengths can be changed across the whole tree using the `scale_tree` keyword argument when calling `pyvolve.read_tree()` (see Section 5).

Importantly, here is what your branch lengths represent for each modeling framework -

- Nucleotide and amino-acid models: Mean number of substitutions per unit time

- Mechanistic codon models (dN/dS): By default, these branch lengths mean the mean number of substitutions per unit time, agnostic in terms of nonsynonymous vs. synonymous substitutions. To instead force branch lengths to represent the mean number of *neutral* substitutions per unit time, supply the argument `neutral_scaling = True` when defining a `Model` instance.
- Mutation–selection models: Mean number of *neutral* substitutions per unit time.

8 Defining Partitions

Partitions are defined using the `Partition()` class, with two required keyword arguments: `models`, the evolutionary model(s) associated with this partition, and `size`, the number of positions (sites) to evolve within this partition.

```
# Define a default nucleotide model
my_model = pyvolve.Model("nucleotide")

# Define a Partition object which evolves 100 positions according to my_model
my_partition = pyvolve.Partition(models = my_model, size = 100)
```

In cases of branch homogeneity (all branches evolve according to the same model), each partition is associated with a single model, as shown above. When branch hetergeneity is desired, a list of models used should be provided to the `models` argument (as detailed, with examples, in Section 11).

8.1 Specifying ancestral sequences

For each partition, you can assign an ancestral sequence which will be automatically used at the root of the phylogeny for the given partition. This can be accomplished using the keyword argument `root_sequence`:

```
# Define a default nucleotide model
my_model = pyvolve.Model("nucleotide")

# Define a Partition object with a specified ancestral sequence
my_partition = pyvolve.Partition(models = my_model, root_sequence = "GATAGAAC")
```

When providing an ancestral sequence, it is not required to also specify a size for the partition. Pyvolve will automatically determine this information from the provided root sequence.

Note that, when ancestral sequences are specified, site heterogeneity is not allowed (even if it was provided to the model used in this partition). Multiple partitions must be specified, each with different rates, to specify root sequences which will experience different rates across sites.

9 Evolving sequences

The callable class `Evolver` is Pyvolve's engine for all sequence simulation. Defining an `Evolver` object requires two keyword arguments: `partitions`, either the name of a single partition or a list of partitions to evolve, and `tree`, the phylogeny along which sequences are simulated.

Examples below show how to define an `Evolver` object and then evolve sequences. The code below assumes that the variables `my_partition` and `my_tree` were previously defined using `Partition` and `read_tree`, respectively.

```
# Define an Evolver instance to evolve a single partition
my_evolver = pyvolve.Evolver(partitions = my_partition, tree = my_tree)
my_evolver() # evolve sequences

# Define an Evolver instance to evolve several partitions
my_multpart_evolver = pyvolve.Evolver(partitions = [partition1, partition2, partition3
    ], tree = my_tree)
my_multpart_evolver() # evolve sequences
```

9.1 Evolver output files

Calling an `Evolver` object will produce three output files to the working directory:

1. **simulated_alignment.fasta**, a FASTA-formatted file containing simulated data
2. **site_rates.txt**, a tab-delimited file indicating to which partition and rate category each simulated site belongs (described in Section 9.2.1)
3. **site_rates_info.txt**, a tab-delimited file indicating the rate factors and probabilities associated with each rate category (described in Section 9.2.2)

In the context of complete homogeneity, in which all sites and branches evolve according to a single model, the files "site_rates.txt" and "site_rates_info.txt" will not contain much useful information. However, when sites evolve under site-wise and/or branch heterogeneity, these files will provide useful information for any necessary post-processing.

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

```
# Define an Evolver object
my_evolver = pyvolve.Evolver(tree = my_tree, partitions = my_partition)
# Evolve sequences with custom file names
my_evolver(ratefile = "custom_ratefile.txt", infofile = "custom_infofile.txt", seqfile
     = "custom_seqfile.fasta" )
```

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)
my_evolver = pyvolve.Evolver(tree = my_tree, partitions = my_partition)
my_evolver(ratefile = None, infofile = None)
```

The output sequence file's format can be changed with the argument `seqfmt`. 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
my_evolver = pyvolve.Evolver(tree = my_tree, partitions = my_partition)
my_evolver(seqfile = "seqs.phy", seqfmt = "phylip")
```

By default, the output sequence file will contain only the tip sequences. To additionally output all ancestral (including root) sequences, provide the argument `write_anc = True` when calling an `Evolver` object. Ancestral sequences will be included with tip sequences in the output sequence file (not in a separate file!). When ancestral sequences are written, the root sequence is denoted with the name "root", and internal nodes are named "internal_node1", "internal_node2", etc. To see precisely to which node each internal node name corresponds, it is useful to print the parsed newick tree with the function `print_tree`, as explained in Section 5.

```
# Output ancestral sequences along with the tip sequences
my_evolver = pyvolve.Evolver(tree = my_tree, partitions = my_partition)
my_evolver(write_anc = True)
```

9.2 Sequence post-processing

In addition to saving sequences to a file, `Evolver` can also return sequences back to you for post-processing in Python. Sequences can be easily obtained using the method `.get_sequences()`. This method will return a dictionary of sequences, where the keys are IDs and the values are sequences (as strings). Note that you must evolve sequences by calling your `Evolver` object before sequences can be returned!

```
# Return simulated sequences as dictionary
my_evolver = pyvolve.Evolver(tree = my_tree, partitions = my_partition)
my_evolver()
simulated_sequences = my_evolver.get_sequences()
```

By default, `.get_sequences()` will contain only the tip (leaf) sequences. To include ancestral sequences (root and internal node sequences) in this dictionary, specify the argument `anc = True`:

```
simulated_sequences = my_evolver.get_sequences(anc = True)
```

9.2.1 Interpreting the "site_rates.txt" output file

The output file "site_rates.txt" has three columns of data:

- **Site_Index**
  - Indicates a given position in the simulated data (indexed from 1)
- **Partition_Index**
  - Indicates the partition associated with this site
- **Rate_Category**
  - Indicates the rate category index associated with this site

The values in "Partition_Index" are ordered, starting from 1, based on the `partitions` argument list specified when setting up the `Evolver()` instance. Similarly, the values in "Rate_Category" are ordered, starting from 1, based on the rate heterogeneity lists (see Section 10 for details) specified when initializing the `Model()` objects used in the respective partition.

9.2.2 Interpreting the "site_rates_info.txt" output file

The output file "site_rates_info.txt" provides more detailed rate information for each partition. This file has give columns of data:

- **Partition_Index**
  - Indicates the partition index (can be mapped back to the Partition_Index column in "site_rates.txt")
- **Model_Name**
  - Indicates the model name (note that, if no name provided, this is None. Also, only relevant for branch het)
- **Rate_Category**
  - Indicates the rate category index (can be mapped back to the Rate_Category column in "site_rates.txt")
- **Rate_Probability**
  - Indicates the probability of a site being in the respective rate category
- **Rate_Factor**
  - Indicates either the rate scaling factor (for nucleotide and amino-acid models), or dN/dS value for this rate category for codon models

9.3 Simulating replicates

The callable `Evolver` class makes simulating replicates of given modeling scheme straight-forward: simply define an `Evolver` object, and then call this object in a for-loop as many times as needed.

```python
# Simulate 50 replicates
my_evolver = pyvolve.Evolver(tree = my_tree, partitions = my_partition)
for i in range(50):
    my_evolver(seqfile = "simulated_replicate" + str(i) + ".fasta") # Change seqfile
        name to avoid overwriting!
```

10 Implementing site-wise rate heterogeneity

This section details how to implement heterogeneity in site-wise rates within a partition.

10.1 Implementing site-wise heterogeneity for nucleotide and amino-acid models

In the context of nucleotide and amino-acid models, rate heterogeneity is applied by multiplying the rate matrix by scalar factors. Thus, sites evolving at different rates exhibit the same evolutionary patterns but differ in how quickly evolution occurs. Two primary parameters govern this sort of rate heterogeneity: the rate factors used to scale the matrix, and the probability associated with each rate factor (in other words, the probability that a given site is in each rate category).

Pyvolve models site-rate heterogeneity discretely, using either a discrete gamma distribution or a user-specified discrete rate distribution. Rate heterogeneity is incorporated into a `Model` object with several additional keyword arguments, detailed below.

10.1.1 Gamma-distributed rate categories

Gamma (Γ) distributed heterogeneity is specified with two-four keyword arguments when initializing a `Model` object:

- `alpha`, the shape parameter of the discrete gamma distribution from which rates are drawn (Note: following convention, $\alpha = \beta$ in these distributions [**?**]).
- `num_categories`, the number of rate categories to draw
- `pinv`, a proportion of invariant sites. Use this option to simulate according to Γ + I heterogeneity.

Examples for specifying Γ rate heterogeneity are shown below.

```python
# Gamma-distributed heterogeneity for a nucleotide model. Gamma shape parameter is 0.5
    , and 6 categories are specified.
nuc_model_het = pyvolve.Model("nucleotide", alpha = 0.5, num_categories = 6)

# Gamma-distributed heterogeneity for an amino-acid model. Gamma shape parameter is 0.
    5, and 6 categories are specified.
aa_model_het = pyvolve.Model("WAG", alpha = 0.5, num_categories = 6)

# Gamma+I heterogeneity for a nucleotide model with a proportion (0.25) invariant
    sites. Remaining sites are distributed according to a discrete gamma, with 5
    categories
nuc_model_het = pyvolve.Model("nucleotide", alpha = 0.2, num_categories = 5, pinv = 0.
    25)
```

10.1.2 Custom-distributed rate categories

A user-determined heterogeneity distribution is specified with one (or two) arguments when initializing a `Model` object:

- `rate_factors`, a list of scaling factors for each category
- `rate_probs`, an optional list of probabilities for each rate category. If unspecified, all rate categories are equally probable. This list should sum to 1!

Examples for specifying custom rate heterogeneity distributions are shown below.

```python
# Custom heterogeneity for a nucleotide model, with four equiprobable categories
nuc_model_het = pyvolve.Model("nucleotide", rate_factors = [0.4, 1.87, 3.4, 0.001])

# Custom heterogeneity for a nucleotide model, with four categories, each with a
    specified probability (i.e. rate 0.4 occurs with a probability of 0.15, etc.)
nuc_model_het = pyvolve.Model("nucleotide", rate_factors = [0.4, 1.87, 3.4, 0.001],
    rate_probs = [0.15, 0.25, 0.2, 0.5])

# Gamma-distributed heterogeneity for an amino-acid model, with four equiprobable
    categories
aa_model_het = pyvolve.Model("WAG", rate_factors = [0.4, 1.87, 3.4, 0.001])
```

If you would like to specify a proportion of invariant sites, simply set one of the rate factors to 0 and assign it a corresponding probability as usual:

```python
# Custom heterogeneity with proportion (0.4) invariant sites
nuc_model_het = pyvolve.Model("nucleotide", rate_factors = [0.4, 1.87, 3.4, 0.],
    rate_probs = [0.2, 0.2, 0.2, 0.4])
```

10.2 Implementing site-wise heterogeneity for mechanistic codon models

Due to the nature of mechanistic codon models, rate heterogeneity is not modeled with scalar factors, but with a distinct model for each rate (i.e. dN/dS value) category. To define a `Model` object with dN/dS heterogeneity, provide a *list* of dN/dS values the custom-parameters dictionary, rather than a single rate ratio value. As with standard codon models, you can provide dN/dS values with keys `"omega"`, `"beta"`, or `"alpha"` and `"beta"` together (to incorporate both synonymous and nonsynonymous rate variation).

By default, each discrete dN/dS category will have the same probability. To specify custom probabilities, provide the argument `rate_probs`, a list of probabilities, when initializing the `Model` object.

Examples for specifying heterogeneous mechanistic codon models are shown below (note that a GY-style model is shown in the examples, but as usual, both GY-style and MG-style are allowed.)

```
# Define a heterogeneous codon model with dN/dS values of 0.1, 0.5, 1.0, and 2.5 .
    Categories are, by default, equally likely.
codon_model_het = pyvolve.Model("GY", {"omega": [0.1, 0.5, 1.0, 2.5]})

# Define a heterogeneous codon model with two dN/dS categories: 0.102 (from 0.1/0.98)
    and 0.49 (from 0.5/1.02). Categories are, by default, equally likely.
codon_model_het = pyvolve.Model("GY", {"beta": [0.1, 0.5], "alpha": [0.98, 1.02]})

# Define a heterogeneous codon model with dN/dS values of 0.102 (with a probability of
     0.4) and 0.49 (with a probably of 0.6).
codon_model_het = pyvolve.Model("GY", {"beta": [0.1, 0.5], "alpha": [0.98, 1.02]},
    rate_probs = [0.4, 0.6])
```

10.3 Implementing site-wise heterogeneity for mutation-selection models

Due to the nature of MutSel models, site-wise heterogeneity should be accomplished using a series of partitions, in which each partition evolves according to a unique MutSel model. These partitions can then be provided as a list when defining an `Evolver` object.

10.4 Implementing site-wise heterogeneity for the Empirical Codon Model

Due to the peculiar features of this model (both empirically-derived transition probabilities and "mechanistic" parameters such as dN/dS), site-wise heterogeneity is not supported for these models, at this time. Pyvolve will simply ignore any provided arguments for site-rate heterogeneity with this model. Feel free to email the author to discuss and/or request this feature.

11 Implementing branch (temporal) heterogeneity

This section details how to implement branch (also known as temporal) heterogeneity within a partition, thus allowing different branches to evolve according to different models. To implement branch heterogeneity, your provided newick phylogeny should contain *model flags* at particular nodes of interest.

Model flags may be specified with either hashtags (#) or underscores (_), and they may be specified to satisfy one of two paradigms:

- Using **both trailing and leading symbols**, e.g. `_flagname_` or `#flagname#` . Specifying a model flag with this format will cause ALL descendents of that node to also follow this model, unless a new model flag is given downstream. In other words, this model will be propagated to apply to all children of that branch.
- Using **only a leading symbol**, e.g. `_flagname` or `#flagname`. Specifying a model flag with this format will cause ONLY that branch/edge to use the provided model. Descendent nodes will NOT inherit this model flag. Useful for changing model along a single branch, or towards a single leaf.

Model flags must be provided **AFTER** the branch length. Model flags may be repeated throughout the tree, but the model associated with each model flag will always be the same. Note that these model flag names **must** have correspondingly named model objects.

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. Trees with model flags, just like any other tree, are defined with the function `read_tree`. Some examples of trees with model flags:

```
# Define a tree with propagating model flags m1 and m2, with a string
het_tree = pyvolve.read_tree(tree = "(t4:0.785,(t3:0.380,(t2:0.806,(t5:0.612,t1:0.660):0.762_m1_):0.921_m2_):0.207);")
#OR
het_tree = pyvolve.read_tree(tree = "(t4:0.785,(t3:0.380,(t2:0.806,(t5:0.612,t1:0.660):0.762\#m1\#):0.921\#m2\#):0.207);")

# Print het_tree to see how model flags are applied:
pyvolve.print_tree(het_tree)
'''
>>>  root None None
>>>      t4 0.785 None
>>>      internalNode3 0.207 None
>>>           t3 0.38 None
>>>           internalNode2 0.921 m2
>>>               t2 0.806 m2
>>>               internalNode1 0.762 m1
>>>                    t5 0.612 m1
>>>                   t1 0.66 m1
'''

# Define a tree with non-propagating model flags m1 and m2
het_tree = pyvolve.read_tree(tree = "(t4:0.785,(t3:0.380,(t2:0.806,(t5:0.612,t1:0.660):0.762_m1):0.921_m2):0.207);")
#OR
het_tree = pyvolve.read_tree(tree = "(t4:0.785,(t3:0.380,(t2:0.806,(t5:0.612,t1:0.660):0.762#m1):0.921#m2):0.207);")

# Print het_tree to see how model flags are applied:
pyvolve.print_tree(het_tree)
'''
```

```
>>>  root None None
>>>      t4 0.785 None
>>>      internalNode3 0.207 None
>>>           t3 0.38 None
>>>           internalNode2 0.921 m2
>>>               t2 0.806 None
>>>               internalNode1 0.762 m1
>>>                    t5 0.612 None
>>>                   t1 0.66 None
'''
```

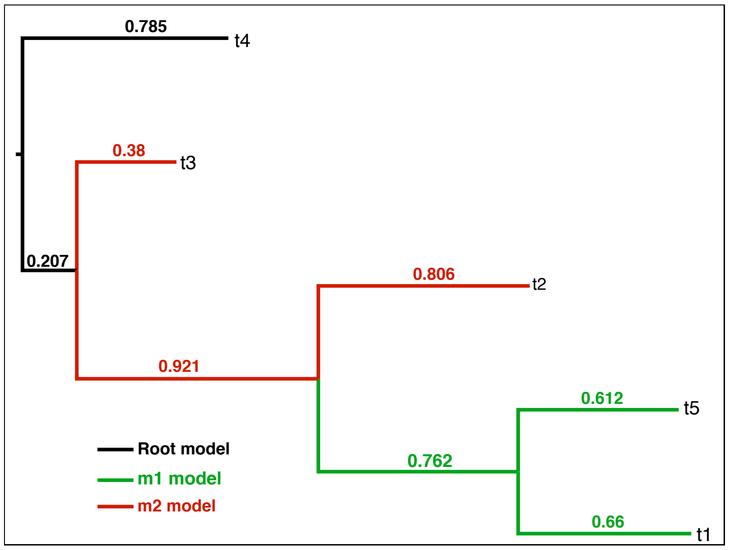

Figure 1: The newick tree with model flags given by
`"(t4:0.785,(t3:0.380,(t2:0.806,(t5:0.612,t1:0.660):0.762_m1_):0.921_m2_):0.207);"` indicates the model assignments shown.

All model flags specified in the newick phylogeny must have corresponding models. To link a model to a model flag, specify a given model's name using the keyword argument `name` when initializing a `Model` object. This name must be identical to a given model flag, *without* the leading and trailing symbols (e.g. the name "m1" corresponds to the flag _m1_ and/or #m1#).

The model at the root of the tree will not have a specific model flag, but nonetheless a model must be used at the root (obviously), and indeed at all other nodes which are not assigned a model flag (not that all branches on the tree which are not assigned a model flag will evolve according to the model used at the root). To specify a model at the root of the tree, simply create a model, with a name, and indicate this name when defining your partition.

Examples for defining models with names are shown below (for demonstrative purposes, nucleotide models with extreme state frequency differences are used here):

```
# Define the m1 model, with frequencies skewed for AT-bias
m1_model = pyvolve.Model("nucleotide", {"state_freqs":[0.4, 0.1, 0.1, 0.4]}, name = "
    m1")

# Define the m2 model, with frequencies skewed for GC-bias
m2_model = pyvolve.Model("nucleotide", {"state_freqs":[0.1, 0.4, 0.4, 0.1]}, name = "
    m2")

# Define the root model, with default equal nucleotide frequecies
root_model = pyvolve.Model("nucleotide", name = "root")
```

Alternatively, you can assign/re-assign a model's name with the `.assign_name()` method:

```
# (Re-)assign the name of the root model
root_model.assign_name("new_root_model_name")
```

Finally, when defining the partition that uses all of these models, provide all `Model` objects in list to the `models` argument. In addition, you *must* specify the name of the model you wish to use at the root of the tree with the keyword argument `root_model_name`.

```
# Define partition with branch heterogeneity, with 50 nucleotide positions
temp_het_partition = pyvolve.Partition(models = [m1_model, m2_model, root_model], size
     = 50, root_model_name = root_model.name)
```

12 Implementing branch-site heterogeneity

Simulating according to so-called "branch-site" models, in which there are both site-wise and branch heterogeneity, is accomplished using the same strategies shown for each individual aspect (branch, Section 11 and site, Section 10). However, there is a critical caveat to these models: all models within a given partition *must* have the same number of rate categories. Furthermore, the rate probabilities must be the same across models within a partition; if different values for `rate_probs` are indicated, then the probabilities provided for the *root model* will be applied to all subsequent branch models. (Note that this behavior is identical for other simulation platforms, like Indelible [**?**].)

The example below shows how to specify a branch-site heterogeneous nucleotide model with two models, root and model1 (note that this code assumes that the provided phylogeny contained the flag `_model1_`), when the rate categories are *not* equiprobable.

```
# Shared rate probabilities. Must be explicitly specified for all models (not just the
     root model)!
shared_rate_probs = [0.25, 0.3, 0.45]

# Construct a nucleotide model with 3 rate categories
root = Model("nucleotide", name = "root", rate_probs = shared_rate_probs, rate_factors
     = [1.5, 1.0, 0.05])

# Construct a second nucleotide model with 3 rate categories
model1 = Model("nucleotide", name = "model1", rate_probs = shared_rate_probs,
    rate_factors = [0.06, 2.5, 0.11])

# Construct a partition with these models, defining the root model nameas "root"
part = Partition(models = [root, model1], root_model_name = "root", size = 50)
```