

---

# **phylomodels**

**Institute for Disease Modeling**

**Mar 28, 2022**



# CONTENTS

|          |                                  |           |
|----------|----------------------------------|-----------|
| <b>1</b> | <b>Documentation</b>             | <b>3</b>  |
| <b>2</b> | <b>Files</b>                     | <b>5</b>  |
| <b>3</b> | <b>Data structures</b>           | <b>7</b>  |
| <b>4</b> | <b>Installation</b>              | <b>9</b>  |
| <b>5</b> | <b>Requirements/Dependencies</b> | <b>11</b> |
| <b>6</b> | <b>Examples</b>                  | <b>13</b> |
| 6.1      | Models . . . . .                 | 13        |
| 6.2      | Features . . . . .               | 15        |
| 6.3      | Calibration . . . . .            | 18        |
| 6.4      | Test . . . . .                   | 20        |
| 6.5      | Examples . . . . .               | 21        |
| 6.6      | API reference . . . . .          | 22        |
| 6.7      | To do . . . . .                  | 29        |



Python-based library with utilities for modeling and calibration, with emphasis on exploiting phylogenetic data.

**Contents**

- *Documentation*
- *Files*
- *Data structures*
- *Installation*
- *Requirements/Dependencies*
- *Examples*



## DOCUMENTATION

Documentation is available at <https://docs.idmod.org/projects/phylomodels/en/latest/>.





## FILES

phyloModels contains the following folders:

**calibration** Algorithms and tools for model calibration.

**docs** Documentation and notes.

**examples** Battery of examples and use-cases of modules.

**features** Summary statistics for structured (e.g., time series) and unstructured (e.g., trees, graphs) data.

**models** Algorithms and tools for disease transmission models.

**network** Algorithms and utilities for generation and manipulation of graphs/trees/networks.

**sampling** Algorithms and utilities for sampling model generated data.

**test** Main programs that test the library and demonstrate its performance.

**utilities** Miscellaneous functions (read/write from/to files, data conversions, etc.)

**visualization** Functions and utilities for visualization.



## **DATA STRUCTURES**

Structured data is represented using Pandas dataframes. Phylogenetic trees are represented using ETE3 tree objects.



## INSTALLATION

1. Download the library:

```
git clone https://github.com/InstituteForDiseaseModeling/phyloModels
```

2. Install dependencies.

Packages not on PyPI will not be installed by `setuptools` in step 3 below. Hence, they should be installed separately. These packages include CUDA and History Matching (at this time, CUDA and History Matching are required only for the calibration module; functions in other modules can run without these libraries).

We recommend using a virtual environment. You can configure a virtual environment using the Conda definition file that is included in this package, or you can create a virtual environment with the required dependencies from scratch. Follow one of the steps below:

- Create and configure virtual environment using an environment definition file.

A Conda environment definition file is provided with this package. This file, called *phylomodels\_env.yml*, includes definitions of all requirements, even packages not on PyPI. To use it for installing dependencies, edit the first line to set the name of the environment, and then execute the following commands for installing and activating a conda environment that satisfies all required dependencies:

```
conda env create -f phylomodels_env.yml
conda activate [environment name]
```

---

**Note:** In some systems, the environment activation command is `source activate [environment name]` instead of `conda activate [environment name]`.

---

- Create and configure virtual environment using Conda.

Other than CUDA libraries, which are only used for the calibration module, most of the dependencies can be installed during the installation step (see item 3 below). However, there is an exception with some of the dependencies of the ETE3 library, which is used for representing tree data structures. These dependencies and their required versions can be installed as follows:

```
conda create --name [environment name] python==3.7
conda activate [environment name]
pip install PyQt5==5.11.3
```

- Create and configure virtual environment using `venv`.

The process is similar to creating a Conda virtual environment:

```
python3 -m venv [environment name]
source [environment name]/bin/activate
pip install PyQt5==5.11.3
```

3. Install the package:

```
python3 setup.py develop
```

---

**Note:** Alternatively, you could run `python3 setup.py install` if you plan on using the package as is, without making any changes to the code.

---

4. Register environment as jupyter kernel (optional).

You may need to manually add the new environment as a jupyter kernel. With the conda environment activated, this can be done as follows:

```
python -m ipykernel install --user --name [environment name] --display-name "Python ↵
↳([environment name])"
```

## REQUIREMENTS/DEPENDENCIES

See dependencies in `setup.py` and `phylomodels_env.yml`.





## EXAMPLES

- Computing summary statistics on phylogenetic trees: `phylomodels/examples/features_computeTreeFeatures/features_computeTree`
- Model calibration using History Matching: `phylomodels/examples/calibration_historyMatching_featureSelectionForSIR/calibratio`

### 6.1 Models

The `phylomodels.models` module contains wrappers to models, as well as any code required for execution of models. The goal of this module is to provide a standard interface for sampling, feature extraction, and calibration functions.

#### 6.1.1 File structure

This folder is organized as a collection of model wrappers. The wrappers themselves are (typically small) Python functions coded inside individual files. Additional functions (and files) that are necessary for the execution of a model must be in an individual subfolder. Optionally, there could be a test script for each individual wrapper.

For example, an SIR stochastic simulator based on Gillespie's tao-leap method is coded as a class called `sirTaoLeap`. This class is coded in a file called `sirTaoLeap.py`. The file structure for phyloModels-standard access to `sirTaoLeap` is as follows (see naming conventions below):

**`sir_taoLeap_getIncidence.py`** Wrapper function for accessing `sirTaoLeap`.

**`test_sir_taoLeap_getIncidence.py`** Optional script for testing the `sir_taoLeap_getIncidence` wrapper.

**`./sir_taoLeap`** `sirTaoLeap` class and auxiliary functions.

---

**Note:** A separate folder with model code may not be always necessary. For example, accessing models from a package that is already installed (and whose code is located in a separate folder/folder structure) may be called directly from the wrapper.

---

#### 6.1.2 Naming conventions

##### Wrappers

Wrappers should be named as follows:

```
[type]_[descriptive name of model/library]_[_configuration name (optional)]
```

**type** Indicates the type of model that is being implemented. Type can be any of the following:

- `sir` for SIR models (or variations of `sir` for other compartmental models (e.g., SEIR))
- `distributions` for random variables and probability distributions

**descriptive name of model/library** References the model or library that is being used by this wrapper.

**configuration name** Optional, describes a particular configuration mode used when calling the model.

The following are examples of wrapper names for different configurations of a model called `sirTaoLeap`:

```
sir_taoLeap
sir_taoLeap_getIncidence
sir_taoLeap_getPrevalence
sir_taoLeap_getIncidenceSampled
sir_taoLeap_getSampledIncidence
```

### Tests

Wrapper test scripts should take the name of the wrapper with the `test_` prefix.

### Subfolders

As mentioned above, any other file necessary for interfacing a model (or executing it) must be placed in a separate subfolder. The subfolder should be named as follows:

```
[type]_[descriptive name of model/library]
```

where `type` and `descriptive name of model/library` follow the descriptions mentioned for the wrappers above.

For example, a subfolder containing files for the `sirTaoLeap` model described for wrappers above should be named `sir_taoLeap`.

Note that the existence of a subfolder for a given model is *optional*. A subfolder should only be created if there is at least one file (other than the wrapper) necessary for calling a model.

### 6.1.3 Inputs

Each wrapper receives only one input argument. This argument is a pandas dataframe. Each row of this dataframe is a set of arguments to be used as input for the model call. Columns are the arguments that will be passed to the model.

The input dataframe must not be modified by the wrapper or model.

### 6.1.4 Outputs

The output is a tuple with the following items (in order):

1. A pandas dataframe, with the same number of rows of the input argument. Each row of this dataframe is the (possibly reformatted) model output corresponding to the input dataframe row. The output dataframe could be, for example, a time series.
2. A list of NetworkX structures. Each item of this list corresponds to the output of the model to the arguments in the corresponding row of the input dataframe.

## 6.2 Features

The `phylomodels.features` module contains the definition of the `features` class, as well as internal methods and functions used by this class. The `features` class is an interface for the definition, computation, and general analyses of features. We may also refer to features as summary statistics.

The ultimate goal of the `features` module is to provide a standard interface that enables easy and streamlined definition, addition, deletion, and overall management of individual features. When using this module, the addition of a new feature in a given Python script should take at most 1 additional line of code. Coding and integration of new features could be also made in a few minutes.

For example, the basic usage of this module within a Python program should look as follows:

```
x = readTimeSeries() # One or more time series to be analyzed
f = features(x)      # Instantiation of features object
f.compute()          # Compute features
y, s = f.get()        # Extract features and statistics as a Pandas DataFrame
```

Computation of features can be enabled/disabled either by groups or individually:

```
f.disableFeature(group="all") # Disable computation of all available
                               # features
f.enableFeature(name="series_sum") # Enable computation of the feature called
                                    # series_sum
f.compute() # Only series_sum is computed
```

### 6.2.1 File structure

**features.py** Definition of features class

**inventory.py** Methods and auxiliary functions for management of features.

**testFeatures.py** Test script.

**./graphs** Methods and auxiliary functions for features that are obtained from graphs.

**./series`** Methods and auxiliary functions for features that are obtained from time series.

**./statistics** Methods and auxiliary functions for computing statistics on computed features.

### 6.2.2 Inputs

The instantiation of a features object can receive any of the following inputs, which are attributes of the `features` class:

**x** Set of time series to analyze. This is a pandas dataframe with  $m$  rows by  $n$  columns.  $m$  is the number of sequences (i.e., each row is a different time series) and  $n$  is the length of each sequence.

**xref** Reference time series. This is a pandas dataframe with 1 row and  $m$  columns, where  $m$  is defined as in **x** above.

**g** Set of graphs (e.g., trees) to analyze. This is a list of NetworkX data structures (see <https://networkx.github.io/>).

**gref** Reference graph. This is a NetworkX data structure.

### 6.2.3 Core methods

There main methods for the computation and analysis of features are:

**compute** Compute all the features and statistics currently enabled (see “Management of features” below for details on enabling/disabling features).

**get** Return features and statistics rendered by the **compute** method.

### 6.2.4 Management of features

The features class can extract and/or generate any number of features. Features are only computed when the **compute** method is called. Important notes regarding the management of features:

**Available features** Potential features to be extracted or generated should be in an inventory of features. This inventory is an attribute called `availableFeatures`. `availableFeatures` is a list of **feature cards**.

**Feature card** A feature card contains a brief description of a feature, as well as information regarding the actual function that computes the feature. It is defined based on the following namedtuple:

```
featureCard \\  
  = collections.namedtuple(  
      "featureCard",  
      ["id", "name", "description", "subroutine"]  
  )
```

where “id” is an integer, “name” is a string (see naming conventions below), “subroutine” is the name of the function that extracts and/or computes the feature, and “parameters” is a list of arguments for said function.

**Enabled features** A feature is to be computed only if it is enabled. In general, the **compute** method loops over `availableFeatures` and checks if they are enabled. If so, then the corresponding subroutines are called. The status of a feature (enabled/disabled) is maintained in the Boolean attribute `enabledFeatures`.

#### Methods

**enableFeature** Enable the computation/extraction of a feature or group of features.

**disableFeature** Disable the computation/extraction of a feature or group of features.

**listFeatures** Display list of available features.

**listActiveFeatures** Display list of active (i.e., enabled) features.

### 6.2.5 Designing and integrating new features

- Each feature must be computed by a single function. The goal is for each of these functions to be very short. This makes it easy to code, easy to review, and easy to maintain.
- Each feature function receives two arguments, namely: input `x` transformed into a numpy array of size `m x n` (with `m` and `n` defined as indicated above), followed by input `xref` transformed into a numpy array. Not all the arguments have to be used by a feature function, but the order in which arguments are passed to the function must be maintained.
- Each feature function returns a Pandas DataFrame. Each row in this DataFrame is the output for the corresponding row/item in the input (e.g., row `k` of the DataFrame is the output corresponding to row `k` in `x` and `g`).
- Each feature function must be in a separate file (typically of just a few lines). The name of the file must be the same name of the feature function. It must also follow the naming conventions described below.

- A new feature function is integrated into the module by moving its corresponding file to the subfolder `./series` or `./graphs`, depending on which type of feature it is. In addition, the `__init__.py` file on the corresponding subfolder must be edited to include a line importing the function. This line looks like this:

```
from [series/graphs] import [function name]
```

- Naming conventions: Features should be named as follows:

```
[type of input]_[transformation of the input]_[other labels]
```

For example:

```
series_sum
series_derivative
graph_nodes
graph_degree
```

## 6.2.6 Statistics

- Statistics describe the behavior of the computed features. For example, the **mean** statistic finds the average of every column in the DataFrame rendered by the method **get**.
- Management and design of statistics follow the same principles than those described for management and design of features, with a few changes:
  - Inventory management is done using the arrays `availableStats` and `enabledStats`.
  - The inventory management methods are: `enableStatistic`, `disableStatistic`, `listStatistics`, `registerStatistic`.
  - Files that contain the code for computing a statistic must be saved into the `./statistics` subfolder.

## 6.2.7 Utilities

The following utility methods are available:

**getArguments** Returns `x`, `xref`, `g`, and `gref`.

**listFeatures** Display available features.

**listActiveFeatures** Display active features.

**printArguments** Display the contents of `x`, `xref`, `g`, and `gref`.

**printFeatures** Display the features dataframe.

**printStats** Display statistics.

## 6.3 Calibration

The `phylomodels.calibration` module contains wrappers to calibration functions and libraries, as well as any code required for their use. The goals of this module are to provide a standard interface for calibration jobs, as well as maintaining a battery of calibration methods that are well suited for our needs.

There are two types of tasks that can be done using functions in this module:

- **calibration initialization**, which generates one or more initial points for faster convergence of calibration algorithms; and
- **calibration**, which executes an actual calibration job.

Calibration algorithms could use the output of an initialization algorithm. However, the use of initialization algorithms is not required.

### 6.3.1 File structure

This directory is organized as a collection of wrappers for calibration and calibration initializers. The wrappers are Python functions. The name of the function should be the same name of the file that contains it. Additional functions (and files) that are necessary for the execution of a calibration algorithm or initializer must be in an individual subdirectory. Optionally, there could be a test script for each wrapper.

For example, a basic configuration of history matching as an initializer could require the following file structure:

**init\_historyMatching\_basic** Wrapper function for history matching.

**test\_init\_historyMatching\_basic** Optional script for testing the basic configuration of history matching.

**./init\_historyMatching** Auxiliary functions for the execution of history matching.

---

**Note:** A separate folder with model code may not be always necessary. For example, accessing models from a package that is already installed (and whose code is located in a separate folder/folder structure) may be done directly from the wrapper.

---

### 6.3.2 Naming conventions

#### Wrappers

Wrappers should be named as follows:

`[type]_[descriptive name of algorithm/method/library]_[_configuration name (optional)]`

**type** Indicates the type of operation that is being done. It can be:

- `init`, for calibration initializer; or
- `cal`, for calibration operations.

**descriptive name of algorithm/method/library** Refers to the algorithm or library that is being used by this wrapper.

**configuration name** Optional, describes a particular configuration mode used when calling the algorithm.

The following are examples of wrapper names for different configurations of a history matching initializer:

```
init_historyMatching
init_historyMatching_poissonBasis
init_historyMatching_gaussianBasis
init_historyMatching_rejectionRules
init_historyMatching_rejectionRules_PoissonBasis
```

## Tests

Wrapper test scripts should take the name of the wrapper with the `test_` prefix.

## Subfolders

As mentioned above, any other file necessary for interfacing a method or library must be placed in a separate subfolder. The subfolder should be named as follows:

```
[type]_[descriptive name of algorithm/method/library]
```

where `type` and `descriptive name of algorithm/method/library` follow the descriptions mentioned for the wrappers above.

For example, a subfolder containing files for using history matching would be named `init_historyMatching`.

---

**Note:** The existence of a subfolder for a given method is *optional*. A subfolder should only be created if there is at least one file (other than the wrapper) necessary for calling a model.

---

## 6.3.3 Inputs

### Initializers

Each wrapper receives the following arguments (in this order):

**xInfo** Name and range of model parameters. This is a pandas dataframe with 3 columns, namely: *name*, *min*, and *max*. Each row of the dataframe contains the corresponding information for a given model parameter.

**y** Observation or measurements. This is a pandas dataframe. Columns in this dataframe are the features that will be used for calibration. The name of the features correspond to names of features that can be computed by the *features* module (those names are used by the calibration algorithm for configuring calls to *features* methods. The first row of *y* contains the observations that will be used by the calibration method for fitting the model. Optionally, the second row may contain the variance of each observation. The variance may be used by initialization and/or calibration algorithms.

**model** Pointer to model or model wrapper. The interface to the model (or its wrapper) should be the one defined in the `phylomodels.models` module.

**params** Other parameters for the initializer. This argument is a Python dictionary (i.e., key-value pairs with the name of the parameters and their value).

Inputs to initializers must not be modified by the initializer (or any of the functions called internally by the initializer).

### Calibration solvers

Each calibration solver receives the following arguments (in this order):

**xInfo** Name and range of model parameters (see description of initializer inputs above).

**xInit** Initial value(s) or guess of model parameters. This is a pandas dataframe where columns refer (and are named according) to parameters of the model, and rows are initial values for the solver.

**y** Observations or measurements (see description of initializer inputs above).

**model** Pointer to model or model wrapper (see description of initializer inputs above).

**params** Other parameters for the calibration method. This argument is a Python dictionary.

Inputs to calibration functions must not be modified by the calibration function, or by any of the subroutines called internally by the function).

### 6.3.4 Outputs

#### Initializers

Each initializer returns a single output. The output is a pandas dataframe containing one or more sets of parameters (i.e., the initial values) that can be used as inputs for a calibration method. Ideally, the output of an initializer either characterizes a reduced parameter space, or provides an initial point (or set of points) for faster convergence of calibration methods. Each column of the output dataframe contains the values for a parameter (the names of the columns are indicated by `xInfo["name"]`). Each row contains the values of a set of parameters for initialization of a calibration method.

#### Calibration solvers

Each calibration solver returns a single output. The output is a pandas dataframe. Columns of this dataframe are:

- Parameters of the model that render the solution. There is one column per parameter. Names of these columns are defined by `xInfo["name"]`.
- Distance (or cost) metric. This is a column with the distance metric obtained for each set of parameters.
- Any other additional column, as defined by the calibration method.

Rows in the output dataframe are solutions to the calibration problem. Rows are sorted in decreasing order (from best to worst). Note that, depending on the calibration method, the output can contain more than one solution.

### 6.4 Test

The *test* directory contains a battery of tests for validating numerical and/or computational performance of library components.



### 6.4.1 File structure

This directory is organized in subdirectories, with each subdirectory containing one or more stand-alone tests (one test per file in the subdirectory).

**Naming conventions.** Subdirectories should be named as follows:

```
[type of test]_[main method/algorithm][_ other labels/description],
```

with:

- `type of test` assigning a category to the example. In general, categories refer to the main operation that is being verified in the test. Possible types of tests are:
  - `buildTree`
  - `calibration_historyMatching_sir`
  - `computeDegree`
  - `features`
  - `sampling`

Note that a test may use components from more than one module of the library. The type of test describes the *main* component that is being tested. It *does not* list all the modules/components that are being used in the test.

- `main method/algorithm` describing the main method or function that is being used in the test. For example, it can be the name of the initializer of a calibration method, or the name of a model contained in the `models` module, or a sampling strategy contained in the `sampling` module.
- `other labels/description` being any other string that completes the description of the test.

**Test functions.** There must be exactly one test per file. The entry point of the file (i.e., the test function) is given by the name of the file and the suffix `_test`. This function should not receive any parameters and it must be self-contained. All the code relevant to a given test should be inside the corresponding test file (this requirement aids in defining tests that are short and self-contained).

Each test must return a boolean state indicating if the test was successful or if it failed.

### 6.4.2 phylomodels test script

The file `phylomodels_test.py` contains a script that executes all the tests necessary for validating the correct installation/operation of the library. Calls to any test that is considered important for library validation should be added to this file.

## 6.5 Examples

Examples of configuration and use of modules can be found in the `examples` directory. This directory is organized in subdirectories, with each subdirectory containing an independent (i.e, stand-alone) example. Examples aid in understanding the use of different modules and, at the same time, they serve as templates for configuring and executing modeling and calibration jobs.

## 6.5.1 Naming conventions

Example subdirectories should be named as follows:

```
[type of example]_[main method/algorithm]_[other labels/description]
```

**type of example** Assigns a category to the example. In general, categories refer to the main operation that is being illustrated in the example. Possible types of examples are:

- calibration
- features
- model
- network
- sampling
- visualization

Note that one example may use components from more than one module. The type of example describes the *main* component that is being used. It *does not* list all the modules that are being used in the example.

**main method/algorithm** Describes the main method or function that is being used in the example. For example, it can be the name of the initializer of a calibration method, or the name of a model contained in the `models` module, or a sampling strategy contained in the `sampling` module.

**other labels/description** Any other string that completes the description of the example.

## 6.5.2 File structure

All the files relevant to a given example (that are not part of a module) should be located inside the corresponding folder. The name of the main script (and notebook, if available) of an example should be the same used for naming the example subdirectory.

## 6.5.3 Data

Data that may be used in more than one example should be placed in the `examples/data` subdirectory.

## 6.6 API reference

### 6.6.1 phylomodels package

Subpackages

`phylomodels.calibration` package

Submodules

`phylomodels.calibration.cal_parameterSweep` module

`phylomodels.calibration.constants` module

phylomodels.calibration.init\_historyMatching\_base module

phylomodels.calibration.init\_historyMatching\_poissonGlmBasis module

phylomodels.examples package

phylomodels.features package

Subpackages

phylomodels.features.series package

Submodules

phylomodels.features.series.series module

phylomodels.features.series.series\_derivative module

phylomodels.features.series.series\_derivative2 module

phylomodels.features.series.series\_derivative2\_cauchyFit module

phylomodels.features.series.series\_derivative2\_gaussianFit module

phylomodels.features.series.series\_derivative2\_laplaceFit module

phylomodels.features.series.series\_derivative\_cauchyFit module

phylomodels.features.series.series\_derivative\_gaussianFit module

phylomodels.features.series.series\_derivative\_laplaceFit module

phylomodels.features.series.series\_diff module

phylomodels.features.series.series\_diff\_L1 module

phylomodels.features.series.series\_diff\_L2 module

phylomodels.features.series.series\_diff\_Linf module

phylomodels.features.series.series\_log10 module

phylomodels.features.series.series\_partialSum10 module

phylomodels.features.series.series\_partialSum15 module

phylomodels.features.series.series\_partialSum2 module

phylomodels.features.series.series\_partialSum30 module

phylomodels.features.series.series\_partialSum7 module

phylomodels.features.series.series\_sum module

phylomodels.features.series.series\_sum\_log10 module

phylomodels.features.statistics package

Submodules

phylomodels.features.statistics.fano module

phylomodels.features.statistics.mean module

phylomodels.features.statistics.qcd module

phylomodels.features.statistics.rsd module

phylomodels.features.statistics.skew module

phylomodels.features.statistics.std module

phylomodels.features.statistics.var module

phylomodels.features.trees package

Subpackages

phylomodels.features.trees.helper package

Submodules

phylomodels.features.trees.helper.calculate\_branch\_length\_metrics module

phylomodels.features.trees.helper.check\_values module

phylomodels.features.trees.helper.get\_LTT module

phylomodels.features.trees.helper.get\_adjacency\_mat module

phylomodels.features.trees.helper.get\_distance\_mat module

phylomodels.features.trees.helper.get\_eigenvalues\_adj module

phylomodels.features.trees.helper.get\_eigenvalues\_dist\_lap module

phylomodels.features.trees.helper.get\_eigenvalues\_lap module

phylomodels.features.trees.helper.get\_groups module

phylomodels.features.trees.helper.get\_node\_properties module

phylomodels.features.trees.helper.process\_optional\_arguements module

phylomodels.features.trees.helper.unique\_node\_attr module

#### Submodules

phylomodels.features.trees.BL\_calculate\_external\_max module

phylomodels.features.trees.BL\_calculate\_external\_mean module

phylomodels.features.trees.BL\_calculate\_external\_median module

phylomodels.features.trees.BL\_calculate\_external\_min module

phylomodels.features.trees.BL\_calculate\_external\_std module

phylomodels.features.trees.BL\_calculate\_internal\_max module

phylomodels.features.trees.BL\_calculate\_internal\_mean module

phylomodels.features.trees.BL\_calculate\_internal\_median module

phylomodels.features.trees.BL\_calculate\_internal\_min module

phylomodels.features.trees.BL\_calculate\_internal\_std module

phylomodels.features.trees.BL\_calculate\_max module

phylomodels.features.trees.BL\_calculate\_mean module

phylomodels.features.trees.BL\_calculate\_median module

phylomodels.features.trees.BL\_calculate\_min module

phylomodels.features.trees.BL\_calculate\_ratio\_max module

**phylomodels.features.trees.BL\_calculate\_ratio\_mean** module  
**phylomodels.features.trees.BL\_calculate\_ratio\_median** module  
**phylomodels.features.trees.BL\_calculate\_ratio\_min** module  
**phylomodels.features.trees.BL\_calculate\_ratio\_std** module  
**phylomodels.features.trees.BL\_calculate\_std** module  
**phylomodels.features.trees.LTT\_calculate\_max\_lineages** module  
**phylomodels.features.trees.LTT\_calculate\_mean\_b\_time** module  
**phylomodels.features.trees.LTT\_calculate\_mean\_s\_time** module  
**phylomodels.features.trees.LTT\_calculate\_slope\_1** module  
**phylomodels.features.trees.LTT\_calculate\_slope\_2** module  
**phylomodels.features.trees.LTT\_calculate\_slope\_ratio** module  
**phylomodels.features.trees.LTT\_calculate\_t\_max\_lineages** module  
**phylomodels.features.trees.calculate\_external\_branch\_length\_metrics** module  
**phylomodels.features.trees.group\_calculate\_max** module  
**phylomodels.features.trees.group\_calculate\_mean** module  
**phylomodels.features.trees.group\_calculate\_median** module  
**phylomodels.features.trees.group\_calculate\_min** module  
**phylomodels.features.trees.group\_calculate\_mode** module  
**phylomodels.features.trees.group\_calculate\_num** module  
**phylomodels.features.trees.group\_calculate\_std** module  
**phylomodels.features.trees.local\_calculate\_LBI\_mean** module  
**phylomodels.features.trees.local\_calculate\_frac\_basal** module  
**phylomodels.features.trees.mean\_NN\_distance** module

phylomodels.features.trees.netSci\_calculate\_betweenness\_max module  
 phylomodels.features.trees.netSci\_calculate\_closeness\_max module  
 phylomodels.features.trees.netSci\_calculate\_diameter module  
 phylomodels.features.trees.netSci\_calculate\_eigen\_centrality\_max module  
 phylomodels.features.trees.netSci\_calculate\_mean\_path module  
 phylomodels.features.trees.smallConfig\_calculate\_cherries module  
 phylomodels.features.trees.smallConfig\_calculate\_double\_cherries module  
 phylomodels.features.trees.smallConfig\_calculate\_fourprong module  
 phylomodels.features.trees.smallConfig\_calculate\_pitchforks module  
 phylomodels.features.trees.spectral\_calculate\_eigen\_gap module  
 phylomodels.features.trees.spectral\_calculate\_kurtosis module  
 phylomodels.features.trees.spectral\_calculate\_max\_adj\_eigen module  
 phylomodels.features.trees.spectral\_calculate\_max\_distLap\_eigen module  
 phylomodels.features.trees.spectral\_calculate\_max\_lap\_eigen module  
 phylomodels.features.trees.spectral\_calculate\_min\_adj\_eigen module  
 phylomodels.features.trees.spectral\_calculate\_min\_lap\_eigen module  
 phylomodels.features.trees.spectral\_calculate\_skewness module  
 phylomodels.features.trees.top\_calculate\_B1 module  
 phylomodels.features.trees.top\_calculate\_B2 module  
 phylomodels.features.trees.top\_calculate\_FurnasR module  
 phylomodels.features.trees.top\_calculate\_WD\_ratio module  
 phylomodels.features.trees.top\_calculate\_colless module  
 phylomodels.features.trees.top\_calculate\_frac\_imbalance module

**phylomodels.features.trees.top\_calculate\_frac\_ladder module**

**phylomodels.features.trees.top\_calculate\_max\_dW module**

**phylomodels.features.trees.top\_calculate\_max\_ladder module**

**phylomodels.features.trees.top\_calculate\_mean\_imbalance\_ratio module**

**phylomodels.features.trees.top\_calculate\_sackin module**

**phylomodels.features.trees.top\_calculate\_sackin\_var module**

**phylomodels.features.trees.tree\_height\_calculate\_max module**

**phylomodels.features.trees.tree\_height\_calculate\_mean module**

**phylomodels.features.trees.tree\_height\_calculate\_min module**

**Submodules**

**phylomodels.features.features module**

**phylomodels.features.inventory module**

**phylomodels.features.test module**

**phylomodels.features.testFeatures module**

**phylomodels.models package**

**Subpackages**

**phylomodels.models.seir\_taoLeap package**

**Submodules**

**phylomodels.models.seir\_taoLeap.seirTaoLeap module**

**Submodules**

**phylomodels.models.distributions\_gaussian module**

**phylomodels.models.seir\_taoLeap\_getIncidenceSampled module**

**phylomodels.models.sir\_taoLeap\_getIncidence module**



phylomodels.models.sir\_taoLeap\_getIncidenceSampled module

phylomodels.models.test\_distributions\_gaussian module

phylomodels.models.test\_sir\_taoLeap\_getIncidence module

phylomodels.models.test\_sir\_taoLeap\_getIncidenceSampled module

phylomodels.network package

**Submodules**

phylomodels.network.phyloTree module

phylomodels.sampling package

**Submodules**

phylomodels.sampling.sample\_treeIndependent module

phylomodels.utilities package

phylomodels.visualization package

**Submodules**

phylomodels.visualization.parallelCoordinates module

## 6.7 To do

The following topic lists the functionality we plan to add to phyloModels in the future.

### 6.7.1 Calibration algorithms

#### Features and summary statistics

- Include multiple correlation coefficients as statistics for the evaluation of dispersion of summary statistics.
- Integrate summary statistics for trees.

### History matching

- Automatic selection of GLM basis. This can be done by running a quick test that characterizes the data to fit as one of the basis supported by History Matching (which are the basis supported by statsmodels).
- Use history matching diagnostics information for deciding if an iteration is successful or not (and make decisions regarding the need for repeating it using a different observation or summary statistic). NOTE: This depends on history matching reporting test information.
- Add support for using a variable number of model simulations per iteration, as well as guidelines (and functions) for selecting them.
- Add calibration example calling a model in R.
- Relax the condition that the number of observations must match the number of outputs rendered by the simulation model. The number of observations to use for feature generation should be the minimum between the number of observations provided as input argument and the number of outputs delivered by the simulation model.
- Incorporate means for robustly dealing with outliers and for capturing rare events.

### Parameter sweep

- Add support for additional cost metrics.

### Others

- Integrate other calibration methods: ELFI, ABC

## 6.7.2 Development

### General

- Document specifications for tests.
- Performance characterization for history matching.

### Features

- Update conventions for the name of outputs of summary statistics. It should be the name of the summary statistic (i.e., feature) followed by underscore “\_” and an integer number.
- Ensure parallelization for the computation of features. We should be able to take advantage of all available cores for computing features and statistics. Parallelization should be done on each feature or statistic independently.
- How can we optimize computations so that data can be shared/reused for the computation of multiple features or statistics? For example, `dx` is used in all `series_derivative` features.

## Visualization

- Superimpose marginal distributions in parallel coordinates plots.
- Superimpose reference curve in parallel coordinates plots.
- Add support for reverse/log axis in any coordinate in parallel coordinates plots (these options are currently supported only for the rightmost axes).

## Distribution

- Package in a Docker image.
- Make library available in Artifactory.