BppSuite Manual

Julien Dutheil, Laurent Guéguen
dutheil@evolbio.mpg.de

1

This is the manual of the Bio++ Program Suite, version 2.4.0.
Copyright © 2007-2018 Bio++ development team

Table of Contents

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 ',' ponctuations.

2.3 Variables

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

```
topo.algo = NNI
topo.algo_nni.method = phyml
output.tree.file = MyData_$(topo.algo)_$(topo.algo_nni.method).dnd
```

You can use this syntax to define global variables:

```
data=MyData
input.sequence.file=$(data).fasta
input.tree.file=$(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
input.sequence.file=$(data).fasta
input.sequence.format=Fasta
```

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

3 Common options encountered in several programs.

3.1 Setting alphabet and genetic code

`alphabet = {DNA|RNA|Protein|Binary|Word(letter={DNA|RNA|Protein},length={int})| Codon(letter={DNA|RNA})}`

The alphabet to use when reading sequences. DNA and RNA alphabet can in addition take an argument:

`bangAsgap={bool}`

Tell is exclamation mark should be considered as a gap character. The default is to consider it as an unknown character such as 'N' or '?'.

`genetic_code = {translation table}`

The genetic code used for codon alphabet, where 'translation table' specifies the code to use, either as a text description, or as the NCBI number. The following table give the currently implemented codes with their corresponding names:

Standard	1
VertebrateMitochondrial	2
YeastMitochondrial	3
MoldMitochondrial	4
InvertebrateMitochondrial	5
CiliateNuclear	6
EchinodermMitochondrial	9
AscidianMitochondrial	13

The states of the alphabets are in alphabetical order. For the proteic alphabet, the amino-acid are in the order of their 3-letters code (ALA, ARG, ASN, ...).

3.2 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.

`input.sequence.keep_names = all(default) | {list of sequence names}`

Provide an optional list of sequences to keep. Sequences whose name is not in the list will be discarded. Names in the list with no match in the sequence file will be ignored.

`input.sequence.remove_names = none(default) | {list of sequence names}`

Procide an optional list of sequences to be discarded. Names in the list with no match in the sequence file will raise an error. This option can be combined with the previous one.

`input.site.selection = {list of integers}`

Will only consider sites in the given list of positions, in extended format : positions separated with ",", and "i-j" for all positions between i and j, included.

`input.site.selection = {Sample(n={integer} [, replace={true}])}`

Will consider {n} random sites, with optional replacement.

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

`Fasta(extended={bool}, strictNames={bool})`
: The fasta format. The argument `extended`, default to 'no', allows to enable the HUPO-PSI extension of the format. The argument `strict_names`, default to 'no', specifies that only the first word in the fasta header is used as a sequence names, the rest of the header being considered as comments.

`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 (default: 'complete'). 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).

`input.sequence.max_unresolved_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 unresolved states 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).

3.3 Reading trees

`input.tree.file = {path}`
: The phylogenetic tree file to use.

`input.tree.format = {Newick|Nexus|NHX}`
: The format of the input tree file.

In case the input tree does not specify node identifiers, some will be generated automatically. Nodes identifiers can be outputed using the following option:

`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.

In case it is supported by the program, the use of that option will cause the program to exit just after producing the tagged tree.

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|NHX}`
: The format of the input tree file.

3.4 Specifying biochemical properties and distances

Some methods require an "alphabet index" to be specified. Alphabet indexes associate a value with each alphabet state (Index1, e.g. a biochemical property) or for a pair of states (Index2, e.g. a biochemical distance). This section describes the supported indexes:

3.4.1 Index1

`None`
: If no index should be used.

`Surface, Mass, Volume, Charge {AA}`
: Basic amino acids properties.

`GranthamPolarity, GranthamVolume {AA}`
: Grantham's polarity and volume index.

`KleinCharge {AA}`
: Klein's charge.

`ChouFasmanAHelix, ChouFasmanBSheet, ChouFasmanTurn {AA}`
: Chou and Fasmani score for secondary structure prediction.

`ChenGuHuangHydrophobicity {AA}`
: Hydrophobicity according to Chen, Gu and Huang.

`SEALow, SEAMedium, SEAHigh {AA}`
: Solvent Exposed Area, percent of amino acids having a SEA below 10, between 10 and 30, or higher than 30, respectively.

`User` A user defined Index1, from a file in the AAIndex1 syntax. The input file is specified using the `file={path}` argument. `file`

3.4.2 Index2

`None` If no index should be used.

`Blosum50 {AA}`
: The BLOSUM 50 amino acid distance matrix.

`Grantham, Miyata {AA}`
: Two biochemical distance matrices. Both accept an optional argument `symmetrical={boolean}` allowing to specify if the matrix should be symmetric or not. If not, the distance measure will be signed.

`Diff` Allow to compute a distance matrix by taking the difference for, each pair of state, of an Index1 value. The Index1 to use is specified using the `index1={Index1 description}` argument. An additional argument allow to specify whether the resulting matrix should be symetric (`symmetrical={boolean}`): if so, the absolute difference will be used. Alternatively, the distance will be signed and d[i,j] = - d[j,i].

`User` A user defined Index2, from a file in the AAIndex2 syntax. The input file is specified using the `file={path}` argument. The `symmetrical={boolean}` argument can be used to specify whether distances should be signed or not.

3.5 Process 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.

3.5.1 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.

Many models have a set of optional parameters denoted here as `"equilibrium frequencies"` that are used to initialize the parameters of the model related with the equilibrium frequencies. These options are:

`initFreqs=values({real]0,1[},...,{real]0,1[})`
: The equilibrium frequency is set equal (as much as possible) to the given frequencies. Those frequencies are given in the same order as the alphabet, and they must sum 1.

`initFreqs=observed`
: The equilibrium frequency is set equal (as much as possible) to the observed frequencies.

`initFreqs.observedPseudoCount={integer}`
: a peusocount integer added to all counts of letters (or words), when the frequencies are computed from observed data.

3.5.1.1 Nucleotide models

`JC69` The Jukes and Cantor model. This model has no additional parameter. See the Bio++ description.

`K80([kappa={real>0}])`
: The Kimura 2 parameters model. *kappa* is the transition over transversion ratio. Default: *kappa*=1. See the Bio++ description.

`F84([kappa={real>0}, theta={real]0,1[}, theta1={real]0,1[},theta2={real]0,1[}, "equilibrium frequencies"] )`
: 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). See the Bio++ description.

`HKY85([kappa={real>0}, theta={real]0,1[}, theta1={real]0,1[}, theta2={real]0,1[}, "equilibrium frequencies"])`
: 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`. See the Bio++ description.

`T92([kappa={real>0}, theta={real]0,1[} ,"equilibrium frequencies"])`
: Tamura 1992's model for nucleotides, similar to `HKY85`, yet assuming that the frequencies of A = T/U and G = C. See the Bio++ description.

`TN93([kappa1={real>0}, kappa2={real>0}, theta={real]0,1[}, theta1={real]0,1[}, theta2={real]0,1[}, "equilibrium frequencies"])`
: Tamura and Nei 1993's model, similar to `HKY85`, but allowing for two distinct transition/transversion ratios. See the Bio++ description.

`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[} ,"equilibrium frequencies"])`
: The General Time-Reversible substitution model. Parameters *a*, *b*, *c*, *d*, *e* are the entries of the exchangeability matrix. See the Bio++ description.

`L95([beta={real>0}, gamma={real>0}, delta={real>0}, theta={real]0,1[}, theta1={real]0,1[}, theta2={real]0,1[} ,"equilibrium frequencies"])`
: The strand-symmetric model of Lobry 1995, for nucleotides. See the Bio++ description.

`SSR([beta={real>0}, gamma={real>0}, delta={real>0}, theta={real]0,1[}])`
: The strand-symmetric reversible model, for nucleotides. See the Bio++ description.

`RN95([thetaR={real]0,1[}, thetaC={real]0,1[}, thetaG={real]0,1[}, kappaP={real[0,1[}, gammaP={real[0,1[}, sigmaP={real>1}, alphaP={real>1}])`
: The model described by Rhetsky and Nei, where the only hypothesis is that the transversion rates are only dependent of the target nucleotide. This model is not reversible. See the Bio++ description.

`RN95s([thetaA={real]0,0.5[}, gamma={real]0,0.5[}, alphaP={real>1}])`
: The instersection of models RN95 and L95. The two hypotheses are that the transversion rates are only dependent of the target nucleotide, and strand symmetry. See the Bio++ description.

3.5.1.2 Protein models

`JC69`
: The Jukes and Cantor model. This model has no additional parameter. See the Bio++ description.

`DSO78`
: Protein substitution model, using the dcmutt implementation of Kosiol and Goldman 2005. See the Bio++ description.

`JTT92`
: Protein substitution model, using the dcmutt implementation of Kosiol and Goldman 2005. See the Bio++ description.

`WAG01`
: Protein substitution model, from Whelan & Goldman 2001. See the Bio++ description.

`LG08`
: Protein substitution model, from Le & Gascuel 2008. See the Bio++ description.

`LLG08_EX2([relrate1={real]0,1[}, relproba1={real]0,1[}])`
: Protein substitution model, from Le, Lartillot & Gascuel 2008. See Section 3.5.1.7 [Mixture], page 18, for the meaning of the variables. See the Bio++ description.

`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 Section 3.5.1.7 [Mixture], page 18, for the meaning of the variables. See the Bio++ description.

`LLG08_EHO([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 Section 3.5.1.7 [Mixture], page 18, for the meaning of the variables. See the Bio++ description.

`LLG08_UL2([relrate1={real]0,1[}, relproba1={real]0,1[}])`
: Protein substitution model, from Le, Lartillot & Gascuel 2008. See Section 3.5.1.7 [Mixture], page 18, for the meaning of the variables. See the Bio++ description.

`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 Section 3.5.1.7 [Mixture], page 18, for the meaning of the variables. See the Bio++ description.

`LGL08_CAT(nbCat={[10,20,30,40,50,60]}, [relrate1={real]0,1[}, relrate2={real]0,1[}, ..., relproba1={real]0,1[}, relproba2={real]0,1[}, ...] ))`
: CAT protein substitution model, from Le, Gascuel & Lartillot 2008, with a given number (*nbCat*) of profiles. See Section 3.5.1.7 [Mixture], page 18, for the meaning of the variables. See the Bio++ description.

`LGL08_CAT_C{[1,...,nbCat]}(nbCat={[10,20,30,40,50,60]})`
: Submodel of a given CAT Protein substitution model, from Le, Gascuel & Lartillot 2008, with a given number (*nbCat*) of profiles. See the Bio++ description.

`DSO78+F([theta={real]0,1[}, theta1={real]0,1[}, theta2={real]0,1[}, ... ,"equilibrium frequencies"])`
: 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. See the Bio++ description.

`JTT92+F([theta={real]0,1[}, theta1={real]0,1[}, theta2={real]0,1[}, ..., "equilibrium frequencies"])`
: Protein substitution model, using the dcmutt implementation of Kosiol and Goldman 2005 and free equilibrium frequencies. See the Bio++ description.

`WAG01+F([theta={real]0,1[}, theta1={real]0,1[}, theta2={real]0,1[}, ..., "equilibrium frequencies"])`
: Protein substitution model, from Whelan & Goldman 2001, and free equilibrium frequencies. See the Bio++ description.

`LG08+F([theta={real]0,1[}, theta1={real]0,1[}, theta2={real]0,1[}, ..., "equilibrium frequencies"])`
: Protein substitution model, from Le & Gascuel 2008, and free equilibrium frequencies. See the Bio++ description.

`Empirical(name={chars}, file={path})`
: Build a protein substitution model from a file in PAML format, and use 'name' as a namespace for parameters.

`Empirical+F(name={chars}, file={path}, [theta={real]0,1[}, theta1={real]0,1[}, theta2={real]0,1[}, ..., "equilibrium frequencies"])`
: Build a protein substitution model from a file in PAML format, and use free equilibrium frequencies. 'name' will be used as a parameter namespace, including for frequencies.

3.5.1.3 Miscellaneous models

`Binary([kappa={real>0} ,"equilibrium frequencies"])`
: Build the model on binary alphabet, where *kappa* is the relative proportion of 1 over 0 in the equilibrium distribution. Default: *kappa*=1. See the Bio++ description.

3.5.1.4 Codon models

Some codon models also take as argument a *frequencies* option specifying the equilibrium frequencies of the model. Any frequencies description can be used here, but the syntax also supports options similar to the ones used in the PAML software:

- 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 22, "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.

An optional option *mgmtStopCodon* can be set to define how the frequencies computed to stop codons in the case of F1X4 et F3X4 are distributed to other codons.

- uniform : each stop frequency is distributed evenly
- linear : each stop frequency is distributed to the neighbour codons (ie 1 substitution away), in proportion to each target codon frequency.

- quadratic (default): each stop frequency is distributed to the neighbour codons (ie 1 substitution away), in proportion to the square of each target codon frequency.

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

`GY94([kappa={real>0}, V={real>0}, "equilibrium frequencies"])`
: Goldman and Yang (1994) substitution model for codons (default values: *kappa*=1 and *V*=10000). See the Bio++ description.

`MG94([rho={real>0}, "equilibrium frequencies"])`
: Muse and Gaut (1994) substitution model for codons (default values: *rho*=1). See the Bio++ description.

`YN98([kappa={real>0}, omega={real>0}, "equilibrium frequencies"])`
: Yang and Nielsen (1998) substitution model for codons (default values: *kappa*=1 and *omega*=1). See the Bio++ description.

`YNGP_M0([kappa={real>0}, omega={real>0}, "equilibrium frequencies"])`
: The M0 model of PAML, ie the same as YN98. See the Bio++ description.

`YNGP_M1([kappa={real>0}, omega={real>0}, p0={real>0 and <1 }, "equilibrium frequencies"])`
: 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). See the Bio++ description.

`YNGP_M2([kappa={real>0}, omega0={real>0 and <1}, theta1={real>0 and <1 }], omega1={real>1}, theta2={real>0 and <1 }, "equilibrium frequencies"])`
: The M2a model of PAML, see Yang, Z., R. Nielsen, N. Goldman, and A.-M. K. Pedersen (2000), with p0=theta1 and p1=(1-theta1)*theta2 (default values: *kappa*=1, *theta1*=0.33333, *theta2*=0.5, *omega0*=0.5, *omega2*=0.5). See the Bio++ description.

`YNGP_M3([n={integer>0}, kappa={real>0}, omega0={real>0 and <1}, delta1={real>0}, ..., deltan-1={real>0}, theta1={real>0 and <1 }, ..., thetan-11={real>0 and <1 }, "equilibrium frequencies"])`
: The M3 model of PAML, see Yang, Z., R. Nielsen, N. Goldman, and A.-M. K. Pedersen (2000), with *n* discrete values, with p0=theta1 and pk=(1-theta1)*...*(1-thetak)*theta(k+1), and omegak=omega0+delta1+....+deltak (default values: *n*=3, *kappa*=1, *thetak*=1/(n-k+1), *omega0*=0.5, *deltak*=0.5). See the Bio++ description.

`YNGP_M7(n={integer>0}, kappa={real>0}, p={real>1}, q={real>1 }, "equilibrium frequencies"])`
: 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: *kappa*=1, *p*=2, *q*=2). See the Bio++ description.

`YNGP_M8(n={integer>0}, [kappa={real>0}, omegas={real>1}, p0={real>0},p={real>1}, q={real>1 }, "equilibrium frequencies"])`
: 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: *kappa*=1, *p*=2, *q*=2, *p0*=0.5, *omegas*=2). See the Bio++ description.

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 <codon model name>.<position set>_<position model name>.<position specific parameter name>.

In the following models, the arguments *model* and *model{i}* are for descriptions of models on bases.

- 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.

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.

The model names est defined through several words that can be mixed together to build models at hand. Some words are exclusive. The model description must begin with *Codon*.

Rate and *Prot* and *Dist* words define how the models are mixed, either with specific rates, or using proteic models, or with non-synonymous vs synonymous substitution rates. They are exclusive, and one of the three must be used. The default model is *Rate*.

`Rate(model... [, relrate1={real>0}, relrate2={real>0}])`

Substitution model on codons with position specific evolution rates.

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)
genetic_code=Standard
model=CodonRate(model=T92)
```

builds a model on codons, such all sites follow the same T92 model. The parameters names are *CodonRate.123_T92.kappa*, *CodonRate.relrate1*, *CodonRate.relrate2*.

```
alphabet=Codon(letter=DNA)
genetic_code=Standard
model=CodonRate(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 *CodonRate.1_T92.kappa*, *CodonRate.2_T92.kappa*, *CodonRate.relrate1*, *CodonRate.relrate2*, and can be initialized as is:

```
model=CodonRate(model1=T92(theta=0.5, kappa=2), \
                model2=T92(theta=0.4, kappa=2), model3=JC69)
```

See the Bio++ description.

`Dist(model...[, beta={real>0}])`

Substitution model on codons that takes into account the difference between synonymous and non-synonymous substitutions.

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

```
alphabet=Codon(letter=DNA)
model=CodonDist(model=T92)
```

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

```
alphabet=Codon(letter=DNA, type=Standard)
model=CodonDist(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 *CodonDist.1_T92.kappa*, *CodonDist.2_T92.kappa*, *CodonDist.beta*.

See the Bio++ description.

`Prot(model..., protmodel={proteic model name}[, beta={real>0}])`

Substitution model on codons that takes into account the substitution rates in a protein model. Those rates are multiplied by a non-synonymous susbtitution factor, aka *beta*.

Prot and *Dist* words are exclusive.

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

```
alphabet=Codon(letter=DNA)
genetic_code=Standard
model=CodonProt(model=T92, protmodel=LG08)
```

builds a model on codons, such all sites follow the same T92 model, and amino-acid rates are proportional to LG08 substition matrice. The parameters names are *CodonProt.123_T92.kappa* and *CodonProt.beta*.

Optional words to describe the use of equilibrium frequencies sets. This word should be used with nucleotidic models which equilibrium distribution is fixed, ans does not depend on parameters. Otherwise there may be problems of identifiability of the parameters.

`Freq(frequencies={frequencies set description})`

Sustitution rates are multiplied by the frequency of the target codon in the given frequencies set. This factor is described by the *frequencies* argument. See the description of the Frequencies Set below.

```
alphabet=Codon(letter=DNA)
genetic_code=Standard
model=CodonDistFreq(frequencies=Full())
```

has parameters *CodonDistFreq.012_T92.kappa*, *CodonDistFreq.Full.theta_1*, ..., *CodonDistFreq.Full.theta_60*, *CodonDistFreq.beta*.

See the Bio++ description.

`PhasFreq(frequencies={frequencies set description})`

The sustitution rates are multiplied by the product of the frequencies of the changed nucleotides – conditioned on the phase – in the given frequencies set. This factor is described by the *frequencies* argument. See the description of the Frequencies Set below.

For example, see the Bio++ description.

In addition some models are defined that allow multiple substitions, with similar logic of included words. These models are prefixed by *Kron*.

`KronDistFreq(model={model name} [,positions=pos1*pos2*...*posn + posx*...*posm + ...)])`
`KronDistFreq(model1={model name}, model1={model name}, ..., modeln={model name}[,positions=pos1*pos2*...*posn + posx*...*posm + ...])`

substitution model on codons as *CodonDistFreq* above, allowing simultaneous substitutions.

Optional argument *positions* can be used to describe which substitutions are allowed. See model See [Kron], page 16.

`KronDist(model={model name} [,positions=pos1*pos2*...*posn + posx*...*posm + ...)])`
`KronDist(model1={model name}, model1={model name}, ..., modeln={model name}[,positions=pos1*pos2*...*posn + posx*...*posm + ...])`

substitution model on codons as *CodonDist* above, allowing simultaneous substitutions.

Optional argument *positions* can be used to describe which substitutions are allowed. See model See [Kron], page 16.

`KCM7() and KCM19()`

Kronecker Codon Model based on a unique (KCM7) or one per position (KCM19) GTR model. From Zaheri \& al, MBE, 2014.

See the Bio++ description.

3.5.1.5 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.relrate1*, *Word.relrate2*, *Word.relrate3*.

See the Bio++ description.

`Kron(model={model name} [,positions=pos1*pos2*...*posn + posx*...*posm + ...)])`
`Kron(model1={model name}, model1={model name}, ..., modeln={model name}[,positions=pos1*pos2*...*posn + posx*...*posm + ...])`

substitution model on words, allowing simultaneous substitutions. 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.

The rate of a multiple substitution is the product of the rates of the single substitutions it is made of.

Without optional argument *positions*, all single and multiple substitutions are allowed.

Optional argument *positions* describes the allowed substitutions. It is written as a formula with positions between 1 and the length of the word, and symbols '*' (to link positions that must change together) and '+' (to link sets of multiple susbtitutions that are allowed).

As examples, on a DNA word with 3 positions:

```
model=Kron(model=K80(), positions=1*2*3)
```

allows only substitutions that change the 3 positions.

```
model=Kron(model=K80(), positions=1*2+3)
```

allows only substitutions that change the positions 1 and 2, and the ones that change position 3 alone.

```
model=Kron(model=K80(), positions=1*2+2*3)
```

allows only substitutions that change two neighbor positions.

```
model=Kron(model=K80(), positions=1+2+3)
```

allows only substitutions that change one position, i.e. *Word* model.

See the Bio++ description.

`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)
genetic_code=Standard
model=Triplet(model=T92)
```

builds a model on codons, such all sites follow the same T92 model. The parameters names are *Triplet.123_T92.kappa*, *Triplet.relrate1*, *Triplet.relrate2*.

```
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.relrate1*, *Triplet.relrate2*.

See the Bio++ description.

`YpR_Sym(model={model description}, [rCgT={real>=0}, rTgC={real>=0}, rCaT={real>=0}, rTaC={real>=0}])`

substitution model on quotiented triplets to handle strand symetric neighbour-dependency inside dinucleotides YpR (see Bérard and Guéguen 2012). See the Bio++ description.

`YpR_Gen(model={model description}, [rCgT={real>=0}, rcGA={real>=0}, rTgC={real>=0}, rtGA={real>=0}, rCaT={real>=0}, rcAG={real>=0}, rTaC={real>=0}, rtAG={real>=0}])`

substitution model on quotiented triplets to handle general symetric neighbour-dependency inside dinucleotides YpR (see Bérard and Guéguen 2012). See the Bio++ description.

3.5.1.6 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} [, "equilibrium frequencies"])`

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. See the Bio++ description.

`G01(model={model description}, rdist={rate distribution description}, mu={real>0} [, "equilibrium frequencies"])`

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. See the Bio++ description.

`RE08(model={model description}, lambda={real>0}, mu={real>0} [, "equilibrium frequencies"])`

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. See the Bio++ description.

3.5.1.7 Mixture of models

Mixed models are sometimes called "site models".

Mixed models combine substitution models with respective probabilities. We call submodels all the models that are mixed in the mixture. A Mixed model is either the mixture of several predefined models, or based on a "simple" model in which some parameters follow given distributions.

During the likelihood computation process, all the submodels of the mixture are successively applied on the branches, and the mean (see below) of all the likelihoods is computed.

A site can follow given paths all along the tree, with given probabilities.

In homogeneous reconstruction, a path corresponds to a same submodel on all the branches, in a stationary condition. The probability of a path is the probability of its submodel. Given a site follows a path, a likelihood can be computed; and the overall likelihood on this site is the mean of these likelihoods (given the probabilities of the paths). This means that the root distribution is a mixture of the equilibrium distributions of the submodels.

With nonhomogeneous reconstruction, several models are applied on the tree, some models are mixed, some are not. A path is a vector which size is the number of mixed models (see below for more details and the declaration of paths).

Since the attribution of a submodel from a mixed model to a given site is a unique random variable, affecting the same mixed model to a set of branches S means that the attribution to this site is the same on all the branches of S. If model M=(Ma,Mb,Mc) is defined on a set of branches S, a site in constrained to follow either Ma on all S, or Mb on all S, or Ms on all S. If we want that two branches of S are independant, two similar mixed models must be defined. Moreover, it is possible to define paths that define dependencies between submodels of different mixtures (see below).

`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 25. The description of the parameters distributions is described below. In case the range of a parameter is limited, the domain of the corresponding distribution is truncated accordingly.

```
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*.

See the Bio++ description.

`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}, "equilibrium frequencies"])`

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 probabilty 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*, *Mixture.1_GY94.V*, *Mixture.2_YN98.kappa*, *Mixture.2_YN98.omega*.

See the Bio++ description.

3.5.1.8 Conditioned models

The transition probabilities on the branches are conditioned by the occurence of given events. The model is then no-markovian, but semi-markovian. The sets of considered events follow the one (ie register) defined for substitution mapping (see the testnh manual).

`OneChange(model={model description})`

The transition probabilities along each branch are conditioned by the fact that there has been at least one substitution on this branch with thid model.

`OneChange(model={model description}, register={register` name}, numReg=num1+num2+...)

The transition probabilities along each branch are conditioned by the fact that there has been on this branch at least one substitution of the specific types in the register. The `"+"` permits the declaration of several types.

3.5.2 Setting up non-stationary / non-homogeneous models

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

3.5.2.1 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<chars>_[list<int>]}`

List the names of the parameters that are shared by all branches, or by a set of branches, defined by nodes ids between brackets. In Galtier & Gouy model, that would be *T92.kappa*, since only the theta parameter is branch-specific.

The '*' wildcard can be used, as in `*theta*` for all the parameters whose name has `theta` in it.

3.5.2.2 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.

Finally, you may find useful the following options:

`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.

3.5.2.3 Paths among non-homogeneous mixture models

To define constraints for sites between submodels, we can set "paths" that any site must follow. For example, in the following description:

```
nonhomogeneous = general
nonhomogeneous.number_of_models = 3

model1=T92()
model2=MixedModel(model=T92(kappa=Simple(values=(4,10,20),\
                                probas=(0.1,0.5,0.4))))
model3=MixedModel(model=TN93(theta1=Simple(values=(0.1,0.5,0.9),\
                                probas=(0.3,0.2,0.5))))

model1.nodes_id=0:1
model2.nodes_id=2:3
model3.nodes_id=4:5
```

In this case, on branches 2 & 3 a site follows any submodel of model 2 (but the same submodel on both branches), and on branches 4 & 5, a site follows any submodel of model 3 (the same on both branches as well). But there is no constraint between models 2 & 3, which means that a site can follow any submodel of model 2 and any submodel of model 3.

If the user wants that a site with *T92.kappa=4* in model 2 has *TN93.theta1=0.1* in model 3, that a site with *T92.kappa=10* in model 2 has *TN93.theta1=0.9* in model 3, and that other cases are free (in this case it means that *T92.kappa=20* in model 2 is linked with *TN93.theta1=0.5* in model 3), then we can use the declarations:

```
site.number_of_paths=2
site.path1=model2[T92.kappa_1] & model3[TN93.theta1_2]
site.path2=model2[T92.kappa_2] & model3[TN93.theta1_3]
```

The third path (for the remaining submodels) is automatically computed.

It is possible to link mixtures of submodels. For example,

```
site.path1=model2[T92.kappa_1] & model3[TN93.theta1_2]\
                               & model3[TN93.theta1_3]
```

means that a site that has *T92.kappa=4* in model2 has either *TN93.theta1=0.5* or *TN93.theta1=0.9* in model3.

It is also possible to use submodels numbers in the declaration of the paths. The submodels are numbered from 1 to the number of submodels, in the increasing order of the values of the mixed parameter. When several parameters are mixed inside a mixed model, it may be more difficult to get the numbers.

Also, when several parameters are mixed inside a mixed model, when only a parameter value is used in a path, it means that all values of the other parameter are used in this path. It is possible to restrain this to a specific combination of parameters values using separator "," inside the brackets. When "," is used before a parameter name, the intersection of the submodels in the path under construction and the submodels linked with this parameter is used. When "," is used before a submodel number, this number is added to the computed path. For example:

```
model1 = MixedModel(model=YN98(kappa=Simple(values=(3,4),probas=(0.1,0.9))\
        ,omega=Simple(values=(0.02,0.05),probas=(0.5,0.5)), frequencies=F0))
model2 = MixedModel(model=YN98(kappa=Simple(values=(1,2),probas=(0.1,0.9))\
        ,omega=Simple(values=(0.1,0.5),probas=(0.2,0.8))))
nonhomogeneous = general
nonhomogeneous.number_of_models = 2
nonhomogeneous.stationarity = yes
site.number_of_paths = 2
site.path1 = model1[YN98.kappa_1,YN98.omega_2,2]&model2[YN98.omega_1,YN98.kappa_2]
site.path2 = model1[YN98.kappa_2,YN98.omega_2,1]&model2[YN98.omega_2,1]
```

will define two paths. In the first path, for the first model successively the submodel numbers are 1,3 (from *YN98.kappa_1*), then restricted to 3 (because submodels of *YN98.omega_2* are 3 and 4), and then extended to 2,3. For the second model, they are successively 1,2, and then restristed to 2. In the first path, for the first model successively the submodel numbers are 2,4 (from *YN98.kappa_2*), then restricted to 4 (because submodels of *YN98.omega_2* are 3 and 4), and then extended to 1,4. For the second model, they are successively 3,4, and then extended to 1.

Because of these constraints, the probabilities of the submodels are linked. In the first example, probability of *T92.kappa=4* in model 2 equals the probability of *TN93.theta1=0.5* in model 3. Since it is contradictory with the probabilities defined in models 2 or 3, the reference probabilities are the ones of the first numbered mixed model, here model 2. In this case, the probabilities in model 3 may have no use, but with the second example the probability of submodel T92.kappa=4 equals the sum of the probabilities of submodels TN93.theta1=0.5 or TN93.theta1=0.9. The relative proportion of those models used in the declaration of model 3 is then used. Here their respective probabilities are then: 0.1*0.2/ (0.2+0.5)=0.0286 and 0.1*0.5/(0.2+0.5)=0.0714.

Concerning the optimization procedure, this choice may entail the non- identifiability of several parameters (here the probabilities in model 3), so the user should be careful about this.

When the submodel is aliased behind a model name, the paths should be defined as combinations of the model that is mixed. For example, YNGP_M2 is made of 3 YN98 models, depending of three *omega* values: <1, =1, >1. If we want a site to switch between <1 and >1 omega values between two sets of branches:

```
nonhomogeneous = general
nonhomogeneous.number_of_models = 2

model1=YNGP_M2(frequencies=F1X4)
model2=YNGP_M2(frequencies=F1X4)

model1.nodes_id=0:1
model2.nodes_id=2:3

site.number_of_paths=3
site.path1=model1[YN98.omega_1] & model2[YN98.omega_3]
site.path2=model1[YN98.omega_2] & model2[YN98.omega_2]
site.path3=model1[YN98.omega_3] & model2[YN98.omega_1]
```

Another example in the case of mixtures of mixed models, where the submodels are defined by their names:

```
nonhomogeneous = general
nonhomogeneous.number_of_models = 2

model1=LLG08_UL2()
model2=LLG08_UL3()

site.number_of_paths=2
site.path1=model1[LLG08_UL2.M2] & model2[LLG08_UL3.Q1]
site.path2=model1[LLG08_UL2.M1] & model2[LLG08_UL3.Q2] \
                                & model2[LLG08_UL3.Q3]
```

3.5.2.4 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`.

When the model is a mixture model, since there is not a set of equilibrium frequencies, with this option the root frequencies are set to be the average (with the respective probabilities of the submodels) of the equilibrium frequencies of the submodels.

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 22, depending on the alphabet used.

3.5.3 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.

`Empirical(file={path} [,col={int}])`
: Read frequencies from a file. Each frequencies is set as plain column in the file. If several columns are in the file, the number of the column can be given with {col} argument (default: 1).

`Empirical+F(name={chars}, file={path}, [theta={real]0,1[}, theta1={real]0,1[}, theta2={real]0,1[}, ..., "equilibrium frequencies"])`
: Build a protein substitution model from a file in PAML format, and use free equilibrium frequencies. 'name' will be used as a parameter namespace, including for frequencies.

`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)
nonhomogeneous.root_freq=Word(frequency=GC())
```

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

```
alphabet=Word(letter=DNA,length=4)
nonhomogeneous.root_freq=Word(frequency1=GC(),frequency2=GC(),\
                              frequency3=Fixed(),frequency4=Full())
```

  builds a root 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 *1_GC.theta*, *2_GC.theta*, *4_Full.theta_1*, *4_Full.theta_2*, *4_Full.theta_3*.

`Codon(frequency={frequency set description})`
: or

`Codon(frequency1={frequency set description}, frequency2={frequency set description}, frequency3={frequency set description})`

frequencies on codons computed as the product of frequencies on the letters, with stop codon frequencies set to zero. The arguments *frequency* and *frequency{i}* are for descriptions of frequency sets on nucleotides. The alphabet must be a Codon alphabet.

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

If the arguments are *frequency1*, *frequency2*, *frequency3*, all single site frequency sets are independent. In that case, all single site frequency set parameters are position dependent.

```
alphabet=Codon(letter=DNA)
genetic_code=Standard
nonhomogeneous.root_freq=Codon(frequency=GC())
```

builds a frequency set on codons, such that all sites frequencies follow the same GC frequency set model. The parameter name is *123_GC.theta*.

```
alphabet=Codon(letter=DNA)
genetic_code=Standard
nonhomogeneous.root_freq=Codon(frequency1=GC(),frequency2=GC(),\
                               frequency3=Fixed())
```

builds a frequency set on codons, such that first and second sites follow independent GC frequency sets, third site follows a Fixed frequency set. Then the parameters names are *1_GC.theta*, *2_GC.theta*.

Predefined codon frequencies are available, with a syntax similar to the one used in the PAML software. See above Codon Models section.

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.

`observedPseudoCount={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.

3.5.4 Rate across site distribution

`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:

`Constant` 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*.

3.5.5 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.

As long as their range match, parameters of several objects (models, root frequencies, rates, etc) can be linked.

For instance:

```
model1 = T92(theta=GC.theta, kappa=3)
model2 = T92(theta=0.39, kappa=T92.kappa_1)

nonhomogeneous.root_freq=GC
```

In the case of nonhomogeneous modelling, a specific syntax is available:

```
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.theta1->T92.theta1
```

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

```
model1=GTR(theta1=0.7)
nonhomogeneous.root_freq=Full(init=balanced)
nonhomogeneous.alias=Full.theta1->GTR.theta1_1
```

Note that this option is only available with the 'general' nonhomogeneous substitution models and will be ignored if used with "one_per_branch".

3.6 Discrete distributions

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

3.6.1 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>} [, ranges={vector<parametername[min;max]>}])`
: a discrete distribution with specific values (in *values*) and their respective non-negative probabibilities (in *probas*). The parameters are *V1*, *V2*, ..., *Vn* for all the values and the relative probabibility parameters are *theta1*, *theta2*, ..., *thetan-1*. Optional argument {ranges} sets the allowed ranges of values taken by the parameters; usage is like '`ranges=(V1[0.2;0.9],V2[1.1;999])`'.

`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.

3.6.2 Mixture Distributions

`Invariant(dist={distribution description}, p={float>0})`
: a Mixture of a given discrete distributution and a 0 Dirac. *p* is the probability of this 0 Dirac.

For example :

```
Invariant(dist=Gaussian(n=4,2,0.5),p=0.1)
```

builds a mixture of a gaussian distribution with 4 categories (and probability 0.9) and a 0 Dirac with probability 0.1. Overall, there are 5 categories. The parameters names are *Invariant.Gaussian.mu*, *Invariant.Gaussian.sigma*, *Invariant.p*.

`Mixture(probas={vector<double>}, dist1={distribution description}, ..., distn={distribution description})`
: a Mixture of discrete distributions with specific probabilities (in *probas*) and their respective desccriptions (in *probas*). The parameters are the relative probabibility parameters *theta1*, *theta2*, ..., *thetan-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),dist1=Beta(n=5,alpha=2,beta=3),\
                         dist2=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*.

3.7 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 positivie-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.backup.file = {path}`
: A backup file where parameters values are stored during optimization process. If this file exists when starting the optimization, parameter values will be set from the ones in this file. When optimization is finished, this file is renamed with suffixe ".def".

`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 there "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), 'BrLen' will ignore all branch lengths and 'Model' will ignore all model parameters.

  The '*' wildcard can be used, as in `*theta*` for all the parameters whose name has `theta` in it.

`optimization.constrain_parameter = {list<chars=interval>}`
: A list of parameters on which the authorized values are limited to a given interval.

```
optimization.constrain_parameter = YN98.omega = [-inf;1.9[,\
                        *theta* = [0.1;0.7[, BrLen*=[0.01;inf]
```

`optimization.tolerance = {float>0}`
: The precision on the log-likelihood to reach.

`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.

The resulting tree will be written to a file specified by the general tree writing options (Section 3.9 [WritingTrees], page 29).

3.8 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 3.2 [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).

3.9 Writing trees to files

`output.tree.file = {path}`
: The phylogenetic tree file to write to.

`output.tree.format = {Newick|Nexus|NHX}`
: The format of the output tree file.

Some programs may require that you write multiple trees to a file. The corresponding options are then:

`output.trees.file = {path}`
: The file that will contain multiple trees.

`output.trees.format = {Newick|Nexus|NHX}`
: The format of the output tree file.

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 3.2 [Sequences], page 5), specifying the model (see Section 3.5.1 [Model], page 8), and estimating parameters (see Section 3.7 [Estimation], page 27).

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.brlen.method = {method description}`
: Set how to initialize the branch lengths. Available methods include:

  `Input(midpoint_root_branch={boolean})`
  : Keep initial branch lengths as is. Additional argument specifies if the root position should be moved to the midpoint position of the branch containing it.

  `Equal(value={float>0})`
  : Set all branch lengths to the same value, provided as argumemt.

  `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.

`input.tree.check_root = {boolean}`
: Tell if the input tree should be checked regarding to the presence of a root. If set to yes (the default), rooted trees will be unrooted if a homogenous model is used. If not, a rooted tree will be fitted, which can lead to optimization issues in most cases. Use the non default option with care!

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|phyml}`
: 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. `phyml`: 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 `phyml` option shows the best performance.

`optimization.topology.nstep = {int>0}`
: Number of phyml 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:

`optimization.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.estimates = {{path}|none}`
: Write parameter estimated values.

`output.estimates.alias = {boolean}`
: Write the alias names of the aliased parameters instead of their values (default: true).

4.1.5 Bootstrap analysis

`bootstrap.number = {int>0}`
: Number of replicates. A reasonable value would be >= 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

Theses 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 3.2 [Sequences], page 5) and tree (see Section 3.3 [Tree], page 7), specifying the model (see Section 3.5.1 [Model], page 8) and writing sequence data (see Section 3.8 [WritingSequences], page 28).

The root sequence can be sampled from the model specification, with additional argument:

`number_of_sites = {int>0}`
: The number of site positions to simulate.

Or the root sequence can be built from a file of a sequence:

`input.sequence.file={path}`
: A sequence is be loaded, from which the simulation will be performed, or from which a root sequence can be sampled. (see Section 3.2 [Sequences], page 5).

`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.

  In this case, additional options are possible:

  `input.infos.rates = {string}`
  : Name of the column on which the rates are described (default: pr).

  `input.infos.states = {string}`
  : Name of the column on which the states are read (default: none, which means a random sequence).

`input.site.selection = {string}`
: used to sample from the given sequence (see Section 3.2 [Sequences], page 5).

Addition optional arguments include:

`input.tree.scale = {float}`
: An optional scaling factor for the branch length (default to 1.0)

`input.tree.method = {single|MS|CoaSim}`
: Format of input tree(s). By default, a single tree is expected ('single'). Ancestral recombination graphs (ARGs), in the form of multiple trees, can also be provided in the MS or CoaSim format. Note that in the case of MS, ARG are given for a certain number of sites, wich should be provided as additional argument (e.g. `MS(number_of_sites=100)`). The ARG will be unscaled according to the given size, and rescaled according to the given number of sites to simulate. ARG in CoaSim format are already in relative scale.

In addition, command line argument `--seed={int>0}` can be used to set the seed of the random generator.

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 3.2 [Sequences], page 5) and tree (see Section 3.3 [Tree], page 7), specifying the model (see Section 3.5.1 [Model], page 8) and writing sequence data (see Section 3.8 [WritingSequences], page 28).

Specific options are:

`input.tree.check_root = {boolean}`
: Tell if the input tree should be checked regarding to the presence of a root. If set to yes (the default), rooted trees will be unrooted if a homogenous model is used. If not, a rooted tree will be fitted, which can lead to optimization issues in most cases. Use the non default option with care!

`asr.method = {none|marginal}`
: Marginal is the only option for now. If set to "none", only nodes frequencies can be output.

`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. It 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 (prefix exp) (see Galtier & Gouy 1998) and a posteriori probabilities of ancestral states (prefix eb).

`output.nodes.add_extant = {boolean}`
: Tell if leaf nodes should be added to the output file.

4.4 BppMixedLikelihoods: Bio++ Site-Likelihoods Inside Mixed Models.

The BppMixedLikelihoods program uses the common syntax introduced in the previous section for setting the alphabet, loading the sequences (see Section 3.2 [Sequences], page 5) and tree (see Section 3.3 [Tree], page 7) and specifying the model (see Section 3.5.1 [Model], page 8).

Given a mixed parameter name of mixed model, or a mixed model made of several models, the BppMixedLikelihoods program computes site per site log-likelihoods of the several values of the parameter, or of the several sub-models of the mixture. If the mixed model is built on a parameter which value follows a distribution, and in an additional column – named "mean" – the a posteriori mean value of the paramater is computed.

Specific options are:

`output.likelihoods.file = {{path}}`
: Ouput file of the program (site specific log-likelihood, and mean of the mixed parameters, if any).

`likelihoods.model_number = {integer}`
: In case of a non-homogeneous modeling, the number of the mixed model which parameter or sub-models are considered.

`likelihoods.parameter_name = {string}`
: If the considered mixed model is built from a distribution on a parameter, the name of the parameter to be considered. In this case, an additional column is written, in which the average a posteriori value of the parameter is.

4.5 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 3.2 [Sequences], page 5) and tree (see Section 3.3 [Tree], page 7) and specifying the model (see Section 3.5.1 [Model], page 8, 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 3.7 [Estimation], page 27).

`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 30).

4.6 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 3.2 [Sequences], page 5) and tree (see Section 3.3 [Tree], page 7)).

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.

4.7 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 3.3 [Tree], page 7) and single-tree writing options for output.

`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 3.3 [Tree], page 7).

  `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.

`bootstrap.format = {int}`
: format of the bootstrap values. If positive, output values as percentages with the specified number of decimals. If negative, output the raw counts (number of trees).

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 outgroup 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 3.2 [Sequences], page 5) and writing the resulting data set (see Section 3.8 [WritingSequences], page 28). 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:

`input.alignment = {boolean}`
: Are the input sequence aligned? If so site selection and filtering is enabled and can be used to preprocess the input data.

`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 [[alphabet = DNA or RNA]]`
: Convert to proteins. The genetic code used for translation is set via the genetic_code option. Genetic code is set once for all sequences.

`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 stop codons in sequences. If sequences are aligned, stop codons will be replaced by gaps. The genetic code used for translation is set via the genetic_code option. Genetic code is set once for all sequences.

`RemoveEmptySequences`
: Remove all empty sequences (ie sequences with only gaps).

`RemoveColumnsWithStop`
: Remove all sites with at least one stop codon. The genetic code used for translation is set via the genetic_code option. Genetic code is set once for all sequences.

`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.

`GetCodonPosition(position={1|2|3})`
: Retrieve the given positions from codon sequences (aligned or not).

`FilterFromTree(tree.file={path}, tree.format={chars})`
: Get a subset of sequences based on a tree file. The order of sequences in the file will reflect the tree structure. All sequences which do not have a corresponding leaf in the tree, based on the sequence name, will be removed. This method can therefore be used for subsetting a list of sequences, and/or rearrange them in a more convenient manner.

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 BppAlnScore: Bio++ Alignment Scoring

This program compares two alignments and computes column scores. Scores are output to a text file, and/or can be used to generate a site selection, to be output in a mase file.

The two input alignments are specified using the input.sequences procedures (see Section 3.2 [Sequences], page 5), with suffixes ".test" for the first one, and ".ref" for the second. Scores will be computed for each column of the ".test" alignment.

Two scores are computed, following work by Thompson (1999):

column score (CS)
: is 1 if the column is found in the reference alignment, 0 otherwise.

sum-of-pairs score (SPS)
: is the proportion of pairs of residues which are also aligned in the reference alignment.

Specific options:

`output.scores = {path}`
: A text file where scores can be written, one row per column. If set to 'none', no file will be produced.

`output.mase = {path}`
: If not 'none', a Mase alignment will be generated, as a copy of the ".test" input alignment, with two sites selections names CS and SPS.

`output.sps_thresholds = {float}`
: The threshold to use for generating the site selection based on SPS score. All positions with at least the threshold value will be included in the selection.

`score.word_size = {int>0}`
: If alignment is for a word alphabet (typically codons), the word size can be specified in order to produce a compatible site selection. Please note that in this case, the alignment must not be loaded with the world alphabet, but the corresponding letter alphabet.

`score.phase = {int>0|chars}`
: Eather a number (1-based) stating the starting position for words, or the starting word. In this latter case, the first occurrence of the word in all sequences will be used to determine the phase.

4.11 BppPopStats: Bio++ Population Genetics Statistics

The `BppPopStats` program computes population genetics statistics from a sequence input alignement. It can compute glabal alignment statistics, as well as site-specific statistics. In the first case, results are output on screen or in a log file. In the second case, results are written in a table file, with one site per line. Statistics available also depend on the type on input data (coding or non-coding).

`BppPopStats` recognizes the standard options for alphabet and genetic code, in case a codon alphabet was specified. Sequences will be considered as coding if encoded with a codon alphabet, and non-coding otherwise.

4.11.1 Specific options

`input.sequence.file.ingroup = {path}`
: Path toward the file containing the ingroup sequences.

`input.sequence.format.ingroup = {string}`
: Alignment input format, following standard options.

`input.sequence.file.outgroup = {path}`
: Path toward the file containing the outgroup sequences.

`input.sequence.format.outgroup = {string}`
: Alignment input format, following standard options.

`input.sequence.file = {path}`
: Path toward the file containing all sequences. This option is only recognized if `input.sequence.file.ingroup` was not specified.

`input.sequence.format = {string}`
: Alignment input format, following standard options.

`input.sequence.outgroup.index = {[int>0]}`
: Vector of positions indicating the positions of the outgroup sequences in the alignment. This option is only recognized if `input.sequence.file.ingroup` was not specified.

`input.sequence.outgroup.name = {[string]}`
: Vector of sequence names indicating the positions of the outgroup sequences in the alignment. This option is only recognized if `input.sequence.file.ingroup` was not specified.

`input.sequence.stop_codons_policy = Keep|RemoveIfLast|RemoveAll`
: Tells what to do with positions containing at least one stop codon: keep them, remove them only if they are at the end of the alignment, or remove them all.

`estimate.kappa = {[boolean]}`
: Tells if the ratio of transitions / transversion should be estimated from the data and used for further analyses. If yes, kappa will be estimated by maximum likelihood using a model of (codon) sequence evolution.

`estimate.ancestor = {[boolean]}`
: If an outgroup sequence is present, it will be used to estimate the ancestral allele for each polymorphic position. A model of (codon) sequence evolution will be used with a marginal ancestral state reconstruction method.

`estimate.sample_ingroup = {[bollean]}`
: Tell if a random subset of ingroup sequences should be used to fit model (speeds up calculations in case of large data sets).

`estimate.sample_ingroup.size = {[integer]}`
: Number of ingroup sequences to sample.

`pop.stats = {[string]}`
: The list of statistics to compute. The next section describes all available statistics.

`logfile = {path}`
: Optional file where to output results.

4.11.2 Available statistics

`SiteFrequencies`
: Output the number of segregating sites as well as the number of singletons.

`Watterson75`
: Compute Watterson's nucleotide diversity estimator (theta, averaged per site).

`Tajima83`
: Compute Tajima's nucleotide diversity estimator (pi, averaged per site).

`TajimaD`
: Compute Tajima's D. If a codon alignment is specified (and alphabet is set to codon type), the `positions` argument further allows to compute Tajima's D on synonymous sites only (`positions=synonymous`), or non-synonymous sites (`positions=non-synonymous`). Default is to use all sites (`positions=all`).

`FuAndLiDStar | FuAndLiFStar`
: Compute Fu and Li's (1993) D and F statistics. If argument `tot_mut` is set to yes, then the total number of mutations is used in the calculation, instead of the number of segregating sites (default). The `positions` argument further allows to compute Fu and Li's statistics on synonymous sites only (`positions=synonymous`), or non-synonymous sites (`positions=non-synonymous`). Default is to use all sites (`positions=all`).

`PiN_PiS`
: For codon sequences only, obviously. Compute nucleotide diversity at synonymous and non-synonymous site, the number of synonymous and non-synonymous sites, as well as the weighted ratio (PiN / NbN) / (PiS / NbS).

`dN_dS`
: For codon sequences only. Build the consensus sequence of both ingroup and outgroup alignments and fit a Yang and Nielsen model of codon sequence evolution with a maximum likelihood approach. Reports the estimated parameters omega (dN / dS ratio) and kappa (transitions / transversions ratio), as well as the divergence between the two sequences.

`MKT`
: Compute the MacDonald-Kreitman table, for codon sequences with outgroup.

`CodonSiteStatistics`
: Generate a table with codon-site specifics statistics, including:

  - Whether the site is complete or not
  - Number of distinct states
  - Minor allele frequency
  - Major allele frequency
  - Minor allele
  - Major allele
  - State in the first outgroup sequence, if any
  - Ancestral state, if computed
  - Mean number of synonymous positions for polymorphism
  - Whether the site is synonymous polymorphic
  - Whether the site is four-fold degenerated
  - Non-synonymous diversity (piN)
  - Synonymous diversity (piS)
  - Mean number of synonymous positions for divergence
  - dN, if an outgroup is available
  - dS, if an outgroup is available

  The `output.file` argument allows to specify the output file (mandatory).

4.12 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.