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 10.2 below). Both site-wise and temporal (branch) heterogeneity are supported. A detailed and highly-recommended overview of Markov process evolutionary models, for DNA, protein, and codons, is available in the book *Computational Molecular Evolution*, by Ziheng Yang [14].

2 Basic Usage

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

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

```
1  ######### General pyvolve framework #########
2  ############################################
3
4  # Import the pyvolve module
5  import pyvolve
6
7  # Read in phylogeny along which pyvolve should simulate
8  my_tree = pyvolve.read_tree(file = "file_with_tree_for_simulating.tre")
9
10 # Define and construct evolutionary models
11 my_model = pyvolve.Model(<model_type>, <custom_model_parameters>)
12 my_model.construct_model()
13
14 # Define partitions
15 my_partition = pyvolve.Partition(models = my_model, size = 100)
```

```
16
17   # Evolve partitions with the callable Evolver() class
18   my_evolver = pyvolve.Evolver(tree = my_tree, partitions = my_partition)
19   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.

```
1   # Read phylogeny from file with the argument "file"
2   phylogeny = read_tree(file = "/path/to/tree/file.tre")
3
4   # Read phylogeny from string with the argument "tree"
5   phylogeny = read_tree(tree = "(t4:0.785,(t3:0.380,(t2:0.806,(t5:0.612,t1:0.660):0.762):0.921):0.
        207);")
```

To implement temporal (branch) heterogeneity, in which different branches on the phylogeny evolve according to different models, you will need to specify *model flags* at particular nodes in the newick tree, 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:

```
1   # Read phylogeny from string
2   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);")
3
4   # Print the parsed phylogeny
5   print_tree(phylogeny)
6   ## Output from the above statement:
7   '''
8    root None None
9        t4 0.785 None
10       internal_node3 0.207 None
11           t3 0.38 None
12           internal_node2 0.921 None
13               t2 0.806 None
14               internal_node1 0.762 None
15                   t5 0.612 None
16                   t1 0.66 None
17   '''
```

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

```
1  # Simple nucleotide model
2  nuc_model = pyvolve.Model("nucleotide")
3  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 9 for details on frequency customization).

```
1   # Define mutation rates in a dictionary with keys giving the nucleotide pair
2   # Below, the rate from A to C is 0.5, and similarly C to A is 0.5
3   custom_mu = {'AC':0.5, 'AG':0.25, 'AT':1.23, 'CG':0.55, 'CT':1.22, 'GT':0.47}
4
5   # Define custom frequencies, in order A C G T. This can be a list or numpy array.
6   freqs = [0.1, 0.45, 0.3, 0.15]
7
8   # Construct nucleotide model with custom mutation rates and frequencies.
9   nuc_model = pyvolve.Model( "nucleotide", {'mu':custom_mu, 'state_freqs':freqs} )
10  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 $A \to T$, it will automatically be applied as the rate $T \to A$.

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

```
1  # Construct nucleotide model with transition-to-transversion bias, and default frequencies
2  nuc_model = pyvolve.Model( "nucleotide", {"kappa":2.75, "state_freqs":freqs} )
3  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 assign equal equilibrium frequencies. A basic amino-acid model can be constructed with,

```
1  # Simple amino-acid model
2  aa_model = pyvolve.Model("WAG") # Here, WAG can be one of JTT, WAG, LG (case-insensitive)
3  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 9 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:

```
1  # Define default WAG state frequencies
2  f = pyvolve.EmpiricalModelFrequencies("WAG") # model name is case-insensitive
3  freqs = f.construct_frequencies()
4
5  # Construct amino-acid model with WAG frequencies
6  aa_model = pyvolve.Model( "WAG", {"state_freqs":freqs} )
7  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 → 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.

5

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

```
1   # Simple GY-style model (specify as GY94)
2   gy_model = pyvolve.Model("GY94", {'omega': 0.5})
3   gy_model.construct_model()
4
5   # Simple MG-style model (specify as MG94)
6   mg_model = pyvolve.Model("MG94", {'alpha': 1.04, 'beta': 0.67})
7   mg_model.construct_model()
8
9   # Specifying "codon" results in a *GY-style* model
10  codon_model = pyvolve.Model("codon", {'beta': 1.25})
11  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 9 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.

```
1   # Define mutation rates in a dictionary with keys giving the nucleotide pair
2   # Below, the rate from A to C is 0.5, and similarly C to A is 0.5
3   custom_mu = {'AC':0.5, 'AG':0.25, 'AT':1.23, 'CG':0.55, 'CT':1.22, 'GT':0.47}
4
5   # Construct codon model with custom mutation rates
6   codon_model = pyvolve.Model( "codon", {'mu':custom_mu, 'omega':0.55} )
7   codon_model.construct_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.

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

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

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

MutSel models are implemented both for codons and nucleotides, and they may be specified either with equilibrium frequencies or with fitness values. Note that equilibrium frequencies must sum to 1, but fitness

values are not constrained in any way. (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,

```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})
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,

```python
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.

```
1  # Example using customized mutation rates to construct a nucleotide model
2  custom_mutation_rates = {"AC":1.5, "AG":0.5, "AT":1.75, "CG":0.6, "CT":1.25, "GT":1.88}
3  my_model = pyvolve.Model("nucleotide", {"mu": custom_mutation_rates})
4  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.

```
1  # Example using customized kappa to construct a nucleotide model
2  my_model = pyvolve.Model("nucleotide", {"kappa": 3.5})
3  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.

```
1  # Define a default nucleotide model
2  my_model = pyvolve.Model("nucleotide")
3  my_models.construct_model()
4
5  # Define a partition which evolves according to this model of 100 sites
6  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.

```
1  # Define an Evolver instance to evolve a single partition
2  my_evolver = pyvolve.Evolver(partitions = my_partition, tree = my_tree)
3  my_evolver() # evolve sequences
4
5  # Define an Evolver instance to evolve several partitions
6  my_evolver = pyvolve.Evolver(partitions = [partition1, partition2, partition3], tree = my_tree)
```

6.1 Evolver output files

By default, **Evolver()** will output three files 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

3. **site_rates_info.txt**, a tab-delimited file indicating the rate factors and probabilities associated with each rate category

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 **ratefile** ("site_rates.txt"), **infofile** ("site_rates_info.txt"), and/or **seqfile** ("simulated_alignment.fasta") when initializing an **Evolver** instance:

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

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

```
1  # Only output a sequence file (suppress the ratefile and infofile)
2  my_evolver = pyvolve.Evolver(tree = my_tree, partitions = my_partition, ratefile = None, infofile
       = None)
3  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.

```
1  # Save the sequence file as seqs.phy, in phylip format
2  my_evolver = pyvolve.Evolver(tree = my_tree, partitions = my_partition, seqfile = "seqs.phy",
       seqfmt = "phylip")
3  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.

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

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.

```
1  # Gamma-distributed heterogeneity for a nucleotide model. Gamma shape parameter is 0.5, and 6
       categories are specified. All categories have an equal probability
2  nuc_model_het = pyvolve.Model("nucleotide")
3  nuc_model_het.construct_model(alpha = 0.5, num_categories = 6)
4
5  # Gamma-distributed heterogeneity for a nucleotide model. Gamma shape parameter is 0.5, and 6
       categories are specified. Categories are assigned specified probabilities
6  nuc_model_het = pyvolve.Model("nucleotide")
7  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])
8
9  # 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
10 aa_model_het = pyvolve.Model("WAG")
11 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.

```
1  # Custom heterogeneity for a nucleotide model, with four equiprobable categories
2  nuc_model_het = pyvolve.Model("nucleotide")
3  nuc_model_het.construct_model(rate_factors = [0.4, 1.87, 3.4, 0.001])
4
5  # 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.)
6  nuc_model_het = pyvolve.Model("nucleotide")
7  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])
8
9  # Gamma-distributed heterogeneity for an amino-acid model, with four equiprobable categories
10 aa_model_het = pyvolve.Model("WAG")
11 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.)

```
1  # 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.
2  codon_model_het = pyvolve.CodonModel("GY94", {"omega": [0.1, 0.5, 1.0, 2.5]})
3  codon_model_het.construct_model()
4
5  # 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.
6  codon_model_het = pyvolve.CodonModel("GY94", {"beta": [0.1, 0.5], "alpha": [0.98, 1.02]})
7  codon_model_het.construct_model()
8
9  # 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).
10 codon_model_het = pyvolve.CodonModel("GY94", {"beta": [0.1, 0.5], "alpha": [0.98, 1.02]})
11 codon_model_het.construct_model(rate_probs = [0.4, 0.6])
```

8   Implementing temporal (branch) heterogeneity

This section details how to implement temporal (also known as branch) heterogeneity within a partition, thus allowing different branches to evolve according to different models. To implement temporal 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`:

```
1  # Define a tree with model flags m1 and m2, with a string
2  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);" )
3
4  # Define a tree with model flags m1 and m2, as read from a file
5  het_tree = pyvolve.read_tree( file = "/path/to/file/containing/tree/with/flags.tre" )
6  '''
```
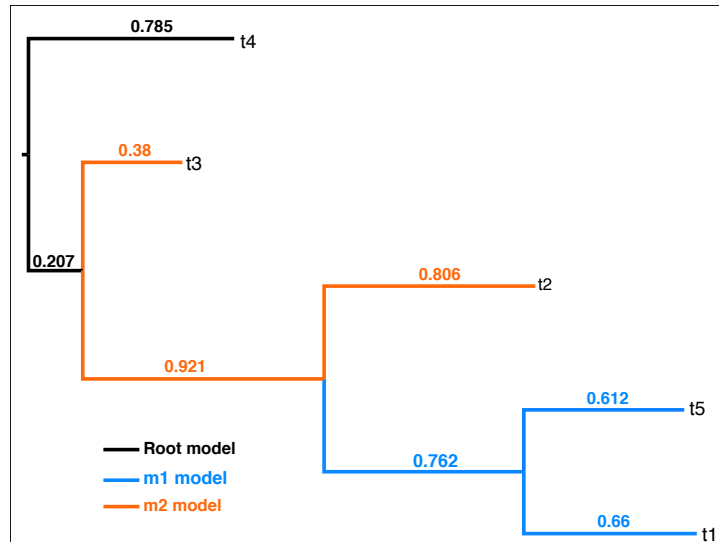
Figure 1: The newick tree with model flags given by
`"(t4:0.785,(t3:0.380,(t2:0.806,(t5:0.612,t1:0.660):0.762_m1_):0.921)_m2_:0.207);"` indicates the model assignments shown.

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, and thus pyvolve will automatically name this model "root". Further, all branches on the tree which are not assigned a model flag will evolve according to this root model. Therefore, you must be sure to define a model named "root"!

Examples for defining models with names are shown below. Nucleotide models with extreme state frequency differences are used as an example.

```
1   # Define the m1 model, with frequencies skewed for AT-bias
2   m1_model = pyvolve.Model("nucleotide", {'state_freqs':[0.4, 0.1, 0.1, 0.4]})
3   m1_model.construct_model(name = "m1")
4
5   # Define the m2 model, with frequencies skewed for GC-bias
6   m2_model = pyvolve.Model("nucleotide", {'state_freqs':[0.1, 0.4, 0.4, 0.1]})
7   m2_model.construct_model(name = "m2")
8
9   # Define the root model, with default equal nucleotide frequecies
10  root_model = pyvolve.Model("nucleotide")
11  root_model.construct_model(name = "root")
```

Finally, when defining the partition that uses all of these models, provide models as a list to the **models** argument.

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

9 Building a vector of equilibrium frequencies

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

- EqualFrequencies (**default**)
  Sets frequencies as equal

- RandomFrequencies
  Computes (semi-)random frequencies

- CustomFrequencies
  Computes frequencies from a user-provided dictionary of frequencies

- ReadFrequencies
  Computes frequencies from a sequence or alignment file.

- EmpiricalModelFrequencies
  Sets frequencies to default values for a given *empirical* model

Basic usage of these classes:

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

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

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

10 Special Features

10.1 Matrix scaling options

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

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

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

Unfortunately, this scaling approach can lead to some unexpected results. In particular, when multiple mechanistic codon (dN/dS) models are used, thus allowing for variable dN/dS values across sites, multiple

rate matrices must be used – one matrix per dN/dS value. This scaling approach, then, would cause sites with $dN/dS = 0.05$ to experience the same average number of substitutions as sites with $dN/dS = 2.5$. From a biological perspective, this result is undesirable, as sites with low dN/dS values should evolve more slowly than sites with high dN/dS values.

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

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

```python
# 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()
```

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

10.2   Specifying custom rate matrices

This is my first python example:

```python
from pyvolve import *

# Read in a newick tree
t = read_tree(tree = "(t4:0.785,(t3:0.380,(t2:0.806,(t5:0.612,t1:0.660):0.762):0.921):0.207);")

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

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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