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 11.2 below). Both site- and branch- (temporal) 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
my_evolver = pyvolve.Evolver(tree = my_tree, partitions = my_partition)
my_evolver() # evolve sequences
```

`Evolver()` will produce several output files, including the simulated data, as detailed in Section 6.1.

<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 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 8.

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 = 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
print_tree(phylogeny)
## Output from the above statement:
'''
 root None None
     t4 0.785 None
     internal_node3 0.207 None
         t3 0.38 None
         internal_node2 0.921 None
              t2 0.806 None
```

```
            internal_node1 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 (as detailed in Section 8)are not provided in the phylogeny.

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 models have 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_model()
```

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 10 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_model()
```

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_model()
```

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 assigns 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_model()
```

To customize an amino-acid model, specify the custom-parameters dictionary with the key **`"state_freqs"`** for custom equilibrium frequencies (see Section 10 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 = pyvolve.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_model()
```

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_model()

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

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

To customize a mechanistic codon model, include the optional keys **`"mu"`** for custom mutation rates and **`"state_freqs"`** for custom equilibrium frequencies (see Section 10 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}

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

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

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

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

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. (The relationship between equilibrium frequencies and fitness values for MutSel models is detailed in refs. [3**?**]). 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,

```
# 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})
mutsel_nuc_model_freqs.construct_model()

# 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})
mutsel_nuc_model_fits.construct_model()
```

Basic codon MutSel models can be constructed with,

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

# 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_model()

# 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_model()

# 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_model()
```

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...

4.6 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 [6]). 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. [4])

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})
my_models.construct_model()
```

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.

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

5 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")
my_models.construct_model()

# Define a partition which evolves according to this model of 100 sites
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 in Section 8).

6 Evolving sequences

The callable class **`Evolver()`** is Pyvolve's engine for all sequence simulation. Defining an **`Evolver()`** instance 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()`** instance and then evolve sequences. The code below assumes that all partition and tree variables provided as arguments to **`Evolver()`** have been 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
```

6.1 Evolver output files

By default, **`Evolver()`** will output three files, to the working directory, when called:

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 6.1.1)

3. **site_rates_info.txt**, a tab-delimited file indicating the rate factors and probabilities associated with each rate category (described in Section 6.1.2)

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 either site-wise 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 initializing an **`Evolver`** instance:

```
# Provide custom file names when initializing the Evolver instance
my_evolver = pyvolve.Evolver(tree = my_tree, partitions = my_partition, ratefile = "
    custom_ratefile.txt", infofile = "custom_infofile.txt", seqfile = "custom_seqfile.fasta" )
my_evolver() #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)
my_evolver = pyvolve.Evolver(tree = my_tree, partitions = my_partition, ratefile = None, infofile
     = None)
my_evolver() #evolve
```

The output sequence file's format can be changed with the argument **`seqfmt`** to **`Evolver()`**. 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, seqfile = "seqs.phy",
    seqfmt = "phylip")
my_evolver() #evolve
```

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`** to **`Evolver()`**. 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 3.

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

6.1.1 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 7 for details) specified when setting up the **`Model()`**/**`CodonModel()`** objects used in the respective partition.

6.1.2 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

7 Implementing site-wise rate heterogeneity

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

7.1 Nucleotide and amino-acid models

Rate heterogeneity is modeled for nucleotide and empirical amino-acid models discretely, using either a discrete gamma distribution or a user-specified rate distribution. Rate heterogeneity is incorporated in a model when **`.model_construct()`** is called.

7.1.1 Gamma-distributed rate categories

Gamma-distributed heterogeneity is specified with two (or three) arguments to the **`.model_construct()`** method:

- **`alpha`**, the shape parameter of the discrete gamma distribution from which rates are drawn (Note: following convention, $\alpha = \beta$ in these distributions [14]).
- **`num_cateogories`**, the number of rate categories to draw
- **`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 gamma-distributed rate heterogeneity are shown below.

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

# Gamma-distributed heterogeneity for a nucleotide model. Gamma shape parameter is 0.5, and 6 categories are specified. Categories are assigned specified probabilities
nuc_model_het = pyvolve.Model("nucleotide")
nuc_model_het.construct_model(alpha = 0.5, num_categories = 6, rate_probs = [0.2, 0.3, 0.3, 0.1, 0.05, 0.05])

# Gamma-distributed heterogeneity for an amino-acid model. Gamma shape parameter is 0.5, and 6 categories are specified. All categories have an equal probability
aa_model_het = pyvolve.Model("WAG")
aa_model_het.construct_model(alpha = 0.5, num_categories = 6)
```

7.1.2 Custom-distributed rate categories

A user-determined heterogeneity distribution is specified with one (or two) arguments to the **`.model_construct()`** method:

- **`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.

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

7.2 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 setup heterogenous codon models, you must define models using the **`CodonModel()`**, rather than the **`Model()`** class. Defining such models is virtually the same as defining **`(Model()`** objects, except a *list* of dN/dS values should be provided to account for rate heterogeneity. 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) in the custom model parameters dictionary.

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 calling the **`.construct_model()`** method.

Examples for specifying heterogeneous mechanistic codon models are shown below (note that the GY94 model is shown in the examples, but as usual, both GY94 and MG94 are accepted.)

```
# 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.CodonModel("GY94", {"omega": [0.1, 0.5, 1.0, 2.5]})
codon_model_het.construct_model()

# Define a heterogeneous codon model with dN/dS values of 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.CodonModel("GY94", {"beta": [0.1, 0.5], "alpha": [0.98, 1.02]})
codon_model_het.construct_model()

# 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.CodonModel("GY94", {"beta": [0.1, 0.5], "alpha": [0.98, 1.02]})
codon_model_het.construct_model(rate_probs = [0.4, 0.6])
```

7.3 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()`** class.

8 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 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.

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`**:

```
# Define a tree with 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);" )

# Define a tree with model flags m1 and m2, as read from a file
het_tree = pyvolve.read_tree( file = "/path/to/file/containing/tree/with/flags.tre" )
'''
```

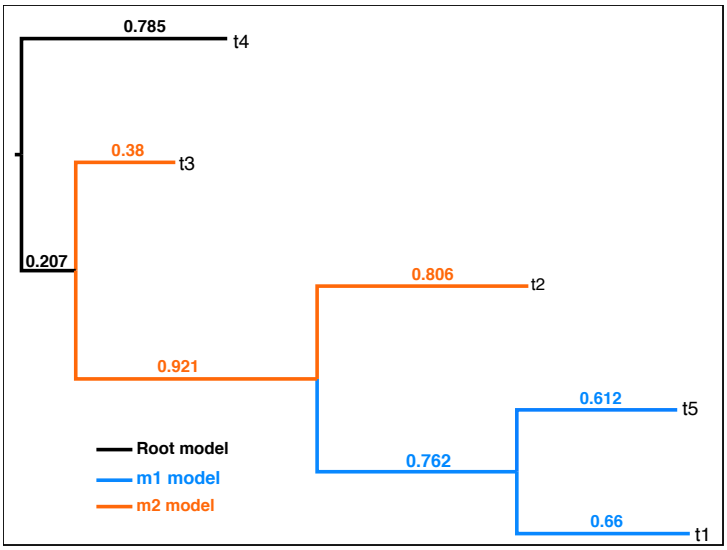

Figure 1: The newick tree with model flags given by `"(t4:0.785,(t3:0.380,(t2:0.806,(t5:0.612,t1:0.660):0.762_m1_):0.921)_m2_:0.207);"` indicates the model assignments shown.

All model flags specified in the newick phylogeny must have corresponding models, named identically (but without the leading/trailing underscores). To link a model to a model flag, specify a given model's name using the argument **`name`** in the **`.construct_model()`** method. 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]})
m1_model.construct_model(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]})
m2_model.construct_model(name = "m2")

# Define the root model, with default equal nucleotide frequecies
root_model = pyvolve.Model("nucleotide")
root_model.construct_model(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 models as a 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`** (the argument can be either the **`.name`** attribute or a the name as a string).

```
# 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)
```

9 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 8 and site, Section 7). 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 [1].)

CHECK THAT THIS CODE WILL WORK!!!!! 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")
root.construct_model(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")
mddel1.construct_model(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)
```

10 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 class, called StateFrequencies, to help you with this step, with several child 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

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.construct_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.construct_frequencies(savefile = "my_frequency_file.txt") # returns a vector of
    equilibrium frequencies and saves them to file
```

10.1 EqualFrequencies module

Pyvolve uses this module to construct the default equilibrium frequencies. Usage should be relatively straight-forward, according to the example above.

10.2 RandomFrequencies module

This module 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.

10.3 CustomFrequencies module

With this module, 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.construct_frequencies()
```

10.4 ReadFrequencies module

The ReadFrequencies module 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.CustomFrequencies("amino_acid", file = "a_file_of_sequences.fasta")
frequencies = f.construct_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 columns 1 through 5, inclusive of the alignment
f = pyvolve.CustomFrequencies("amino_acid", file = "alignment_file.fasta", columns = range(1,6))
frequencies = f.construct_frequencies()

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

10.5 EmpiricalModelFrequencies module

The EmpiricalModelFrequencies model will return the default vector of equilibrium frequencies for a given empirical model [amino-acid models JTT, WAG, and LG and the codon model ECM, restricted and unrestricted versions (see ref. [8] for details)]. These default frequencies correspond to the frequencies originally published with each respective matrix. To obtain these frequencies, provide, as an argument to EmpiricalModelFrequencies, the name of the desired model.

```
# Obtain frequencies for the WAG model
f = pyvolve.CustomFrequencies("WAG")
frequencies = f.construct_frequencies()

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

10.6 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 respective 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 among the provided 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.construct_frequencies()
```

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

10.7 Converting frequencies between alphabets

When defining a StateFrequencies variable, you always have to define 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.construct_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.construct_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.construct_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. Pyvolve does, however, allow you to specify codon bias by providing the argument, **`codon_bias`**, to the **`.construct_frequencies()`** method. This argument should be a value $x \in (0, 1]$ (exclusive of 0, inclusive of 1). Pyvolve will, for each amino acid, randomly select a preferred codon and assign this preferred codon a frequency x times the parent amino acid's frequency. All other codons are designated as non-preferred, and the remaining frequency amount is divided equally among them. In this way, the overall amino acid frequency is preserved, but it is partitioned different among its constituent codons. For example, if the specified bias is 0.05, and a given 4-fold degenerate amino acid's overall frequency is 0.1, then (randomly selected) preferred codon for this amino acid will have a frequency of 0.05, and the remaining three non-preferred codons will each have frequencies of $0.05/3$.

11 Special Features

11.1 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 for modeling frameworks which contain explicit parameters for natural selection (mechanistic codon and MutSel models). For example, 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 substitution rate is 1 when all states (nucleotides/codons) have equal fitness. Note that invoking neutral-scaling option has 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 = pyvolve.Model("GY94", {"omega":0.5}, "neutral") # Indicate neutral scaling with a third
    argument, "neutral"
m.construct_model()
```

While we believe that this neutral scaling approach leads to more realistic simulated data, we urge caution when using this scaling approach. 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 due to conventions in third-party softwares.

11.2 Specifying custom rate matrices

Rather than using a built-in modeling framework, you can specify a custom rate matrix. Any provided matrix must be a square matrix with dimensions of 4×4, 20×20, or 61×61 (for nucleotide, amino-acid, or codon evolution, respectively). Further, 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.

Note that you can still specify custom equilibrium frequencies, but these frequencies will not be applied to any matrix scaling - they will be used only for sampling the root sequence in simulation.

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 parameters dictionary using the key **`matrix`**. Any custom

matrix specified should be either a 2D numpy array or a python list of lists. Pyvolve orders nucleotides in the order ACGT, amino-acids in the order ACDEFGHIKLMNPQRSTVWY, and codons in the order AAA, AAC, AAG, ... TTT (without stop codons!). Below is an example of specifying a custom nucleotide rate matrix:

```
# Define the custom rate matrix (4x4 for nucleotide evolution)
custom_matrix= np.array([[-1.0, 0.33, 0.33, 0.33],
                         [0.25, -1.0, 0.25, 0.50],
                         [0.10, 0.80, -1.0, 0.10],
                         [0.33, 0.33, 0.33, -1.0]] )

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

11.3 Incorporating noise into branch lengths

Conventional sequence simulation algorithms apply a given branch length uniformly across a given branch. For example, if a given branch has a length of 0.1, then every site along that branch will evolve with a branch length of exactly 0.1. However, phylogenetic inference methods compute branch lengths as an average value for all sites along that branch, and there is no reasonable justification to assume all sites should have an identical branch length.

Therefore, Pyvolve allows you to specify some amount of noise in the branch lengths using the argument **`noisy_branch_lengths = True`** when defining an **`Evolver()`** object:

```
# Specify some amount of noise in branch lengths
evolver = Evolver(partitions = my_partitions, tree = my_tree, noisy_branch_lengths = True)
```

In this scheme, the *average* branch length for each branch will be the value specified in the newick phylogeny, but each site will evolve according to a slightly different value. Specifically, Pyvolve will sample, for each branch, branch lengths from a uniform distribution with a mean value equal to the branch length specified in the newick tree, and upper and lower bounds of $\pm$ 10% of this branch length. For example, the range of values drawn for a branch length of 0.25 will fall in 0.225 - 0.275 (calculated from $0.25 \pm (0.1 \times 0.25)$). By default, 10 values will be drawn from this distribution, and these branch lengths will be randomly assigned to sites along the branch.

These parameters can be further customized. To sample a different number of branch lengths per branch (rather than 10), specify the additional argument **`noisy_branch_lengths_n`** when defining an **`Evolver`** object. You can provide either a specific number to this argument, or the word "full" to indicate that each site should have a uniquely-sampled branch length (note that specifying "full" might be quite slow!)

```
# Specify 15 branch length categories per branch
evolver = Evolver(partitions = my_partitions, tree = my_tree, noisy_branch_lengths = True,
    noisy_branch_lengths_n = 15)

# Specify that every site should have a unique branch length
evolver = Evolver(partitions = my_partitions, tree = my_tree, noisy_branch_lengths = True,
    noisy_branch_lengths_n = "full")
```

Further, you can customize the uniform distribution's bounds with the argument **`noisy_branch_lengths_scale`**. The default value for this scale is 0.1, as described above. The scale value is applied as a percentage of the mean. For example, for a given branch length of 0.6 with a scale of 0.5, sampled branch lengths will be in the range 0.3 - 0.9 (calculated from $0.6 \pm (0.5 \times 0.6)$).

```
# Specify a wider range for branch length sampling
evolver = Evolver(partitions = my_partitions, tree = my_tree, noisy_branch_lengths = True,
    noisy_branch_lengths_scale = 0.75)

# Specify a very narrow range for branch length sampling
evolver = Evolver(partitions = my_partitions, tree = my_tree, noisy_branch_lengths = True,
    noisy_branch_lengths_scale = 0.01)
```

To customize both the number of branch length categories and the range, provide all arguments:

```
# Specify noisy branch lengths with wider sampling range and more categories
evolver = Evolver(partitions = my_partitions, tree = my_tree, noisy_branch_lengths = True,
    noisy_branch_lengths_n = 15, noisy_branch_lengths_scale = 0.75)
```

Note that the arguments **`noisy_branch_lengths_n`** and **`noisy_branch_lengths_scale`** will be ignored if **`noisy_branch_lengths`** is not set to **`True`**.

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.