

# BppSuite Manual

---

Julien Dutheil, Laurent Guéguen  
[julien.dutheil@univ-montp2.fr](mailto:julien.dutheil@univ-montp2.fr)

---

1

This is the manual of the Bio++ Program Suite, version 0.4, 1.

Copyright © 2007, 2008, 2009, 2010, 2011 Bio++ development team

# Table of Contents

<b>1</b>	<b>Introduction</b> .....	<b>1</b>
<b>2</b>	<b>Syntax description</b> .....	<b>2</b>
2.1	Calling the programs and writing the option files. ....	2
2.2	Different types of options .....	3
2.3	Variables .....	3
<b>3</b>	<b>Common options encountered in several programs.</b> .....	<b>5</b>
3.1	Setting alphabet and reading sequences .....	5
3.2	Reading trees .....	6
3.3	Model specification .....	6
3.3.1	Setting up the substitution model .....	6
3.3.1.1	Nucleotide models .....	7
3.3.1.2	Protein models .....	7
3.3.1.3	Codon models .....	9
3.3.1.4	General multiple site models .....	12
3.3.1.5	Meta models .....	13
3.3.1.6	Mixture of models (beta feature) .....	14
3.3.1.7	Linking parameters .....	14
3.3.2	Setting up non-stationary / non-homogeneous models .....	14
3.3.2.1	One-per-branch non-homogeneous models .....	14
3.3.2.2	General non-homogeneous models .....	15
3.3.2.3	Root frequencies .....	15
3.3.3	Frequencies sets .....	16
3.3.4	Rate across site distribution .....	17
3.4	Discrete distributions .....	17
3.4.1	Standard Distributions .....	17
3.4.2	Mixture Distributions .....	18
3.5	Numerical parameters estimation .....	18
3.6	Writing sequences/alignments to files .....	19
<b>4</b>	<b>Bio++ Program Suite Reference</b> .....	<b>21</b>
4.1	BppML: Bio++ Maximum Likelihood .....	21
4.1.1	Branch lengths initial values .....	21
4.1.2	Topology optimization .....	21
4.1.3	Molecular clock .....	22
4.1.4	Output results .....	22
4.1.5	Bootstrap analysis .....	22
4.1.6	Rather technical options .....	22
4.2	BppSeqGen: Bio++ Sequence Simulator .....	23
4.3	BppAncestor: Bio++ Ancestral Sequence and Rate Reconstruction .....	23
4.4	BppDist: Bio++ Distance Methods .....	24
4.5	BppPars: Bio++ Maximum Parsimony .....	24
4.6	BppConsense: Bio++ Consensus Trees .....	25
4.7	BppPhySamp: Bio++ Phylogenetic Sampler .....	25
4.8	BppReroot: Bio++ Serial Tree Re-rooting .....	25
4.9	BppSeqMan: Bio++ Sequence Manipulation .....	26
4.10	BppTreeDraw: Bio++ Tree Drawing .....	27

# 1 Introduction

The Bio++ Program Suite is a package of programs using the Bio++ libraries and dedicated to Phylogenetics and Molecular Evolution. All programs are independent, but can be combined to perform rather complex analyses. These programs use the interface helper tools of the libraries, and hence share the same syntax. They also have several options in common, which may also be shared by third-party software. This manual was hence split into three parts:

*Bio++ option file syntax*

A general description of the language used to interact with the programs.

*Shared options*

A more detailed description about several options that are encountered in several programs. This includes input/output of data and model specifications.

*The Bio++ Program Suite reference*

Include a reference of all available options for each program in the package.

## 2 Syntax description

### 2.1 Calling the programs and writing the option files.

The programs in the Bio++ Program Suite are command line-driven. Arguments may be passed as `parameter=value` options, either directly to the command line or using an option file:

```
{program} parameter1=value1 parameter2=value2 ... parameterN=valueN
```

or

```
{program} param=option_file
```

where `{program}` is the name of the program to use (`bppml`, `bppseqgen`, etc.). Option files contain `parameter=value` lines, with only one parameter per line. They can be written from scratch using a regular text editor, but since these files can potentially turn to be quite complex, it is probably wiser to start with a sample provided along with the program (if any!).

Extra-space may be included between parameter names, equal sign and value:

```
first_parameter    = value1  
second_parameter  = value2
```

and lines can be broken using the backslash character:

```
parameter = value1,\  
           value2,\  
           value3
```

Comment may be included, in either scripting format:

```
# This is a comment
```

C format:

```
/* This is a comment  
*/
```

or C++ format:

```
// This is a comment
```

Command line and file options may be combined:

```
{program} param=option_file parameterX=valueX
```

In case `parameterX` is specified in both option file and command line, the command line value will be used. This allows to run the programs several times by changing a single option, like the name of the data set for instance.

Option files can be nested, by using `param=nestedoptionfile` within an option file, as with the command line. It is possible to use this option as often as needed, this will load all the required option files.

## 2.2 Different types of options

The next chapters describe the whole set of options available in BppSuite. For each parameter, the type of parameter value expected is defined as:

`{chars}` A character chain

`{path}` A file path, which may be absolute or related to the current directory

`{int}` An integer

`{int}`, `{int>0}`, `{int>=0}`, `{int[2,10]}`

An integer, a positive integer, a positive non-null integer, an integer falling between 2 and 10

`{real}`, `{real>0}`, etc

A real number, a positive real number, etc.

`{boolean}`

A Boolean value, which may be one of 'yes', 'no', 'true' or 'false'

`{xxx|yyy|zzz}`

A set of allowed values

`{list<type>}`

A list of values of specified type, separated by comas.

If an option availability or choice depends on another parameters, it will be noted as

```
parameter1={xxx|yyy|zzz}
```

```
parameter2={chars} [[parameter1=zzz]]
```

meaning that parameter2 is available only if parameter1 is set to 'zzz'.

Any optional argument will be noted within hooks [].

In some cases, the argument value is more complexe and follows the 'keyval' syntax. This syntax will be quite familiar for users using languages like R, Python, or certain LaTeX packages. A keyval procedure is a name that does no contain any space, together with some arguments within parentheses. The arguments take the form `key=value`, separated by comas:

```
parameter=Function(name1=value1, name2=value2)
```

Space characters are allowed around the '=' and ',' punctuations.

## 2.3 Variables

It is possible to recall anywhere the value of an option by using `$(parameter)`.

```
optimization.topology.algorithm = NNI
optimization.topology.algorithm_nni.method = phycl
output.tree = MyData_$(optimization.topology.algorithm)_$(optimization.topology.algorithm_nni.method)
```

You can use this syntax to define global variables:

```
data=MyData
sequence.file=$(data).fasta
input.tree=$(data).dnd
output.infos=$(data).infos
```

Important note: it is not possible to use a macro with the 'param' option. This is because all nested option files are parsed before the variable resolution. Writing `param=$(model1).bpp` will not work, but this allows the user to override variables in nested files, as with the command line. For instance:

```
#Option file 1:
param=options2.bpp
sequence.file=$(data).fasta
sequence.format=Fasta
```

```
#Option file 2:
data=LSU
#etc
```

## 3 Common options encountered in several programs.

### 3.1 Setting alphabet and reading sequences

`alphabet =`

`{DNA|RNA|Protein|Word(letter={DNA|RNA|Protein},length={int})|Codon(letter={DNA|RNA},type={Standard|EchinodermMitochondrial|InvertebrateMitochondrial|VertebrateMitochondrial})}` The alphabet to use when reading sequences.

`input.sequence.file={path}`

The sequence file to use. Depending on the program, these sequences have or do not have to be aligned.

`input.sequence.format = {sequence format description}`

The sequence file format.

Since Bio++ Program Suite version 0.4.0, the format description uses the keyval syntax. The format is a function, with optional parameters:

`Fasta()` The fasta format.

`Mase(siteSelection={chars})`

The Mase format (as read by Seaview and Phylo\_win for instance), with an optional site selection name.

`Phylip(order={interleaved|sequential}, type={classic|extended}, split={spaces|tab})`

The Phylip format, with several variations. The argument `order` distinguishes between sequential and interleaved format, while the option `type` distinguished between the plain old Phylip format and the more recent extension allowing for sequence names longer than 10 characters, as understood by PAML and PhyML. Finally, the `split` argument specifies the type of character that separates the sequence name from the sequence content. The conventional option is to use one (classic) or more (extended) spaces, but tabs can also be used instead.

`Clustal(extraSpaces={int})`

The Clustal format. In its basic set up, sequence names do not have space characters, and one space splits the sequence content from its name. The parser can however be configured to allow for spaces in the sequence names, providing a minimum number of space characters is used to split the content from the name. Setting `extraSpaces` to 5 for instance, the sequences are expected to be at least 6 spaces away for their names.

`Dcse()` The DCSE alignment format. The secondary structure annotation will be ignored.

`Nexus()` The Nexus alignment format. Only very basic support is provided.

For programs that do not require the sequences to be aligned, the following formats are also available:

`GenBank()`

Very basic support: only retrieves the sequence content for now, all features are ignored.

Basic operations can be performed on the sequences:

`input.sequence.sites_to_use = {all|nogap|complete}`

This option only works if the program requires an alignment. Tells which sites to use. The 'nogap' option removes all sites containing at least one gap, and the 'complete'

option removes all sites containing at least one gap or one generic character, as 'X' for instance.

```
input.sequence.remove_stop_codons = {boolean}
```

This option only works if the alphabet is a codon alphabet. Removes the sites where there is a stop codon (default: 'yes').

```
input.sequence.max_gap_allowed=100%
```

This option only works if the program requires an alignment. Only works when the 'all' option is selected. It specifies the maximum amount of gap allowed per site, as a number of sequence or a percentage. Sites not matching the criterion will not be included in the analysis, but the original site numbering will be used in the output files (if relevant).

## 3.2 Reading trees

```
input.tree.file = {path}
```

The phylogenetic tree file to use.

```
input.tree.format = {Newick|Nexus}
```

The format of the input tree file.

Some programs may require that your file contains several trees. The corresponding options are then:

```
input.trees.file = {path}
```

The file containing multiple trees.

```
input.trees.format = {Newick|Nexus}
```

The format of the input tree file.

## 3.3 Model specification

The substitution model specification over the tree is set up in different parts.

```
nonhomogeneous = {no|one_per_branch|general}
```

Set the type of model. The 'no' option is used for homogeneous models. The 'one\_per\_branch' option is used as a short cut for setting branch models (for instance Galtier and Gouy 97 for branch GC content, or PAML branch model), and the 'general' option for the more general case, including PAML clade models. In either of the last two cases, the model is potentially non-stationary, that is, possibly not at the equilibrium and hence includes the root frequencies as additional parameters. If the substitution model is not the same across the tree, then the model is also non-homogeneous.

In combination with those models, one can also specify a distribution of site-specific rate.

### 3.3.1 Setting up the substitution model

```
model = {model description}
```

A description of the substitution model to use, using the keyval syntax.

From version 0.4.0 of BppSuite, the model specification uses a completely new syntax, based on the keyval (key = value) scheme. The old option files will hence not be compatible with new version of the software. The new syntax however is hopefully more intuitive, and more generalizable, so that few changes are expected when new models will be built. The substitution model is a function, potentially including parameters. The following table lists the set of usable functions, and their parameters.

### 3.3.1.1 Nucleotide models

JC69 The Jukes and Cantor model. This model has no additional parameter.

K80(*kappa*={real>0})

The Kimura 2 parameters model. *kappa* is the transition over transversion ratio.

F84(*kappa*={real>0}, *theta*={real]0,1[]}, *theta1*={real]0,1[]}, *theta2*={real]0,1[]}, *useObservedFreqs*={boolean}, *useObservedFreqs.pseudoCount*={int>0})

Felsenstein's 1984 substitution model, with transition/transversion ratio and 4 distinct equilibrium frequencies, set using three independent parameters: *theta* is the GC content, *theta1* is the proportion of G / (G + C) and *theta2* is the proportion of A / (A + T or U). The *useObservedFreqs* option set the *thetas* parameters according to the observed counts in the data set, and *useObservedFreqs.pseudoCount* is a quantity, defaulting to 0, that can be used in case some counts are zero, on small data sets for instance. The corrected values are computed as:

$$\pi_i = \frac{f_i + \psi}{4 \cdot \psi + \sum_j f_j}$$

HKY85(*kappa*={real>0}, *theta*={real]0,1[]}, *theta1*={real]0,1[]}, *theta2*={real]0,1[]}, *useObservedFreqs*={boolean}, *useObservedFreqs.pseudoCount*={int>0})

Hasegawa, Kishino and Yano 1985's substitution model. The model is similar to F84, but with a different implementation. The *kappa* parameter used here is comparable to the one in K80.

T92(*kappa*={real>0}, *theta*={real]0,1[]}, *useObservedFreqs*={boolean}, *useObservedFreqs.pseudoCount*={int>0})

Tamura 1992's model for nucleotides, similar to HKY85, yet assuming that the frequencies of A = T/U and G = C.

TN93(*kappa1*={real>0}, *kappa2*={real>0}, *theta*={real]0,1[]}, *theta1*={real]0,1[]}, *theta2*={real]0,1[]}, *useObservedFreqs*={boolean}, *useObservedFreqs.pseudoCount*={int>0})

Tamura and Nei 1993's model, similar to HKY85, but allowing for two distinct transition/transversion ratios.

GTR(*a*={real>0}, *b*={real>0}, *c*={real>0}, *d*={real>0}, *e*={real>0}, *theta*={real]0,1[]}, *theta1*={real]0,1[]}, *theta2*={real]0,1[]}, *useObservedFreqs*={boolean}, *useObservedFreqs.pseudoCount*={int>0})

The General Time-Reversible substitution model. Parameters *a*, *b*, *c*, *d*, *e* are the entries of the exchangeability matrix.

L95(*beta*={real>0}, *gamma*={real>0}, *delta*={real>0}, *theta*={real]0,1[]}, *theta1*={real]0,1[]}, *theta2*={real]0,1[]}, *useObservedFreqs*={boolean}, *useObservedFreqs.pseudoCount*={int>0})

The strand-symmetric model of Lobry 1995, for nucleotides.

### 3.3.1.2 Protein models

JC69 The Jukes and Cantor model. This model has no additional parameter.

DS078 Protein substitution model, using the dcmutt implementation of Kosiol and Goldman 2005.

JTT92 Protein substitution model, using the dcmutt implementation of Kosiol and Goldman 2005.

- WAG01** Protein substitution model, from Whelan & Goldman 2001.
- LG08** Protein substitution model, from Le & Gascuel 2008.
- LLG08\_EX2**(relrate1={real}0,1[], relproba1={real}0,1[])  
Protein substitution model, from Le, Lartillot & Gascuel 2008. See the meaning of the variables in the Mixture model below.
- LLG08\_EX3**(relrate1={real}0,1[], relrate2={real}0,1[], relproba1={real}0,1[], relproba2={real}0,1[])  
Protein substitution model, from Le, Lartillot & Gascuel 2008. See the meaning of the variables in the Mixture model below.
- LLG08\_EH0**(relrate1={real}0,1[], relrate2={real}0,1[], relproba1={real}0,1[], relproba2={real}0,1[])  
Protein substitution model, from Le, Lartillot & Gascuel 2008. See the meaning of the variables in the Mixture model below.
- LLG08\_UL2**(relrate1={real}0,1[], relproba1={real}0,1[])  
Protein substitution model, from Le, Lartillot & Gascuel 2008. See the meaning of the variables in the Mixture model below.
- LLG08\_UL3**(relrate1={real}0,1[], relrate2={real}0,1[], relproba1={real}0,1[], relproba2={real}0,1[])  
Protein substitution model, from Le, Lartillot & Gascuel 2008. See the meaning of the variables in the Mixture model below.
- DS078+F**(theta={real}0,1[], theta1={real}0,1[], theta2={real}0,1[], ..., useObservedFreqs={boolean}, useObservedFreqs.pseudoCount={int>0})  
Protein substitution model, using the dcmutt implementation of Kosiol and Goldman 2005 and free equilibrium frequencies. The *thetaX* are frequencies parameters, where X is 1 to 19. Parameter *theta1* is the proportion of A, *theta2* is the proportion of R over (1-A), *theta3* the proportion of N over (1-A-R), etc.
- JTT92+F**(theta={real}0,1[], theta1={real}0,1[], theta2={real}0,1[], ..., useObservedFreqs={boolean}, useObservedFreqs.pseudoCount={int>0})  
Protein substitution model, using the dcmutt implementation of Kosiol and Goldman 2005 and free equilibrium frequencies.
- WAG01+F**(theta={real}0,1[], theta1={real}0,1[], theta2={real}0,1[], ..., useObservedFreqs={boolean}, useObservedFreqs.pseudoCount={int>0})  
Protein substitution model, from Whelan & Goldman 2001, and free equilibrium frequencies.
- LG08+F**(theta={real}0,1[], theta1={real}0,1[], theta2={real}0,1[], ..., useObservedFreqs={boolean}, useObservedFreqs.pseudoCount={int>0})  
Protein substitution model, from Le & Gascuel 2008, and free equilibrium frequencies.
- Empirical**(file={path})  
Build a protein substitution model from a file in PAML format.
- Empirical+F**(file={path}, theta={real}0,1[], theta1={real}0,1[], theta2={real}0,1[], ..., useObservedFreqs={boolean}, useObservedFreqs.pseudoCount={int>0})  
Build a protein substitution model from a file in PAML format, and use free equilibrium frequencies.

### 3.3.1.3 Codon models

Standard codon models: the optional *genetic\_code* argument describes the genetic code. If it is not given, the one related with the alphabet is used. The several values available are described below.

- EchinodermMitochondrialGeneticCode
- InvertebrateMitochondrialGeneticCode
- StandardGeneticCode
- VertebrateMitochondrialGeneticCode
- YeastMitochondrialGeneticCode

The next codon models also take as argument a *codon\_freq* specifying the equilibrium frequencies of the model. The syntax refers to the one used in the PAML software, and possible values are

- F0: all frequencies are assumed to be fixed and equal to 1/61, 0 for stop codons.
- F1X4: 4 distinct frequencies are used, with parameters *theta*, *theta1*, *theta2* (See [\[Frequencies sets\]](#), page 16, “Full” method).
- F3X4: 4 distinct frequencies are used for each position, resulting in 9 parameters in total (3 independent “Full” frequencies set).
- F61: free equilibrium frequencies, stop codons set to 0.

The same words can be used to specify root frequencies for codon models, in the case of non reversibility.

Otherwise, the *frequencies={frequencies set description}* argument can as well be used.

GY94([genetic\_code={genetic code description} , kappa={real>0}, V={real>0}])  
Goldman and Yang (1994) substitution model for codons (default values: *kappa*=1 and *V*=10000).

MG94([genetic\_code={genetic code description}, rho={real>0}])  
Muse and Gaut (1994) substitution model for codons (default values: *rho*=1).

YN98([genetic\_code={genetic code description}, kappa={real>0}, omega={real>0}])  
Yang and Nielsen (1998) substitution model for codons (default values: *kappa*=1 and *omega*=1).

YNGKP\_M0([genetic\_code={genetic code description}, kappa={real>0}, omega={real>0}])  
The M0 model of PAML, ie the same as YN98.

YNGKP\_M1([genetic\_code={genetic code description}, kappa={real>0}, omega={real>0}, p0={real>0 and <1 }])  
The M1a model of PAML, see Yang, Z., R. Nielsen, N. Goldman, and A.-M. K. Pedersen (2000) (default values: *kappa*=1, *p0*=0.5, *omega*=0.5).

YNGKP\_M2([genetic\_code={genetic code description}, kappa={real>0}, omega0={real>0 and <1}, theta1={real>0 and <1 }, omega1={real>1}, theta2={real>0 and <1 }])  
The M2a model of PAML, see Yang, Z., R. Nielsen, N. Goldman, and A.-M. K. Pedersen (2000), with  $p_0 = \theta_1$  and  $p_1 = (1 - \theta_1) * \theta_2$  (default values: *kappa*=1, *theta1*=0.33333, *theta2*=0.5, *omega0*=0.5, *omega2*=0.5).

YNGKP\_M3(genetic\_code={genetic code description}, n={integer>0}, kappa={real>0}, omega0={real>0 and <1}, delta1={real>0}, ..., deltan-1={real>0}, theta1={real>0 and <1 }, ..., thetan-1={real>0 and <1 }])  
The M3 model of PAML, see Yang, Z., R. Nielsen, N. Goldman, and A.-M. K. Pedersen (2000), with *n* discrete values, with  $p_0 = \theta_1$  and  $p_k = (1 - \theta_1) * \dots * (1 -$

$\text{thetak} * \theta_{k+1}$ , and  $\text{omegak} = \omega_0 + \delta_1 + \dots + \delta_k$  (default values:  $n=3$ ,  $\text{kappa}=1$ ,  $\text{thetak}=1/(n-k+1)$ ,  $\text{omega0}=0.5$ ,  $\text{deltak}=0.5$ ).

```
YNGKP_M7(n={integer>0}, [genetic_code={genetic code description},kappa={real>0},
p={real>1}, q={real>1 }])
```

The M7 model of PAML, see Yang, Z., R. Nielsen, N. Goldman, and A.-M. K. Pedersen (2000), with the Beta distribution discretized in  $n$  classes (default values:  $\text{kappa}=1$ ,  $p=2$ ,  $q=2$ ).

```
YNGKP_M7(n={integer>0}, [genetic_code={genetic code description},kappa={real>0},
omegas={real>1}, p0={real>0},p={real>1}, q={real>1 }])
```

The M8 model of PAML, see Yang, Z., R. Nielsen, N. Goldman, and A.-M. K. Pedersen (2000), with the Beta distribution discretized in  $n$  classes (default values:  $\text{kappa}=1$ ,  $p=2$ ,  $q=2$ ,  $p0=0.5$ ,  $\text{omegas}=2$ ).

It is also possible to setup more specific models, by specifying a nucleotide model for each position. Model parameters names then take the form of `CodonModel.<position set>_<position model name>.<position specific parameter name>`.

```
CodonNeutral(model={model name} [, relrate1={real>0}, relrate2={real>0}])
or
```

```
CodonNeutral(model1={model name}, model2={model name}, model3={model name}[,
relrate1={real>0}, relrate2={real>0}])
```

Substitution model on codons. The arguments *model* and *model{i}* are for descriptions of models on bases. The alphabet must be a codon alphabet.

If the argument is *model*, the *same* single site model is used on all positions (ie the parameters are shared between all positions).

If the arguments are *model1*, *model2*, *model3*, each single site model stands for a single-site substitution model. In that case, all single site models parameters are position dependent.

Each single site model is normalized and the substitution rates between codons that differ on more than one letter are null.

Arguments *relrate{i}* stands for the relative substitution rates of the sites. Default:  $\text{relrate}\{i\} = 1/(4-i)$ , such that the rate of each site is  $1/3$ .

The generator is first computed with these models and parameters on the whole triplet alphabet, and then the substitution rates to and from stop codons are set to zero and the generator is normalized with this modification.

```
alphabet=Codon(letter=DNA, type=Standard)
model=CodonNeutral(model=T92)
```

builds a model on codons, such all sites follow the same T92 model. The parameters names are *CodonNeutral.123.T92.kappa*, *CodonNeutral.relrate1*, *CodonNeutral.relrate2*.

```
alphabet=Codon(letter=DNA, type=Standard)
model=CodonNeutral(model1=T92, model2=T92, model3=JC69)
```

builds a model on codons, such that first and second sites follow independent T92 models, and third site follows a JC69 model. Then the parameters names are *CodonNeutral.1.T92.kappa*, *CodonNeutral.2.T92.kappa*, *CodonNeutral.relrate1*, *CodonNeutral.relrate2*, and can be initialized as is:

```
model=CodonNeutral(model1=T92, model2=T92, model3=JC69,\
1_T92.theta=0.5, 1_T92.kappa=2.0, 2_T92.theta=0.4, 2_T92.kappa=2.0)■
```

```
CodonAsynonymous(model={model name}[, genetic_code={genetic code description},
beta={real>0}])
```

or

```
CodonAsynonymous(model1={model name}, model2={model name}, model3={model name}[,
geneticcode={genetic code description}, beta={real>0}])
```

substitution model on codons.

The arguments *model* and *model*{*i*} are for descriptions of models on bases. The alphabet must be a codon alphabet.

If the argument is *model*, the *same* single site model is used on all positions (ie the parameters are shared between all positions).

If the arguments are *model1*, *model2*, *model3*, each single site model stands for a single-site substitution model. In that case, all single site models parameters are position dependent.

Each single site model is normalized and the substitution rates between codons that differ on more than one letter are null.

In addition to these models, the optional *geneticcode* argument describes the genetic code. If it is not given, the one related with the alphabet is used. The several values available are described below.

Optional argument *beta* is the ratio between non-synonymous substitution rate and synonymous substitution rate. Default value: 1.

```
alphabet=Codon(letter=DNA, type=Standard)
model=CodonAsynonymous(model=T92)
```

builds a model on codons, such all sites follow the same T92 model. The parameters names are *CodonAsynonymous.123\_T92.kappa* and *CodonAsynonymous.beta*.

```
alphabet=Codon(letter=DNA, type=Standard)
model=CodonNeutral(model1=T92, model2=T92, model3=JC69)
```

builds a model on codons, such that first and second sites follow independent T92 models, and third site follows a JC69 model. Then the parameters names are *CodonAsynonymous.1\_T92.kappa*, *CodonAsynonymous.2\_T92.kappa*, *CodonAsynonymous.beta*.

```
CodonNeutralFrequencies(frequencies={frequencies set description} [,
retrate1={real>0}, retrate2={real>0}])
```

substitution model on codons. The exchangeability model on each site is the same K80 model, and the equilibrium distribution of the model is description by the *frequencies* argument. See the description of the Frequencies Set below.

Each single site model is normalized and the substitution rates between codons that differ on more than one letter are null.

Arguments *retrate*{*i*} stands for the relative substitution rates of the sites. Default: *retrate*{*i*}=1/{4-*i*}, such that the rate of each site is 1/3.

The generator is first computed with these model and distribution on the whole triplet alphabet, and then the substitution rates to and from stop codons are set to zero and the generator is normalized with this modification.

```
alphabet=Codon(letter=DNA, type=Standard)
model=CodonNeutralFrequencies(frequencies=Full())
```

has parameters *CodonNeutralFrequencies.123\_K80.kappa*, *CodonNeutralFrequencies.Full.theta\_1*, ..., *CodonNeutralFrequencies.Full.theta\_60*, *CodonNeutralFrequencies.retrate1*, *CodonNeutralFrequencies.retrate2*.

```
CodonAsynonymousFrequencies(frequencies={frequencies set description}
[geneticcode={genetic code description}, beta={real>0}])
```

substitution model on codons. The exchangeability model on each site is the same K80 model, and the equilibrium distribution of the model is description by the *frequencies* argument. See the description of the Frequencies Set below.

Each single site model is normalized and the substitution rates between codons that differ on more than one letter are null.

In addition to these models, the optional *geneticcode* argument describes the genetic code. If it is not given, the one related with the alphabet is used. The several values available are described below.

Optional argument *beta* is the ratio between non-synonymous substitution rate and synonymous substitution rate. Default value: 1.

The generator is first computed with these model and distribution on the whole triplet alphabet, and then the substitution rates to and from stop codons are set to zero and the generator is normalized with this modification.

```
alphabet=Codon(letter=DNA, type=Standard)
model=CodonAsynonymousFrequencies(frequencies=Full())
```

has parameters *CodonAsynonymousFrequencies.012\_T92.kappa*, *CodonAsynonymousFrequencies.Full.theta\_1*, ..., *CodonAsynonymousFrequencies.Full.theta\_60*, *CodonAsynonymousFrequencies.beta*.

### 3.3.1.4 General multiple site models

```
Word(model={model name} [,relrate1={1>real>0}, ..., relrate{n-1}={1>real>0}])
or
```

```
Word(model1={model name}, model1={model name}, ..., modeln={model name}[,
relrate1={1> real>0}, ..., relrate{n-1}={1> real>0}])
```

substitution model on words. The arguments *model* and *model{i}* are for descriptions of models on single sites such as nucleotides or proteins. The alphabet must be a Word alphabet.

If the argument is *model*, the length of the words in the substitution model is determined by the length of the words in the alphabet, and the *same* single site model is used (ie the parameters are shared between all positions).

If the arguments are *model1*, ..., *model{n}*, the length of the words in the alphabet must be *n*, and each single site model stands for a single-site substitution model. In that case, all single site models parameters are position dependent.

Each single site model is normalized and the substitution rates between words that differ on more than one letter are null.

Arguments *relrate{i}* stands for the relative substitution rates of the sites. Default:  $relrate\{i\}=1/\{n-i+1\}$ , such that the rate of each site is  $1/n$ .

```
alphabet=Word(letter=DNA,length=4)
model=Word(model=T92())
```

builds a model on 4 bases words, such all sites follow the same T92 model. The parameters names are *Word.1234\_T92.kappa*, *Word.relrate1*, *Word.relrate2*, *Word.relrate3*.

```
alphabet=Word(letter=DNA,length=4)
model=Word(model1=T92(), model2=T92(), model3=JC69(), model4=HKY85())■
```

builds a model on 4 bases words, such first and second sites follow independent T92 models, third site follows a JC69 model, and fourth site follows a HKY85

model. Then the parameters names are *Word.1\_T92.kappa*, *Word.2\_T92.kappa*, *Word.4\_HKY85.kappa*, *Word.4\_HKY85.theta*, *Word.4\_HKY85.theta1*, *Word.4\_HKY85.theta2*, *Word.relrates1*, *Word.relrates2*, *Word.relrates3*.

```
Triplet(model={model description} [, relrate1={real>0}, relrate2={real>0}])
or
```

```
Triplet(model1={model description}, model2={model description}, model3={model
description}[, relrate1={real>0}, relrate2={real>0}])
```

substitution model on 3 letters words. The arguments *model* and *model{i}* are for descriptions of models on single sites such as nucleotides or proteins. The alphabet must be a 3-letters word alphabet or a codon alphabet.

If the argument is *model*, the *same* single site model is used on all positions (ie the parameters are shared between all positions).

If the arguments are *model1*, *model2*, *model3*, each single site model stands for a single-site substitution model. In that case, all single site models parameters are position dependent.

Each single site model is normalized and the substitution rates between triplets that differ on more than one letter are null.

Arguments *relrate{i}* stands for the relative substitution rates of the sites. Default: *relrate{i}=1/{4-i}*, such that the rate of each site is 1/3.

```
alphabet=Codon(letter=DNA, type=Standard)
model=Word(model=T92)
```

builds a model on codons, such all sites follow the same T92 model. The parameters names are *Triplet.123\_T92.kappa*, *Triplet.relrates1*, *Triplet.relrates2*.

```
alphabet=Word(letter=DNA, length=3)
model=Triplet(model1=T92, model2=T92, model3=JC69)
```

builds a model on 3 bases words, such first and second sites follow independent T92 models, and third site follows a JC69 model. Then the parameters names are *Triplet.1\_T92.kappa*, *Triplet.2\_T92.kappa*, *Triplet.relrates1*, *Triplet.relrates2*.

### 3.3.1.5 Meta models

These substitution models take as argument another substitution model, and add several parameters.

```
TS98(model={model description}, s1={real>0}, s2={real>0})
```

Tuffley and Steel 1998's 'covarion' model, taking a nested substitution model as argument for *model*. The nested model can be any substitution model for any alphabet.

```
G01(model={model description}, rdist={rate distribution description},
mu={real>0})
```

Galtier 2001's 'covarion' model, taking a nested substitution model as argument for *model* and a rate distribution for parameter *rdist* (see below). The nested model can be any substitution model for any alphabet.

```
RE08(model={model description}, lambda={real>0}, mu={real>0})
```

Rivas and Eddy 2008's substitution model with gaps, taking a nested substitution model as argument for *model*. Parameter *lambda* is the insertion rate, while *mu* is the deletion rate.

### 3.3.1.6 Mixture of models (beta feature)

Mixed models combine any substitution models with a priori distribution of parameters. Such models are still experimental and have not been yet fully tested. They should hence be used with extra care!

During the likelihood computation process, all the submodels of the mixture are successively applied on the branches, and the mean of all the likelihoods is computed. With nonhomogeneous reconstruction, since a mixed model is a random variable, affecting a mixed model to a set of branches means that all these branches are dependent, and in this case all the branches of the set have the same submodel at the same time.

`MixedModel(model={model description})`

Mixture model from a given *model* in which some parameters follow a probabilistic distribution. Any discrete distribution available can be used See [Discrete distributions], page 17. The description of the parameters distributions is described below.

```
model=MixedModel(model=TN93(kappa1=Gamma(n=4,alpha=3,beta=1),\
                             kappa2=Exponential(lambda=2),\
                             theta=0.5,theta1=0.2,theta2=0.1))
has parameters TN93.kappa1.Gamma.alpha, TN93.kappa1.Gamma.beta,
TN93.kappa2.Exponential.lamba, TN93.theta, MixedModel.TN93.theta1,
TN93.theta2.
```

`Mixture(model1={model description},...,modeln={model description} [, relrate1={1>real>0},...,relrate{n-1}={1>real>0},relproba1={1>real>0},...,relproba{n-1}={1>real>0}])`

Mixture model built from several *models*: each model has its own probability and rate.

Arguments *relproba*{*i*} stands for the relative probability and *relrate*{*i*} stands for the relative rate of each model (in the order the models are given). Default: *relproba*{*i*}= $1/(n-i+1)$ , such that the probability of each site is  $1/n$ , and *relrate*{*i*}= $1/(n-i+1)$  such that the rate of each site is 1.

```
model=Mixture(model1=GY94(), model2=YN98(), relrate1=0.1)
has parameters Mixture.relrate1, Mixture.relproba1, Mixture.1.GY94.kappa, Mix-
ture.1.GY94.V, Mixture.2.YN98.kappa, Mixture.2.YN98.omega.
```

### 3.3.1.7 Linking parameters

It is possible to reduce the parameter space by putting extra constraints on parameters, using for instance

```
model=TN93(kappa1=1.0, kappa2=kappa1, theta=0.5)
```

In that particular case the resulting model is strictly equivalent to the HKY85 model. This syntax however allows to define a larger set of models.

## 3.3.2 Setting up non-stationary / non-homogeneous models

You can specify a wide range of non-homogeneous models, by combining different options.

### 3.3.2.1 One-per-branch non-homogeneous models

This option share the same parameters as the homogeneous case, since the same kind of model is used for each branch. The additional options are the following:

`nonhomogeneous_one_per_branch.shared_parameters = {list<chars>}`

List the names of the parameters that are shared by all branches. In Galtier & Gouy model, that would be *T92.kappa*, since only the theta parameter is branch-specific.

The '\*' wildcard can be used as a suffix, as in `YN98.freq_Word.1_*` for all the parameters whose names start with `YN98.freq_Word.1_`.

### 3.3.2.2 General non-homogeneous models

Bio++ provides a general syntax to specify almost any non-homogeneous model.

```
nonhomogeneous.number_of_models = {int>0}
```

Set the number of distinct models to use.

You now have to configure each model individually, using the syntax introduced for the homogeneous case, excepted that model will be numbered, for instance:

```
model1 = T92(theta=0.39, kappa=2.79)
```

The additional option is available to attach the model to branches in the tree, specified by the id of the upper node in the tree:

```
model1.nodes_id = 1,5,10:15,19
```

Specify the ids of the nodes to which the node is attached. Id ranges can be specified using the 'begin:end' syntax.

You can also make a given model share parameters with another one by writing for instance:

```
model2 = T92(theta=0.39, kappa=model1.T92.kappa)
```

Please note the syntax, parameters are referred to as `[model name].[parameter name]` in that case. Only parameter from identical models can be aliased in this manner. To link parameters from different models, you have to use the more general option (warning, currently beta feature!)

```
nonhomogeneous.alias = {list of aliases}
```

where each alias is described as 'param1->param2'. The full name of the parameters have to be used, see for example:

```
model1 = T92(theta=0.4, kappa=4)
model2 = GTR(theta=0.4, a = 1.1, b=0.4, c=0.4, d=0.25, e=0.1)
nonhomogeneous.alias=GTR.theta_1->T92.theta_1
```

This option can be used to link parameters of the root frequencies if the model is non-stationary:

```
nonhomogeneous.root_freq=Full(init=balanced)
nonhomogeneous.alias=Full.theta->GTR.theta_1
```

Finally, you may find useful the following options:

```
output.tree_ids.file = {{path}|none}
```

A tree file in newick format, with node ids instead of bootstrap values, and leaf names with their id as suffix. The use of that option will cause the program to exit just after producing the tagged tree.

```
output.parameter_names.file = {{path}|none}
```

A text file listing all parameter names. This might come handy in order to specify the parameter that should not be optimized (see `optimization.ignore_parameter`) or aliased (see above). The use of that option will cause the program to exit just after producing the list file.

### 3.3.2.3 Root frequencies

In case of nonstationary models, the ancestral frequencies are distinct parameters. If a model is assumed to be stationary, the "None" parameter value can be used, which is strictly equivalent to setting `nonhomogeneous.stationary=yes`.

As since version 0.4.0, BppSuite uses the keyval syntax to set up root frequencies,

`nonhomogeneous.root_freq={frequency set description}`

The Frequencies set used can be any of the ones described below See [\[Frequencies sets\]](#), page 16, depending on the alphabet used.

### 3.3.3 Frequencies sets

The following frequencies distributions are available:

`Fixed()` All frequencies are fixed to their initial value and are not estimated.

`GC(theta={real}0,1[ ])`

For nucleotides only, set the G content equal to the C content.

`Full(theta1={real}0,1[ ], theta2={real}0,1[ ], ..., thetaN={real}0,1[ ])`

Full parametrization. Contains N free parameters, where N is equal to the size of the alphabet - 1. For codon models, N is the size of the alphabet - 1 - the number of stop codons, whose frequencies are set to 0. For nucleotide sequences, theta is the GC content, theta1 is the proportion of A over A+T, and theta2 is the proportion of G over G+C.

`Word(frequency={frequency set description})`

or

`Word(frequency1={frequency set description}, frequency2={frequency set description}, ..., frequencyn={frequency set description})`

frequencies on words computed as the product of frequencies on the letters. The arguments *frequency* and *frequency{i}* are for descriptions of frequency sets on single sites such as nucleotides or proteins. The alphabet must be a Word alphabet.

If the argument is *frequency*, the number of multiplied single site frequencies is the length of the words in the alphabet, and the *same* single site frequency set is used (ie the parameters are shared between all positions).

If the arguments are *frequency1*, ..., *frequency{n}*, the length of the words in the alphabet must be *n*, and all single site frequency sets are independent. In that case, all single site frequency set parameters are position dependent.

```
alphabet=Word(letter=DNA,length=4)
```

```
Word(frequency=GC())
```

builds a frequency set on 4 bases words, such that all sites frequencies follow the same GC frequency set model. The parameter name is *Word.1234\_GC.theta*.

```
alphabet=Word(letter=DNA,length=4)
```

```
Word(frequency1=GC(),frequency2=GC(),frequency3=Fixed(),\
      frequency4=Full())
```

builds a frequency set on 4 bases words, such first and second sites follow independent GC frequency sets, third site follows a Fixed frequency set, and fourth site follows a Full frequency set. Then the parameters names are *Word.1\_GC.theta*, *Word.2\_GC.theta*, *Word.4\_Full.theta\_1*, *Word.4\_Full.theta\_2*, *Word.4\_Full.theta\_3*.

All functions accept the following arguments, that take priority over the parameter specification:

`init={balanced,observed}`

Set all frequencies to the same value, or to their observed counts.

`pseudoCount={integer}`

If the frequencies are set from observed counts, a pseudoCount is added to all the counts.

`values={vector<double>}`)

Explicitly set all frequencies manually. The size of the input vector should equal the number of resolved states in the alphabet, be in alphabetical order of states, and sum to one.

### 3.3.4 Rate across site distribution

From version 0.4.0, BppSuite uses the keyval syntax for specifying the distributions of substitution rate across sites.

`rate_distribution = {rate distribution description}`

Specify the rate across sites distribution.

The rate distribution is set to have a mean of 1. The following distributions are currently available:

**Uniform** Uses a constant rate across sites.

`Gamma(n={int}>=2, alpha={float}>0)`

A discretized gamma distribution of rates, with  $n$  classes, and a given shape, with mean 1 (scale=shape).

`Invariant(dist={rate distribution description}, p={real}[0,1])`

A composite distribution allowing a special class of invariant site, with a probability  $p$ .

## 3.4 Discrete distributions

Bio++ contains several probability distributions (currently only discrete or discretized ones). These are:

### 3.4.1 Standard Distributions

`Constant(value={float})`

a Dirac distribution on  $value$ , with parameter  $value$ .

`Beta(n={int}>=2, alpha={float}>0, beta={float}>0)`

a discretized beta distribution, with  $n$  classes, with standard parameters  $alpha$  and  $beta$ .

`Gamma(n={int}>=2, alpha={float}>0, beta={float}>0)`

a discretized gamma distribution, with  $n$  classes, a shape  $alpha$  and a rate  $beta$ , with parameters  $alpha$  and  $beta$ .

`Gaussian(n={int}>=1, mu={float}, sigma={float}>0)`

a discretized gaussian distribution, with  $n$  classes, a mean  $mu$  and a standard deviation  $sigma$ , with parameters  $mu$  and  $sigma$ .

`Exponential(n={int}>=2, lambda={float}>0)`

a discretized exponential distribution, with  $n$  classes and parameter  $lambda$ .

`Simple(values={vector<double>}, probas={vector<double>})`

a discrete distribution with specific values (in  $values$ ) and their respective non-negative probabilities (in  $probas$ ). The parameters are  $V1, V2, \dots, Vn$  for all the values and the relative probability parameters are  $theta1, theta2, \dots, thetan-1$ .

`TruncExponential(n={int}>=2, lambda={float}>0, tp={float}>0)`

a discretized truncated exponential distribution, with  $n$  classes, parameter  $lambda$  and a truncation point  $tp$ . The parameters are  $lambda$  and  $tp$ .

`Uniform(n={int}>=1, begin={float}>0, end={float}>0)`

a uniform distribution, with  $n$  classes in interval  $[begin, end]$ . There are no parameters.

### 3.4.2 Mixture Distributions

`Mixture(probas={vector<double>}, distribution1={distribution description}, ..., distributionn={distribution description})`

a Mixture of discrete distributions with specific probabilities (in *probas*) and their respective descriptions. (in *probas*). The parameters are the relative probability parameters  $theta_1, theta_2, \dots, theta_{n-1}$ , and the parameters of the included distributions prefixed by *Mixture.i* where  $i$  is the order of the distribution.

For example:

```
Mixture(probas=(0.3,0.7),distribution1=Beta(n=5,alpha=2,beta=3),\
        distribution2=Gamma(n=10,alpha=9,beta=2))
```

builds a mixture of a discrete beta distribution and of a discrete gamma distribution, with a total of 15 classes. The parameters names are *Mixture.theta1*, *Mixture.1.Beta.alpha*, *Mixture.1.Beta.beta*, *Mixture.2.Gamma.alpha* and *Mixture.2.Gamma.beta*.

## 3.5 Numerical parameters estimation

Some programs allow you to (re-)estimate numerical parameters, including

- Branch lengths
- Entries of the substitution matrices, included base frequencies values)
- Parameters of the rate distribution (currently shape parameter of the gamma law, proportion of invariant sites).

`optimization = {method}`

where “method” can be one of

`None` (no optimization is performed, initial values are kept “as is”.)

`FullD(derivatives={Newton|Gradient})`

Full-derivatives method. Branch length derivatives are computed analytically, others numerically. The *derivatives* arguments specifies if first or second order derivatives should be used. In the first case, the optimization method used is the so-called conjugate gradient method, otherwise the Newton-Raphson method will be used.

`D-Brent(derivatives={Newton|Gradient}, nstep={int}>0)`

Branch lengths parameters are optimized using either the conjugate gradient or the Newton-Raphson method, other parameters are estimated using the Brent method in one dimension. The algorithm then loops over all parameters until convergence. The *nstep* arguments allow to specify a number of progressive steps to perform during optimization. If ‘*nstep=3*’ and ‘*precision=E-6*’, a first optimization with ‘*precision=E-2*’, will be performed, then a round with ‘*precision*’ set to E-4 and finally ‘*precision*’ will be set to E-6. This approach generally increases convergence time.

`D-BFGS(derivatives={Newton|Gradient}, nstep={int}>0)`

Branch lengths parameters are optimized using either the conjugate gradient or the Newton-Raphson method, other parameters are estimated

using the BFGS method. The algorithm then loops over all parameters until convergence. The *nstep* arguments allow to specify a number of progressive steps to perform during optimization. If 'nstep=3' and 'precision=E-6', a first optimization with 'precision=E-2', will be performed, then a round with 'precision' set to E-4 and finally 'precision' will be set to E-6. This approach generally increases convergence time.

`optimization.reparametrization = {boolean}`

Tells if parameters should be transformed in order to remove constraints (for instance positive-only parameters will be log transformed in order to obtain parameters defined from -inf to +inf). This may improve the optimization, particularly for parameter-rich models, but the likelihood calculations will take a bit more time.

`optimization.final = {powell|simplex}`

Optional final optimization step, useful if numerical derivatives are to be used. Leave the field empty in order to skip this step.

`optimization.profiler = {{path}|std|none}`

A file where to dump optimization steps (a file path or std for standard output or none for no output).

`optimization.message_handler = {{path}|std|none}`

A file where to dump warning messages.

`optimization.max_number_f_eval = {int<0}`

The maximum number of likelihood evaluations to perform.

`optimization.ignore_parameter = {list<chars>}`

A list of parameters to ignore during the estimation process. The parameter name should include their "namespace", that is their model name, for instance K80.kappa, TN93.theta, GTR.a, Gamma.alpha, etc. For nested models, the syntax is the following: G01.rdist\_Gamma.alpha, TS98.model\_T92.kappa, RE08.lamba, RE08.model\_G01.model\_GTR.a, etc. 'Ancient' will ignore all parameters in the ancestral frequency set (non-homogeneous models), and 'BrLen' will ignore all branch lengths.

The '\*' wildcard can be used as a suffix, as in YN98.freq\_Word.1\_\* for all the parameters whose names start with YN98.freq\_Word.1\_.

`optimization.tolerance = {float>0}`

The precision on the log-likelihood to reach.

`output.tree.file = {{path}|none}`

File path where to write the optimized tree.

`output.infos = {{path}|none}`

A text file containing several statistics for each site in the alignment. These statistics include the posterior rate, rate class with maximum posterior probability and whether the site is conserved or not.

### 3.6 Writing sequences/alignments to files

`output.sequence.file = {path}`

The output file where to write the sequences.

`output.sequence.format = {sequence format description}`

The output file format, using the same syntax as for reading (see [Section 3.1 \[Sequences\]](#), page 5). Only formats Fasta, Mase and Phylip are supported for writing.

In addition, most of the formats support the `length` argument, that specifies the maximum number of sequence characters to output on each line (default set to 100).

## 4 Bio++ Program Suite Reference

This section now details the specific options for each program in the Bio++ Program suite.

### 4.1 BppML: Bio++ Maximum Likelihood

The BppML program uses the common syntax introduced in the previous section for setting the alphabet, loading the sequences (see [Section 3.1 \[Sequences\]](#), page 5), specifying the model (see [Section 3.3 \[Model\]](#), page 6), and estimating parameters (see [Section 3.5 \[Estimation\]](#), page 18).

The BppML program allows you to optimize tree topologies and model parameters and perform a bootstrap analysis.

#### 4.1.1 Branch lengths initial values

`init.tree = {user|random}`

Set the method for the initial tree to use. The ‘user’ option allows you to use an existing file using the method described in the Common options section. This file may have been built using another method like neighbor joining or parsimony for instance. The ‘random’ option picks a random tree, which is handy to test convergence. This may however slows down significantly the optimization process.

`init.brln.method = {method description}`

Set how to initialize the branch lengths. Available methods include:

`Input`      Keep initial branch lengths as is.

`Equal(value={float>0})`

Set all branch lengths to the same value, provided as argument.

`Clock`      Coerce to a clock tree.

`Grafen(height={{real>0}|input}, rho = {real>0})`

Uses Grafen’s method to compute branch lengths. In Grafen’s method, each node is given a weight equal to the number of underlying leaves. The length of each branch is then computed as the difference of the weights of the connected nodes, and further divided by the number of leaves in the tree. The height of all nodes are then raised to the power of ‘rho’, a user specified value. The tree is finally scaled to match a given total height, which can be the original one (‘height=input’), or fixed to a certain value (usually ‘height=1’). A value of rho=0 provides a star tree, and the greater the value of rho, the more recent the inner nodes.

#### 4.1.2 Topology optimization

`optimization.topology = {boolean}`

Enable the tree topology estimation.

`optimization.topology.algorithm = {NNI}`

Algorithm to use for topology estimation: only NNI available for now.

`optimization.topology.algorithm_nni.method = {fast|better|phym1}`

Set the NNI method to use. ‘fast’: test sequentially all NNI, if a NNI improving the likelihood is found, it is performed. ‘better’: test all possible NNIs, do the one with the biggest likelihood increase. ‘phym1’: test all possible NNIs, try doing all the improving ones. If the final likelihoods is better, perform all NNIs. Otherwise, try to do half of them, and so on. In most cases the ‘phym1’ option shows the best performance.

`optimization.topology.nstep = {int>0}`  
 Number of phylml topology movement steps before re-optimizing parameters.

`optimization.topology.numfirst = {boolean}`  
 Shall we estimate parameters before looking for topology movements?

`optimization.topology.tolerance.before = {real>0}`  
 Tolerance for the prior-topology estimation. The tolerance numbers should not be too low, in order to save computation time and also for a better topology estimation. The ‘`optimization.tolerance`’ parameter will be used for the final optimization of numerical parameters (see Common options).

`optimization.topology.tolerance.during = 100`  
 Tolerance for the during-topology estimation

`optimization.scale_first = no`  
 Shall we first scale the tree before optimizing parameters?

`optimization.scale_first.tolerance = {double}`  
 The convergence criterion to achieve in the optimization.

### 4.1.3 Molecular clock

BppML can also optimize branch lengths with a molecular clock:

`optimize.clock={no|global}`  
 Tell if a molecular clock should be assumed. Topology estimation is not possible with a clock constraint.

### 4.1.4 Output results

`output.infos = {{path}|none}`  
 Alignment information log file (site specific rates, etc):

`output.estimated = {{path}|none}`  
 Write numerical parameter estimated values.

### 4.1.5 Bootstrap analysis

`bootstrap.number = {int>0}`  
 Number of replicates. A reasonable value would be  $\geq 100$ .

`bootstrap.approximate = {boolean}`  
 Tell if numerical parameters should be kept to their initial value when bootstrapping.

`bootstrap.verbose = {boolean}`  
 Set this to yes for detailed output when bootstrapping.

`bootstrap.output.file = {{path}|none}`  
 Where to write the resulting trees (multi-trees newick format).

### 4.1.6 Rather technical options

These options are mainly for debugging or testing purpose, in most case you will be happy with the default setting.

`likelihood.recursion = {simple|double}`  
 Set the type of likelihood recursion to use. ‘`simple`’: derivatives take more time to compute, but likelihood computation is faster. For big data sets, it can save a lot of memory usage too, particularly when the data are compressed. ‘`double`’: uses more

memory and need more time to compute likelihood, due to the double recursion. Analytical derivatives are however faster to compute.

This command has no effect in the following cases: (i) topology estimation: this requires a double recursive algorithm, (ii) optimization with a molecular clock: a simple recursion with data compression is used in this case, due to the impossibility of computing analytical derivatives.

```
likelihood.recursion_simple.compression = {simple|recursive}
```

Site compression for the simple recursion: ‘simple’: identical sites are not computed twice, ‘recursive’: look for site patterns to save computation time during optimization, but requires extra time for building the patterns. This is usually the best option, particularly for nucleotide data sets.

## 4.2 BppSeqGen: Bio++ Sequence Simulator

The BppSeqGen program uses the common syntax introduced in the previous section for setting the alphabet, loading the sequences (see [Section 3.1 \[Sequences\], page 5](#)) and tree (see [Section 3.2 \[Tree\], page 6](#)), specifying the model (see [Section 3.3 \[Model\], page 6](#)) and writing sequence data (see [Section 3.6 \[WritingSequences\], page 19](#)).

```
number_of_sites = {int>0}
```

The number of site positions to simulate.

```
input.infos = {path}
```

A info file like the one output by bppML. The estimated site-specific rates will then be used to simulate the same number of sites as found in the info file, with the corresponding rates.

## 4.3 BppAncestor: Bio++ Ancestral Sequence and Rate Reconstruction

The BppAncestor program uses the common syntax introduced in the previous section for setting the alphabet, loading the sequences (see [Section 3.1 \[Sequences\], page 5](#)) and tree (see [Section 3.2 \[Tree\], page 6](#)), specifying the model (see [Section 3.3 \[Model\], page 6](#)) and writing sequence data (see [Section 3.6 \[WritingSequences\], page 19](#)).

Specific options are:

```
asr.method = {marginal}
```

That’s the only option for now!

```
asr.probabilities = {boolean}
```

Tells if we should output the site specific probabilities in each case.

```
asr.sample = {boolean}
```

Tell if we should sample from the posterior distribution instead of using the maximum probability.

```
asr.sample.number = 10 [[asr.sample=yes]]
```

Number of sample sequences to output.

```
asr.add_extant = {boolean}
```

Should extant (observed) sequences be added to the output sequence file? The sequences added are the ones which are used for the actual calculation. If they contained gaps for instance, and that these have been replaced by the unknown character (N or X for example), then the sequence with unknown characters will be used.

```
output.sites.file = {{path}|none}
```

Alignment information log file (site specific rates, probabilities, etc).

```
output.nodes.file = {{path}|none}
```

Ancestral nodes information: expected frequencies of ancestral states.

## 4.4 BppDist: Bio++ Distance Methods

The BppDist program uses the common syntax introduced in the previous section for setting the alphabet, loading the sequences (see [Section 3.1 \[Sequences\]](#), page 5) and tree (see [Section 3.2 \[Tree\]](#), page 6) and specifying the model (see [Section 3.3 \[Model\]](#), page 6, only the section corresponding to the homogeneous case).

Specific options are:

```
output.matrix.file = {{path}|none}
```

Where to write the matrix file (only philip format supported for now).

```
method = {wpgma|upgma|nj|bionj}
```

The algorithm to use to build the tree.

```
optimization.method = {init|pairwise|iterations}
```

There are several ways to optimize substitution parameters. The ‘init’ option corresponds to the standard behavior, that is, keeping them to their initial, user-provided value. The ‘pairwise’ option estimate those parameters in a pairwise manner. This should be avoided, particularly with parameter-rich models. Finally the ‘iterations’ option corresponds to Ninio et al, *Bioinformatics* (2007) recursive algorithm: After each distance tree, a global ML estimation of the substitution parameters is performed. The estimated values are then used to rebuild a distance matrix and a tree. The algorithm stops when the topology does not change anymore. The ML optimization uses the parameters described in (see [Section 3.5 \[Estimation\]](#), page 18).

```
output.tree.file = {{path}|none}
```

The final tree, possibly with bootstrap values: BppDist uses the same options for bootstrap analysis than the BppML program (see [Section 4.1 \[bppml\]](#), page 21).

## 4.5 BppPars: Bio++ Maximum Parsimony

The BppPars program is currently quite limited and should not be used for serious phylogenetic analysis. It can compute parsimony scores and perform topology estimation using the same algorithm of BppML. It uses the common syntax introduced in the previous section for setting the alphabet, loading the sequences (see [Section 3.1 \[Sequences\]](#), page 5) and tree (see [Section 3.2 \[Tree\]](#), page 6).

Specific options are:

```
optimization.topology = {boolean}
```

Tell if topology has to be estimated.

```
output.tree.file = {{path}|none}
```

Where to print the output file.

```
bootstrap.number = {int>0}
```

Number of bootstrap replicates to perform.

```
bootstrap.output.file = {{path}|none}
```

Where to write bootstrap trees.

## 4.6 BppConsense: Bio++ Consensus Trees

Probably one of the simplest program to use in the suite, just takes a list of trees (for instance produced by BppML, BppDist or BppPars with the bootstrap option enabled) and compute bootstrap values for a reference tree, provided as input, or constructed using a consensus method. The program uses the multiple-trees reading options for input (see [Section 3.2 \[Tree\], page 6](#)) and single-tree writing options for output.

There are only specific options here:

`tree = {tree methods}`

The method to use for getting the reference tree. Available function are:

**Input**        The tree is loaded using the single-tree reading options (see [Section 3.2 \[Tree\], page 6](#)).

`Consensus(threshold = {int[0,1]})`

Build a consensus tree according to a given threshold. 0 will output a fully resolved tree, 0.5 corresponds to the majority rule and 1 to the strict consensus, but any intermediate value can be specified.

## 4.7 BppPhySamp: Bio++ Phylogenetic Sampler

The Bio++ Phylogenetic Sampler samples sequences from a file according to phylogenetic information. The goal is to clean a big data set by removing redundant sequences, bringing only few additional information for evolutionary analyses.

The BppPhySamp programs uses the common options for setting the alphabet, loading the sequences (see [Section 3.1 \[Sequences\], page 5](#)) and (see [Section 3.2 \[Tree\], page 6](#)) and writing the resulting data set (see [Section 3.6 \[WritingSequences\], page 19](#)).

`input.method = {tree|matrix}`

The method to provide phylogenetic information, either by a tree or a matrix. If the ‘tree’ option is used, then the options for reading trees are used (see [Section 3.2 \[Tree\], page 6](#)).

`input.matrix = {path} [[input.method = matrix]]`

The input matrix file.

`deletion_method = {random|threshold|sample}`

Method to use to remove sequence.

`threshold = {float>0} [[deletion_method = threshold ]]`

The minimum distance separating two sequences in the sampled data set. Any sequences closer than this threshold in the original data set will be confronted so that only one is kept.

`sample_size = {int>0} [[deletion_method = sample|random ]]`

The number of sequences to keep in the final data set.

`choice_criterion = {length|length.complete|random}`

How to chose between closely related sequences? ‘length’ takes the longest (maximum number of non-gap positions), ‘length.complete’ takes the sequence with the maximum number of fully resolved positions and ‘random’ picks one sequence at random.

## 4.8 BppReroot: Bio++ Serial Tree Re-rooting

`input.trees.file={path}`

A path toward multi-trees file (newick).

`outgroups.file={path}`

A path toward a file containing the different levels of outgroups.

`print.option={boolean}`

If set to true, the unrootable trees are printed as unrooted in the output file, otherwise the unrootable trees are not printed.

`tryAgain.option={boolean}`

If set to true and ReRoot finds a non-monophyletic outgroup, it tries the next outgroup. Otherwise, if ReRoot finds a non-monophyletic outgroup, the analysis for this tree is interrupted. No more outgroups are analysed.

`output.trees.file={path}`

File where to write the rerooted trees.

## 4.9 BppSeqMan: Bio++ Sequence Manipulation

The Bio++ Sequence Manipulator convert between various file formats, and can also perform various operations on sequences. It uses the common options for setting the alphabet, loading the sequences (see [Section 3.1 \[Sequences\], page 5](#)) and writing the resulting data set (see [Section 3.6 \[WritingSequences\], page 19](#)). It can use the “Generic” option for alphabets if only file format conversion is to be performed, but the correct alphabet must be specified for more advanced manipulations, like in silico molecular biology.

BppSeqMan can perform any number of elementary operation, in any order, providing the output of operation n is compatible with input of operation n+1, and that the input of operation 1 is compatible with the input data.

Specific options:

`sequence.manip = {list<string>}`

The list, in appropriate order, of elementary operations to perform. See below for a list of these operations.

‘Complement [[alphabet = DNA or RNA]]’

Convert to the complementary sequence, keeping the original alphabet.

‘Transcript [[alphabet = DNA or RNA]]’

Convert to the complementary sequence, switching the type of alphabet (DNA<->RNA).

‘Switch [[alphabet = DNA or RNA]]’

Change the alphabet type (DNA<->RNA).

‘Translate(code = {genetic code}) [[alphabet = DNA or RNA]]’

Convert to proteins. You have to specify a genetic code, see specific options. ‘code’: The genetic code to use for the translation, one of

- EchinodermMitochondrialGeneticCode
- InvertebrateMitochondrialGeneticCode
- StandardGeneticCode
- VertebrateMitochondrialGeneticCode
- YeastMitochondrialGeneticCode

‘Invert’ Invert the sequence 5’ <-> 3’ or N <-> C

‘RemoveGaps’

Remove all gaps in sequences (ie, ‘unalign’).

‘GapToUnknown’

Change gaps to fully unresolved characters, N for nucleotides and X for proteins.

- ‘UnknownToGap’**  
Change (partially) unresolved characters to gaps.
- ‘RemoveStops’**  
Remove all sites with at least one stop codon.
- ‘GetCDS’** Remove the first stop codon and everything after in codon sequences.
- ‘CoerceToAlignment’**  
Try to convert a set of sequence to an alignment. This will fail if sequences do not have the same length. This step is required before trying commands **‘ResolveDotted’** or **‘KeepComplete’**.
- ‘ResolveDotted(alphabet={RNA|DNA|Proteins}) [[Aligned sequences]]’**  
Convert a human-readable alignment to a machine-readable alignment. This manipulation must be first if it is used, and the data must be load with the **‘Generic’** alphabet. **‘alphabet’**: The alphabet to use in order to resolve a dotted alignment.
- ‘KeepComplete(maxGapAllowed={int>0} or {float[0,100]}+%) [[Aligned sequences]]’**  
Keep only complete sites, ie sites without any gap. Sites with unresolved characters are not removed. It is also possible to fix a maximum proportion of gaps, see specific options. **‘maxGapAllowed’**: The maximum proportion of gaps allowed.

Examples of use:

- Just change file format:  
`sequence.manip=`
- Change DNA to RNA:  
`sequence.manip=Switch`
- Unalign sequences, perform transcription and translate to proteins:  
`sequence.manip=RemoveGaps,Transcript,Translate`
- Change all unresolved characters to gaps and keep only positions with less than 5 gaps:  
`sequence.manip=UnknownToGap,KeepComplete(maxGapAllowed=5)`
- Keep only positions with less than 30% of gaps, and change them to unresolved characters:  
`sequence.manip=KeepComplete(maxGapAllowed=30%),GapToUnknown`

## 4.10 BppTreeDraw: Bio++ Tree Drawing

This is a simple program that outputs a tree in various vector formats. It takes as input a tree following the standard syntax.

Specific options:

`output.drawing.file = {path}`

The file where to output the figure.

`output.drawing.format = {Svg|Xfig|Inkscape|Pgf}`

The file format.

`output.drawing.plot = {plotting algorithm}`

The plotting algorithm can be either Phylogram or Cladogram. They follow the keyval syntax, with the following arguments:

**‘xu, yu {float}’**

The scale units for x and y axis.

**‘direction.h {left2right|right2left}’**

Horizontal orientation of the tree plot.

`'direction.v {top2bottom|bottom2top}'`

Vertical orientation of the tree plot.

`'draw.leaves, draw.ids, draw.brLen, draw.bs {boolean}'`

Tell if leaf names, node ids, branch lengths and/or bootstrap should be drawn.