GARLI v0.96 manual (May 2008)
Derrick J. Zwickl
garli{dot}support{at}gmail{dot}com

GARLI manual

(program version 0.96, manual revision 1)

Table of Contents

Note: all of the information contained in this manual also appears on the GARLI support wiki at http://www.nescent.org/wg/garli. More up to date information, especially FAQ items, may also be available there. When possible, I suggest using that online resource and saving some trees by not printing this manual.

New in version 0.96

1. Rigorous reading of Nexus datasets using Paul Lewis and Mark Holder's Nexus Class Library (NCL).
2. Ability to read Nexus starting trees using NCL.
3. Ability to perform inference under amino acid and codon-based models of sequence evolution (datatype = aminoacid, datatype = codon).
4. Ability to specify multiple search replicates in a single config file (searchreps = #).
5. Ability to specify outgroups for orientation of inferred trees (outgroup = # # #).
6. Ability to use backbone as well as normal topological constraints.
7. Ability to create fast likelihood stepwise addition starting trees (streefname = stepwise).
8. MPI version that spreads a specified number of serial runs across processors using a single config file, writing output to different output files (for example, to do 25 bootstrap replicates simultaneously on each of 8 processors).
9. Ability to perform nucleotide inference using any sub-model of the General Time-Reversible model (GTR), in addition to all of the common “named” models (K2P, HKY, etc).
10. Speed increases for non-parametric bootstrapping.

GARLI v0.96 manual (May 2008)
Derrick J. Zwickl
garli{dot}support{at}gmail{dot}com

Short algorithm description

GARLI (Genetic Algorithm for Rapid Likelihood Inference) performs phylogenetic searches on aligned sequence datasets using the maximum-likelihood criterion. Version 0.96 of the program allows tree inference using amino acid and codon-based models, in addition to the standard nucleotide models available in previous versions. Available substitution models include:

Nucleotide models: All models nested within the General Time Reversible (GTR) model, optionally with discrete gamma distributed rate heterogeneity and/or an inferred proportion of invariable sites.

Amino acid models: Many of the well known fixed amino acid rate matrices (Dayhoff, Jones, WAG, mtRev, mtmam), with either fixed or observed (aka "+F") amino acid frequencies, and discrete gamma distributed rate heterogeneity and/or an inferred proportion of invariable sites

Codon models: The basic Goldman and Yang (1994) model and other related models, with a number of options for codon frequencies (equal, "F1x4", "F3x4", observed) and one or more estimated non-synonymous rate categories (aka dN/dS or ω parameters)

GARLI is loosely based on the program GAML, by Paul O. Lewis (1998). It uses a stochastic genetic algorithm-like approach to simultaneously find the topology, branch lengths and substitution model parameters that maximize the log-likelihood (lnL). This involves the evolution of a population of solutions termed individuals, with each individual encoding a tree topology, a set of branch lengths and a set of model parameters. Each individual is assigned a fitness based on its lnL score. Each generation random mutations are applied to some of the components of the individuals, and their fitnesses are recalculated. The individuals are then chosen to be the parents of the individuals of the next generation, in proportion to their fitnesses. This process is repeated many times, and the population of individuals evolves toward higher fitness solutions. Note that the highest fitness individual is automatically maintained in the population, ensuring that it is not lost due to chance (genetic drift).

The mutation types used by GARLI are divided into three types: topological mutations, model parameter mutations and branch-length mutations. Topological mutations consist of the

standard NNI and SPR rearrangement types, as well as a localized form of SPR in which the pruned subtree may only be reattached to branches within a certain radius of its former location. Topological mutations are followed by a variable amount of branch-length optimization. Model mutations simply choose one of the model parameters and multiply it by a gamma-distributed variable with mean 1.0. When branch-length mutations are performed, a number of branches are chosen and each has its current length multiplied by a different gamma-distributed variable.

A quick guide to running the program

Step 1: Get your dataset ready

Version 0.96 accepts aligned data in a variety of file formats (Nexus, Phylip, Fasta). This version also includes rigorous support for features of the Nexus format such as assumptions blocks, which may include commands to exclude alignment columns or entire sequences.

Step 2: Tell GARLI what you want it to do

Graphical OS X version – (NOTE – GUI VERSION NOT YET AVAILABLE FOR VERSION 0.96) Double click on the icon to start the program. On the File menu, choose Open and select your dataset. You will now see an interface with several tabs that you can use to tell GARLI what to output and how to run. Default values will already be entered, and they typically work well (at least for an initial exploratory run). The rest of this manual gives details on what the settings mean, which you might want to alter, and why.

All other versions – Most GARLI versions have no interface at all. All directions to the program are provided through a text-based configuration file, which by default is named **garli.conf**. You will need to open this file in a text editor (Textedit, BBedit, TextWrangler, Notepad, vi, emacs, etc) to make changes. In the file you will see a series of fairly cryptically named options. The only thing that *must* be changed in the file is to enter your dataset name as the **datafname.** You should also change the **ofprefix** setting to give the program a prefix to be added to the beginning of output filenames. The default values for most settings should work well, at least for an initial exploratory run. The rest of this manual gives details on what the settings mean, which you might want to alter, and why. Make sure that you save the configuration file after making any changes.

Step 3: Run the program

Graphical OS X version – Simply press the **Run** button. The program can be terminated prematurely, but should ideally be allowed to decide when to stop on its own.

Windows version – Double-click on the executable to start it. It will automatically read the garli.conf file and start. (If you like, you may also start the windows version from the command line, which allows for executing batch scripts. See the FAQ.)

All other versions – Linux and non-graphical OS X versions must be started from the command line. From a terminal or shell window, get to the directory containing the executable and your dataset. Type "./Garli0.95" (or whatever the executable you have is named). The "./" is important, and tells the computer to look for the executable in the current directory. If you know what you are doing, you may of course put the executable somewhere in your path and omit the "./". (See the FAQ for details on performing multiple runs in batch)

Step 4: Interpret results

A number of important files are created during and after a run. Assuming that the output file prefix is "run1" ("ofprefix = run1" in the config file), then the files will be:

- run1.screen.log – This contains all information output to the screen during the run. Things of interest include a summary of the chosen model of sequence evolution, the final model parameter estimates and a summary of the results of multiple search replicates.
- run1.best.tre – This will contain a Nexus trees block with the best scoring result across all search replicates, with optimized branch lengths. For nucleotide data it will contain a Nexus PAUP block that can be used to load GARLI's model parameter estimates into PAUP.
- run1.all.best.tre – If multiple search replicates were done (searchreps > 1 in the config file), this will contain a Nexus trees block with the best tree found by each of the individual replicates.

You should always either perform multiple search replicates within a single program execution (searchreps > 1 in the config file), or do multiple program executions to verify the results that you obtain. If doing multiple runs, be sure to first change the **ofprefix** setting in the config file (or the "Prefix for output files" in the GUI version) so that you don't overwrite your previous results. The search algorithm of GARLI is stochastic, so it is entirely possible that another run

with the same settings could give a different result (although hopefully not). You may also try specifying a starting tree or changing other settings. Look through the FAQ for more info.

Serial FAQ

1. **How many generations/seconds should I run for?** This is dataset specific, and there is no way to tell in advance. It is recommended to set the maximum generations and seconds to very large values (>$1x10^6$) and use the automated stopping criterion (see **enforcetermconditions** in the settings list below). Note that the program can be stopped gracefully at any point by pressing Ctrl-C, although the results may not be fully optimal at that point.
2. **How many runs/search replicates should I do?** That somewhat depends on how much time/computational resources you have. You should ALWAYS do multiple searches. If you perform a few runs or replicates and get very similar trees/lnL scores (ideally within about one lnL of each other), that should give you some confidence that the program is doing a good job searching and is find the best or nearly best topology, and it suggests that you don't need to do many more searches. If there is a lot of variation between runs,try using different starting tree options (see the next FAQ entry) and choose the best scoring result that you obtain. Note that the program is stochastic, and runs performed with exactly the same starting conditions and settings (but different random number seeds) may give different results. You may also try changing some of the search parameters to make each search replicate more intensive (see further FAQ entries).
3. **Should I use random starting topologies, stepwise-addition starting topologies or provide starting topologies myself?**
   - Unless the number of sequences in your dataset numbers in the hundreds, it is recommended to perform multiple searches with both random (**streefname** = random) and stepwise-addition (**streefname** = stepwise) starting trees.
   - For datasets consisting of up to several hundred sequences, searches using a random starting tree often perform well (although they have slightly longer runtimes). Because the search starts from very different parts of the search space, getting consistent results from random starting trees provides good evidence that the search is doing a good job and really is finding the best trees. For datasets of more than a few hundred sequences, random starting trees sometimes perform quite poorly.

- The creation of fast ML stepwise-addition trees to begin searching from is a new feature of version 0.96. The quality of stepwise-addition starting trees can be controlled with the **attachmentspertaxon** setting. This allows the creation of starting trees that fall somewhere between completely random and very optimal. See the description of the **attachmentspertaxon** setting.
- Providing your own starting trees should not usually be necessary with version 0.96 of the program (because of the stepwise-addition option), but it might be helpful on datasets consisting of hundreds of sequences, where the creation of the stepwise-addition tree itself may take quite a long time.
- Version 0.96 also allows user-specified starting trees to contain polytomies (not be fully bifurcating), so for example a parsimony strict consensus tree could be used to get the search in the right ballpark to start with without biasing it very much. Before searching, a polytomous tree will be arbitrarily resolved, with the resolution being different for each search replicate or program execution.

4. **What is the proper format for specifying a starting topology?** The tree should be contained in a separate file (with that filename specified on the **streefname** line of the configuration file) either in a Nexus trees block, or in standard Newick format (parenthetical notation). Note that the tree description may contain either the taxon numbers (corresponding to the order of the taxa in the dataset), or the taxon names. The tree can optionally contain branch lengths. As of version 0.96 it can also have polytomies. If multiple trees are contained in a single starting tree file, they will be used in order to start each successive search replicate (if **searchreps** > 1). See the **streefname** configuration entry for more details.
5. **Should I specify a starting topology with branch lengths?** It doesn't appear to make much of a difference, so I would suggest not doing so. Note that it is probably NOT a good idea to provide starting branch lengths estimated under a different likelihood model or by Neighbor Joining. When in doubt, leave out branch lengths.
6. **How do I specify starting/fixed model parameter values?** Model parameter values are specified using a fairly cryptic scheme of specifying a single letter representing a particular parameter(s), followed by the value(s) of that parameter(s). The format of this string is

always the same (see the **streefname** entry for the details of the format), but it can be provided to GARLI in a few different ways:

- If your dataset is in Nexus format, include a Nexus GARLI block in the *same* file (in this case the values are automatically read and used, regardless of the **streefname** entry in the configuration file)
- If your dataset is in Nexus format, include a Nexus GARLI block in a *different* file, with that filename specified on the **streefname** line of the configuration file
- Regardless of the format of your dataset, you may include the raw parameter specification string in a different file, with that filename specified on the **streefname** line of the configuration file

  To use the provided parameter values as starting points, you don't need to do anything special in the configuration file. If you want to fix them, you'll need to set the configuration entries for those parameters to "fixed" (for example, **ratematrix** = fixed). See the descriptions of the model settings.

7. **Should I specify starting model parameters?** If you do not intend to fix the model parameters, specifying a starting model is generally of little help. One case in which you might want to specify starting parameter values would be when doing many search replicates or bootstrap replicates, in which case getting the starting values in the right ballpark can reduce total runtimes by an appreciable amount. If you do intend to fix the parameters at values obtained elsewhere or in a previous GARLI run, then you obviously must include the starting parameter valus. See the **streefname** configuration entry for details on how to specify model parameter values.
8. **Should I fix the model parameters?** The main reason one would fix parameters is to increase the speed of the search. Fixing model parameters results in a huge speed increase in some inference programs (such as PAUP*), but not very much in GARLI (approx. 10-30%). Unless you have good model estimates (under exactly the same model), do not fix them. One case in which you might want to fix parameter values would be in the case bootstrapping. You might want to estimate parameter values on the real data, and then fix those parameter values for the searches on each of the pseudo-replicate datasets.
9. **What DNA/RNA substitution models can I use?** Version 0.96 of the program now allows all possible submodels of the GTR (General Time Reversible) model, with or

without gamma distributed rate heterogeneity and a proportion of invariable sites. This is same set of models allowed by PAUP* (excepting site-specific rates) and represents the full set of models considered by the model selection program MODELTEST (http://darwin.uvigo.es/software/modeltest.html). See the "Model Specification Settings" section below.

10. **Do I need to perform statistical model selection when using GARLI?** Yes! Just as when doing an ML search in PAUP* or a Bayesian analysis in MrBayes, you should pick a model that is statistically justified given your data. You may use a program like MODELTEST (http://darwin.uvigo.es/software/modeltest.html) to do the testing. However, most good sized datasets (which is mainly what GARLI is designed to analyze) do support the use of the most complex time-reversible model, GTR with a class of invariable sites and gamma distributed rate heterogeneity ("GTR+I+G"). As of GARLI version 0.96, all of the models examined by MODELTEST can now be estimated. See the "Model Specification Settings" section below.
11. **For nucleotide models: Is the score that GARLI reports at the end of a run equivalent to what PAUP* would calculate after fully optimizing model parameters and branch lengths on the final topology?** It depends. The model implementations in GARLI are intentionally identical to those in PAUP, so in general the scores should be quite close, although PAUP* does more intensive optimization. If you've run GARLI for sufficiently long and not played with the optimization settings, the score will probably be within a few tenths of a log-likelihood unit from the score one would get optimizing in PAUP*. On very large trees it may be somewhat more. In some very rare conditions the score given by GARLI is better than that given by PAUP* after optimization, which appears to be due to PAUP* getting trapped in local branch-length optima. This should not be cause for concern. If you want to be absolutely sure of the lnL score of a tree inferred by GARLI, optimize it in PAUP*.
12. **For nucleotide models: Is the lnL score that GARLI reports at the end of a run comparable to the lnL scores reported by other ML search programs?** In general, you should not assume that lnL scores output by other ML search programs (such as PHYML and RAxML) are directly comparable to those output by GARLI, even if they apparently use the same model. To truly know which program has found a better tree you will need to

score and optimize the resulting trees using a single program, under the same model. Because of the way that optimization is done in the various programs, PHYML, RAxML and PAUP* are all probably better suited for this final optimization and scoring of a fixed tree than GARLI is. Also see the previous question.

13. **What amino acid models can I use?** Amino acid analyses are typically done using fixed rate matrices that have been estimated on large datasets and published. Typically the only model parameters that are estimated during tree inference relate to the rate heterogeneity distribution. Each of the named matrices also has corresponding fixed amino acid frequencies, and a given matrix can either be used with those frequencies or with the amino acid frequencies observed in your dataset. Amino acid models may be used with the same forms of rate heterogeneity available for nucleotide models (gamma-distributed rate heterogeneity and a proportion of invariable sites). See the model specification settings section for more details on amino acid models.
14. **How do I choose which amino acid model to use?** As with choosing a nucleotide model, your choice of an amino acid model should be based on some measure of how well the available models fit your data. The program PROTTEST (http://darwin.uvigo.es/software/prottest.html) does for amino acid models what MODELTEST does for nucleotide models, testing a number of amino acid models and helping you choose one. Note that although GARLI can internally translate aligned nucleotide sequences into amino acids and analyze them at that level, to use PROTTEST you will need to convert your alignment into a Phylip formatted amino acid alignment first.
15. **What codon models can I use?** The codon models that can be used are related to the Goldman and Yang (1994) model. See the codon section of the model specification settings for a discussion of the various options.
16. **How do I choose which codon model to use?** I don't currently have a good answer for this. The codon models should probably be considered experimental at the moment. Experiments to investigate the use of codon models for tree inference on large datasets are underway, and I should eventually have some general guidelines on how best to apply them. Feel free to give them a try with your data.
17. **Which GARLI settings should I play around with?** Besides specifying your own dataset, most settings don't need to be tinkered with, although you are free to do so if you

understand what they do. Settings that SHOULD be set by the user are **ofprefix**, **availablememory**, **stopgen**, **stoptime** and **genthreshfortopoterm**. If you want to tinker further, you might try changing **uniqueswapbias**, **nindiv**, **selectionintensity**, **limsprrange** **startoptprec**, **minoptprec** and **numberofprecisionreductions**. In general, using a different starting topology tends to have more of an effect on the results than any of these settings do.

18. **Can I specify alignment columns of my data matrix to be excluded?** As of version 0.96, yes. This is done through an "exset" command in a Nexus assumptions block, included in the same file as a Nexus data matrix. For example, to exclude characters 1-10 inclusive and character 20, the block would look like this:

    Begin assumptions;
    exset * myExsetName = 1-10 20;
    end;

    The * means to automatically apply the exset (otherwise the command simply defines the exset), and the exset name doesn't matter. Note that this assumes that the file has only one characters or data block, and that the characters block is not named. If you use Mesquite to edit your data or visualize your alignment, any characters that you exclude there will automatically be written to an assumptions block in the file and will be read by GARLI. (Another option for removing alignment columns is to use PAUP*. Simply execute your dataset in PAUP*, exclude the characters that you don't want, and then export the file to a new name. The new file will include only the columns you want.)
19. **How do I specify a topological constraint?** In short, this requires deciding which branches (or bipartitions) you would like to constrain, specifying those branches in a file and telling GARLI where to find that file. See the **constraintfile** option below for details on constraint formats.
20. **Why might I want to specify a topological constraint?** There are two main reasons: to reduce the topology search space or to perform hypothesis testing (such as parametric bootstrapping). For large datasets in which you are certain of some groupings, it may help the search to constrain a few major groups. Note that if constraints are specified without a starting tree, GARLI will create a random or stepwise-addition tree that is compatible with those constraints. This may be an easy way of improving searching without the potential

bias of using a given starting tree. A discussion of parametric bootstrapping (sometimes called the SOWH test) is out of the scope of this manual. It is a method of testing topological null hypotheses with a given dataset through simulation. See: Huelsenbeck et. al, (1996). Other statistical tests of tree topologies (attempting to answer the question "Is topology A significantly better than topology B") are nicely reviewed in Goldman et al. (2000).

21. **How do I perform a non-parametric bootstrap in GARLI?** Set up the config file as normal, and set the **bootstrapreps** setting to the number of replicates you want. The program will perform searches on that number of bootstrap reweighted datasets, and store the best tree found for each replicate dataset in a single file called <**ofprefix**>.boot.tre. The trees in this file will need to be read into a program such as PAUP* (or CONSENSE, or Bootscore) to do a majority-rule consensus and obtain your bootstrap support values. Note that in version 0.96 you can now specify searchreps > 1 while bootstrapping to perform multiple searches on each bootstrap resampled dataset. The best tree across across all searches replicates for each bootstrapped dataset will be written to the bootstrap file. See the **bootstrapreps** configuration entry for more info.
22. **How do I use checkpointing (i.e., stop and later restart a run)?** Set the **writecheckpoints** option to "1" before doing a run. If the run is stopped for some reason (intentionally or not), it can be restarted by changing the **restart** option to "1" in the config file and executing the program. DO NOT make any other changes to the config file before attempting a restart, or bad things may happen. See the **writecheckpoints** and **restart** options below.
23. **What are the differences between the Graphical (GUI) OS X version and other versions?** (NOTE THAT A GUI VERSION OF GARLI 0.96 HAS NOT YET BEEN RELEASED) The main differences are in how the user interacts with the program. The GUI version changes the cryptic option names detailed below into normal English. If you hold your mouse pointer over an option in the GUI it will give you a description of what that option does (generally taken directly from this manual). There may be some options that are not available in the GUI.
24. **Can I use GARLI to do batches of runs, one after another?** Yes, any of the non-GUI versions can do this. First create a different config file for each run you need to do, and

name them something like run1.conf, run2.conf, etc. Assuming that the GARLI executable is named Garli0.96 and is in the current directory, you may then make a shell script that runs each config file through the program like this:

./Garli0.96 –b run1.conf

./Garli0.96 –b run2.conf

etc.

The "–b" tells the program to use batch mode and to not expect user input before terminating. The details of making a shell script are beyond the scope of this manual, but you can find help online or ask your nearest Unix guru.

25. **Should I use a multi-threaded (openMP) version of GARLI if I'm using a computer with multiple processors/cores?** The multi-threaded versions will increase the speed of runs by approximately 1.2 to 1.8 times, but will otherwise give results identical to those obtained with the normal version (i.e., the search algorithm is exactly the same). It will perform the best when there are many columns in the alignment, or when using amino acid or codon models. This does not, however, mean that running this version is the best use of computing resources. In particular, if you intend to do multiple search replicates or bootstrap replicates, simply running two independent executions of the program will give a speedup of nearly 2 times, and will therefore get a given number of searches done more quickly than a multithreaded version. One case in which the multithreaded version may be of particular use is when analyzing extremely large datasets for which the amount of memory that would be required for two simultaneous executions of the program is near or greater than the amount of memory installed in the system. Note that the multi-threaded versions by default will use all of the cores/processors that are available on the system. To change this, you can set the OMP_NUM_THREADS environment variable (you can find information on how to do that online). Note that the performance of the multithreaded version when it is only using one processor or core is actually worse than the normal version, so when in doubt use the normal version.

Descriptions of GARLI configuration settings

The format for these configuration settings descriptions is generally:

entryname (possible values, **default value in bold**) – description.

Entries and options that are new in version 0.96 are <u>underlined</u>.

GARLI v0.96 manual (May 2008)
Derrick J. Zwickl
garli{dot}support{at}gmail{dot}com

General settings

datafname – Name of the file containing the aligned sequence data. Formats accepted are **PHYLIP**, **NEXUS** and **FASTA**. Version 0.96 allows robust reading of Nexus formatted datasets using the Nexus Class Library. This accommodates things such as interleaved alignments and exclusion sets (exset's) in Nexus assumptions blocks (see the FAQ for an example of exset usage). Use of Nexus datafiles is recommended, and some PHYLIP and FASTA formatted files may not read properly.

constraintfile – Name of the file containing any topology constraint specifications, or "none" if there are no constraints. The easiest way to explain the format of the constraint file is by example. Consider a dataset of 8 taxa, in which your constraint consists of grouping taxa 1, 3 and 5. You may specify either positive constraints (inferred tree MUST contain constrained group) or negative constraints (also called converse constraints, inferred tree CANNOT contain constrained group). These are specified with either a '+' or a '-' at the beginning of the constraint specification, for positive and negative constraints, respectively.

- For a positive constraint on a grouping of taxa 1, 3 and 5:
  +((1,3,5), 2, 4, 6, 7, 8);
- For a negative constraint on a grouping of taxa 1, 3 and 5:
  -((1,3,5), 2, 4, 6, 7, 8);
- Note that there are many other equivalent parenthetical representations of these constraints.
- Multiple groups may be positively constrained, but currently only a single negatively constrained group is allowed.
- Multiple constrained groupings may be specified in a single string:
  +((1,3,5), 2, 4, (6, 7), 8);
  or in two separate strings on successive lines:
  +((1,3,5), 2, 4, 6, 7, 8);
  +(1,3,5, 2, 4, (6, 7), 8);
- Constraint strings may also be specified in terms of taxon names (matching those used in the alignment) instead of numbers.
- Positive and negative constraints cannot be mixed.
- GARLI also accepts another constraint format that may be easier to use in some cases. This involves specifying a single branch to be constrained with a string of '*' (asterisk) and '.'

(period) characters, with one character per taxon. Each taxon specified with a '*' falls on one side of the constrained branch, and all those specified with a '.' fall on the other. This should be familiar to anyone who has looked at PAUP* bootstrap output.

With this format, a positive constraint on a grouping of taxa 1, 3 and 5 would look like this:

+*.*.*…

or equivalently like this:

+.*.*.***

With this format each line only designates a single branch, so multiple constrained branches must be specified as multiple lines in the file.

- New in version 0.96, the program now allows "backbone" constraints, which are simply constraints that do not include all of the taxa. For example if one wanted to constrain (1, 3, 5) but didn't care where taxon 6 appeared in the tree, this string could be used:

  +((1,3,5), 2, 4, 7, 8);

  Nothing special needs to be done to identify this as a backbone constraint, simply leave out some taxa.

streefname (random, **stepwise**, <filename>) – Specifies where the starting tree topology and/or model parameters will come from. The tree topology may be a completely random topology (constraints will be enforced), a tree provided by the user in a file, or a tree generated by the program using a fast ML stepwise-addition algorithm (see **attachmentspertaxon** below). Starting or fixed model parameter values may also be provided in the specified file, with or without a tree topology. Some notes on starting trees/models:

- New in version 0.96: Nexus starting trees are allowed.
- New in version 0.96: Specified starting trees may have polytomies, which will be arbitrarily resolved before the run begins.
- Starting tree formats:
  - Plain newick tree string (with taxon numbers or names, with or without branch lengths)
  - NEXUS trees block. The trees block can appear in the same file as a NEXUS data or characters block that contains the alignment, although the same filename should then be specified on both the datafname and streefname lines.

- If multiple trees appear in the specified file and multiple search replicates are specified (see **searchreps** setting), then the first tree is used in the first replicate, the second in the second replicate, etc.
- Model parameter specification strings: Each model parameter is specified by a letter representing the parameter type, followed by the value or values assigned. For example:
  r 1.4 3.4 0.55 1.09 4.94 e 0.297 0.185 0.213 0.305 a 0.66 p 0.43
  specifies starting values for the relative rate matrix parameters (in the order AC, AG, AT, CG, CT), the equilibrium base frequencies (in the order A, C, G, T), the alpha shape parameter of the gamma rate-heterogeneity distribution and the proportion of invariable sites. The format and letter codes are:
  - relative rate matrix parameters (nucleotide or codon models):
    r <AC rate> <AG rate> <AT rate> <CG rate> <CT rate>
  - equilibrium state frequencies (nucleotide models or amino acid models):
    e <4 or 20 state frequency values in alphabetical order>
  - alpha shape parameter of discrete gamma rate heterogeneity distribution (nucleotide or amino acid models):
    a <value>
  - proportion of invariable sites (nucleotide or amino acid models):
    p <value>
  - omega (dN/dS) parameters (codon models):
    o <omega value> (for a single dN/dS class model, i.e., **ratehetmodel** = none)
    o <omega0 value> <omega0 proportion> <omega1 value> <omega1 proportion> etc...
    (for multiple dN/dS class models, i.e., **ratehetmodel** = nonsynonymous)
- The parameter value string as defined above can be used in two ways: either within a GARLI Nexus block, or as a raw string in a file.
  - An example GARLI block is just this:

    ```
    begin GARLI;
    r 1.4 3.4 0.55 1.09 4.94 e 0.297 0.185 0.213 0.305 a 0.66 p 0.43;
    end;
    ```

    The block can either appear in the same file as a Nexus dataset, or in a separate Nexus file who's name is specified on the **streefname** line of the configuration file.

- As a raw string, the model specification can optionally be followed by a starting tree topology:

  r 1.4 3.4 0.55 1.09 4.94 e 0.297 0.185 0.213 0.305 a 0.66 p 0.43 ((((((140:……etc

- Note that to actually fix the provided parameter values during a search, you will also need to properly set the model configuration entries in the configuration file (e.g., **ratematrix** = fixed)
- Example starting model/tree files are provided with the program package.

attachmentspertaxon (1 to infinity, **50**) – The number of attachment branches evaluated for each taxon to be added to the tree during the creation of an ML stepwise-addition starting tree. Briefly, stepwise addition is an algorithm used to make a tree, and involves adding taxa in a random order to a growing tree. For each taxon to be added, a number of randomly chosen attachment branches are tried and scored, and then the best scoring one is chosen as the location of that taxon. The **attachmentspertaxon** setting controls how many attachment points are evaluated for each taxon to be added. A value of one is equivalent to a completely random tree (only one randomly chosen location is evaluated). A value of greater than 2 times the number of taxa in the dataset means that all attachment points will be evaluated for each taxon, and will result in very good starting trees (but may take a while on large datasets). Even fairly small values (< 10) can result in starting trees that are much, much better than random, but still fairly different from one another.

ofprefix (text) – Prefix of all output filenames, such as log, treelog, etc. Change this for each run that you do or the program will overwrite previous results.

randseed (-1 or positive integers, **-1**) – The random number seed used by the random number generator. Specify "–1" to have a seed chosen for you. Specifying the same seed number in multiple runs will give exactly identical results, if all other parameters and settings are also identical.

availablememory – Typically this is the amount of available physical memory on the system, in megabytes. This lets GARLI determine how much system memory it may be able to use to store computations for reuse. The program will be conservative and use at most about 90% of this value, although for typical datasets it will use much, much less. If other programs must be open or used when GARLI is running, you may need to reduce this value. This setting can also have a significant effect on performance (speed), but more is not always

better. When a run is started, GARLI will output the **availablememory** value necessary for that dataset to achieve each of the "memory levels" (from best to worst, termed "great", "good", "low", and "very low"). More memory is generally better because more calculations are stored and can be reused, but when the amount of memory needed for "great" becomes more than 512 megabytes or so, performance can be slowed because the operating system has difficulty managing that much memory. In general, chose an amount of memory that allows "great" when this is less than 512 MB, and if it is greater reduce the amount of memory into "good" or "low" as necessary. Avoid "very low" whenever possible. You can find the value is approximately optimal for your dataset by setting the **randseed** to some positive value (so that the searches are identical) and doing runs with various **availablememory** values. Typically it will only make at most a 30% difference, so it isn't worth worrying about too much.

logevery (1 to infinity, **10**) – The frequency with which the best score is written to the log file.

saveevery (1 to infinity, **100**) – If **writecheckpoints** or **outputcurrentbesttopology** are specified, this is the frequency (in generations) at which checkpoints or the current best tree are written to file.

refinestart (0 or 1, **1**) – Specifies whether some initial rough optimization is performed on the starting branch lengths and alpha parameter. This is always recommended.

outputcurrentbesttopology (0 or 1, **0**) – If true, the current best tree of the current search replicate is written to <**ofprefix**>.best.current.tre every **saveevery** generations. In versions before 0.96 the current best topology was always written to file, but that is no longer the case. Seeing the current best tree has no real use apart from satisfying your curiosity about how a run is going.

outputeachbettertopology (0 or 1, **0**) – If true, each new topology encountered with a better score than the previous best is written to file. In some cases this can result in *really* big files (hundreds of MB) though, especially for random starting topologies on large datasets. Note that this file is interesting to get an idea of how the topology changed as the searches progressed, but the collection of trees should NOT be interpreted in any meaningful way. This option is not available while bootstrapping.

enforcetermconditions (0 or 1, **1**) – Specifies whether the automatic termination conditions will be used. The conditions specified by *both* of the following two parameters must be met. See

the following two parameters for their definitions. If this is false, the run will continue until it reaches the time (**stoptime**) or generation (**stopgen**) limit. It is highly recommended that this option be used!

genthreshfortopoterm (1 to infinity, 2**0,000**) – This specifies the first part of the termination condition. When no new significantly better scoring topology (see **significanttopochange** below) has been encountered in greater than this number of generations, this condition is met. Increasing this parameter may improve the lnL scores obtained (especially on large datasets), but will also increase runtimes.

scorethreshforterm (0 to infinity, **0.05**) – The second part of the termination condition. When the total improvement in score over the last **intervallength** x **intervalstostore** generations (default is 500 generations, see below) is less than this value, this condition is met. This does not usually need to be changed.

significanttopochange (0 to infinity, **0.01**) – The lnL increase required for a new topology to be considered significant as far as the termination condition is concerned. It probably doesn't need to be played with, but you might try increasing it slightly if your runs reach a stable score and then take a very long time to terminate due to very minor changes in topology.

outputphyliptree (0 or 1, **0**) **–** Whether a phylip formatted tree files will be output in addition to the default nexus files for the best tree across all replicates (<ofprefix>.best.phy), the best tree for each replicate (<ofprefix>.best.all.phy) or in the case of bootstrapping, the best tree for each bootstrap replicate (<ofprefix.boot.phy>.

outputmostlyuselessfiles (0 or 1, **0**) – Whether to output three files of little general interest: the "fate", "problog" and "swaplog" files. The fate file shows the parentage, mutation types and scores of every individual in the population during the entire search. The problog shows how the proportions of the different mutation types changed over the course of the run. The swaplog shows the number of unique swaps and the number of total swaps on the current best tree over the course of the run.

writecheckpoints (0 or 1, **0**) – Whether to write three files to disk containing all information about the current state of the population every **saveevery** generations, with each successive checkpoint overwriting the previous one. These files can be used to restart a run at the last written checkpoint by setting the **restart** configuration entry.

restart (0 or 1, **0**) – Whether to restart at a previously saved checkpoint. To use this option the **writecheckpoints** option must have been used during a previous run. The program will look for checkpoint files that are named based on the **ofprefix** of the previous run. If you intend to restart a run, NOTHING should be changed in the config file except setting **restart** to 1. A run that is restarted from checkpoint will give exactly the same results it would have if the run had gone to completion.

outgroup (ougroup taxa numbers, separated by spaces) – This option allow for orienting the tree topologies in a consistent way when they are written to file. Note that this has NO effect whatsoever on the actual inference and the specified outgroup is NOT constrained to be present in the inferred trees. If multiple outgroup taxa are specified and they do not form a monophyletic group, this setting will be ignored. If you specify a single outgroup taxon it will always be present, and the tree will always be consistently oriented. To specify an outgroup consisting of taxa 1, 3 and 5 the format is this:

outgroup = 1 3 5

searchreps (1 to infinity, **2**) – The number of independent search replicates to perform during a program execution. You should always either do multiple search replicates or multiple program executions with any dataset to get a feel for whether you are getting consistent results, which suggests that the program is doing a decent job of searching. Note that if this is > 1 and you are performing a bootstrap analysis, this is the number of search replicates to be done per bootstrap replicate. That can increase the chance of finding the best tree per bootstrap replicate, but will also increase bootstrap runtimes enormously.

Model specification settings:

With version 0.96 there are now many more options dealing with model specification because of the inclusion of amino acid and codon-based models. The description of the settings will be broken up by data type. Note that in terms of the model settings in GARLI, "empirical" means to fix parameter values at those observed in the dataset being analyzed, and "fixed" means to fix them at user specified values. See the **streefname** setting above for details on how to provide parameter values to be fixed during inference.

datatype (nucleotide, aminoacid, codon-aminoacid, codon) – The type of data and model that is to be used during tree inference. The codon-aminoacid datatype means that the data will be supplied as a nucleotide alignment, but will be internally translated and analyzed using an

amino acid model. The codon and codon-aminoacid datatypes require nucleotide sequence that is aligned in the correct reading frame. In other words, all gaps in the alignment should be a multiple of 3 in length, and the alignment should start at the first position of a codon. If the alignment has extra columns at the start, middle or end, they should be removed or excluded with a Nexus exset (see the FAQ for an example of exset usage). The correct **geneticcode** must also be set.

Settings for datatype = nucleotide:

ratematrix (1rate, 2rate, **6rate**, fixed, custom string) – The number of relative substitution rate parameters (note that the number of free parameters is this value minus one). Equivalent to the "nst" setting in PAUP* and MrBayes. 1rate assumes that substitutions between all pairs of nucleotides occur at the same rate, 2rate allows different rates for transitions and transversions, and 6rate allows a different rate between each nucleotide pair. These rates are estimated unless the fixed option is chosen. New in version 0.96, parameters for any submodel of the GTR model may now be estimated. The format for specifying this is very similar to that used in the "rclass' setting of PAUP*. Within parentheses, six letters are specified, with spaces between them. The six letters represent the rates of substitution between the six pairs of nucleotides, with the order being A-C, A-G, A-T, C-G, C-T and G-T. Letters within the parentheses that are the same mean that a single parameter is shared by multiple nucleotide pairs. For example,

ratematrix = (a b a a b a)

would specify the HKY 2-rate model (equivalent to ratematrix = 2rate). This entry,

ratematrix = (a b c c b a)

would specify 3 estimated rates of subtitution, with one rate shared by A-C and G-T substitutions, another rate shared by A-G and C-T substitutions, and the final rate shared by A-T and C-G substitutions.

statefrequencies (equal, empirical, **estimate**, fixed) – Specifies how the equilibrium state frequencies (A, C, G and T) are treated. The empirical setting fixes the frequencies at their observed proportions, and the other options should be self-explanatory.

For datatype = nucleotide or aminoacid:

invariantsites (none, **estimate**, fixed) – Specifies whether a parameter representing the proportion of sites that are *unable* to change (i.e. have a substitution rate of zero) will be included. This is typically referred to as "invariant sites", but would better be termed "invariable sites".

ratehetmodel (none, **gamma**, gammafixed) – The model of rate heterogeneity assumed. "gammafixed" requires that the alpha shape parameter is provided, and a setting of "gamma" estimates it.

numratecats (1 to 20, **4**) – The number of categories of variable rates (not including the invariant site class if it is being used). Must be set to 1 if **ratehetmodel** is set to none. Note that runtimes and memory usage scale linearly with this setting.

For datatype = aminoacid or codon-aminoacid:

Amino acid analyses are typically done using fixed rate matrices that have been estimated on large datasets and published. Typically the only model parameters that are estimated during tree inference relate to the rate heterogeneity distribution. Each of the named matrices also has corresponding fixed amino acid frequencies, and a given matrix can either be used with those frequencies or with the amino acid frequencies observed in your dataset. This second option is often denoted as "+F" in a model description, although in terms of the GARLI configuration settings this is referred to as "empirical" frequencies. In GARLI the Dayhoff model would be specified by setting both the **ratematrix** and **statefrequencies** options to "dayhoff". The Dayhoff+F model would be specified by setting the **ratematrix** to "dayhoff", and **statefrequencies** to "empirical".

The following named amino acid models are implemented:

ratematrix/statefrequencies setting	reference
dayhoff	Dayhoff, Schwartz and Orcutt. 1978
jones	Jones, Taylor and Thornton (JTT), 1992
WAG	Whelan and Goldman, 2001
mtREV	Adachi and Hasegawa, 1996
mtmam	Yang, Nielsen and Hasegawa. 1998

Note that most programs allow either the use of a named rate matrix and its corresponding state frequencies, or a named rate matrix and empirical frequencies. GARLI technically allows the mixing of different named matrices and equilibrium frequencies (for example, wag matrix with jones equilibrium frequencies), but this is not recommended.

ratematrix (poisson, jones, dayhoff, wag, mtmam, mtrev) – The fixed amino acid rate matrix to use. You should use the matrix that gives the best likelihood, and could use a program like PROTTEST (very much like MODELTEST, but for amino acid models) to determine which fits best for your data. Poisson assumes a single rate of substitution between all amino acid pairs, and is a very poor model.

statefrequencies (equal, **empirical**, estimate, fixed, jones, dayhoff, wag, mtmam, mtrev) – Specifies how the equilibrium state frequencies of the 20 amino acids are treated. The "empirical" option fixes the frequencies at their observed proportions (when describing a model this is often termed "+F").

For datatype = codon:

The codon models are built with three components: (1) parameters describing the process of individual nucleotide substitutions, (2) equilibrium codon frequencies, and (3) parameters describing the relative rate of nonsynonymous to synonymous substitutions. The nucleotide substitution parameters within the codon models are exactly the same as those possible with standard nucleotide models in GARLI, and are specified with the **ratematrix** configuration entry. Thus, they can be of the 2rate variety (inferring different rates for transitions and transversions, K2P or HKY-like), the 6rate variety (inferring different rates for all nucleotide pairs, GTR-like) or any other sub-model of GTR. The options for codon frequencies are specified with the **statefrequencies** configuration entry. The options are to use equal frequencies (not a good option), the frequencies observed in your dataset (termed "empirical" in GARLI), or the codon frequencies implied by the "F1x4" or "F3x4" methods (using PAML's terminology). These last two options calculate the codon frequencies as the product of the frequencies of the three nucleotides that make up each codon. In the "F1x4" case the nucleotide frequencies are those observed in the dataset across all codon positions, while the "F3x4" option uses the nucleotide frequencies observed in the data at each codon position separately. The final component of the codon models is the nonsynonymous to synonymous relative rate parameters

(aka dN/dS or omega parameters). The default is to infer a single dN/dS value. Alternatively, a model can be specified that infers a given number of dN/dS categories, with the dN/dS values and proportions falling in each category estimated (**ratehetmodel** = nonsynonymous). This is the "discrete" or "M3" model in PAML's terminology.

ratematrix (1rate, **2rate**, 6rate, fixed, custom string) – This determines the relative rates of nucleotide substitution assumed by the codon model. The options are exactly the same as those allowed under a normal nucleotide model. A codon model with **ratematrix = 2rate** specifies the standard Goldman and Yang (1994) model, with different substitution rates for transitions and transversions.

statefrequencies (equal, empirical, f1x4, **f3x4**) - The options are to use equal codon frequencies (not a good option), the frequencies observed in your dataset (termed "empirical" in GARLI), or the codon frequencies implied by the "F1x4" or "F3x4" methods (using PAML's terminology). These last two options calculate the codon frequencies as the product of the frequencies of the three nucleotides that make up each codon. In the "F1x4" case the nucleotide frequencies are those observed in the dataset across all codon positions, while the "F3x4" option uses the nucleotide frequencies observed in the data at each codon position separately.

ratehetmodel (**none**, nonsynonymous) – For codon models, the default is to infer a single dN/dS parameter. Alternatively, a model can be specified that infers a given number of dN/dS categories, with the dN/dS values and proportions falling in each category estimated (ratehetmodel = nonsynonymous). This is the "discrete" or "M3" model of Yang et al. (2000).

numratecats – When **ratehetmodel** = nonsynonymous, this is the number of dN/dS parameter categories.

For datatype = codon or codon-aminoacid:

geneticcode (**standard**, vertmito, invertmito) – The genetic code to be used in translating codons into amino acids.

Derrick J. Zwickl
garli{dot}support{at}gmail{dot}com

Population Settings:

nindivs (2 to 100, **4**)- The number of individuals in the population. This may be increased, but doing so is generally not beneficial. Note that typical genetic algorithms tend to have much, much larger population sizes than GARLI's defaults.

holdover (1 to nindivs-1, **1**)- The number of times the best individual is copied to the next generation with no chance of mutation. It is best not to mess with this.

selectionintensity (0.01 to 5.0, **0.5**)- Controls the strength of selection, with larger numbers denoting stronger selection. The relative probability of reproduction of two individuals depends on the difference in their log likelihoods (ΔlnL) and is formulated very similarly to the procedure of calculating Akaike weights. The relative probability of reproduction of the less fit individual is equal to:

$$e^{-(selectionIntensity * \Delta lnL)}$$

In general, this setting does not seem to have much of an effect on the progress of a run. In theory higher values should cause scores to increase more quickly, but make the search more likely to be entrapped in a local optimum. Low values will increase runtimes, but may be more likely to reach the true optimum. The following table gives the relative probabilities of reproduction for different values of the selection intensity when the difference in log likelihood is 1.0

Selection intensity	Ratio of probabilities of reproduction
0.05	0.95:1.0
0.1	0.90:1.0
0.25	0.78:1.0
0.5	0.61:1.0
0.75	0.47:1.0
1	0.37:1.0
2	0.14:1.0

holdoverpenalty – (0 to 100, **0**) This can be used to bias the probability of reproduction of the best individual downward. Because the best individual is automatically copied into the next generation, it has a bit of an unfair advantage and can cause all population variation to be lost due to genetic drift, especially with small populations sizes. The value specified here is subtracted from the best individual's lnL score before calculating the probabilities of

reproduction. It seems plausible that this might help maintain variation, but I have not seen it cause a measurable effect.

stopgen – The maximum number of generations to run. Note that this supersedes the automated stopping criterion (see **enforcetermconditions** above), and should therefore be set to a very large value if automatic termination is desired.

stoptime – The maximum number of seconds for the run to continue. Note that this supersedes the automated stopping criterion (see **enforcetermconditions** above), and should therefore be set to a very large value if automatic termination is desired.

Branch-length optimization settings:

After a topological rearrangement, branch lengths in the vicinity of the rearrangement are optimized by the Newton-Raphson method. Optimization passes are performed on a particular branch until the expected improvement in likelihood for the next pass is less than a threshold value, termed the **optimization precision**. Note that this name is somewhat misleading, as the precision of the optimization algorithm is inversely related to this value (i.e., smaller values of the optimization precision lead to more precise optimization). If the improvement in likelihood due to optimization for a particular branch is greater than the optimization precision, optimization is also attempted on adjacent branches, spreading out across the tree. When no new topology with a better likelihood score is discovered for a while, the value is automatically reduced. The value can have a large effect on speed, with smaller values significantly slowing down the algorithm. The value of the optimization precision and how it changes over the course of a run are determined by the following three parameters.

startoptprec (0.005 to 5.0, **0.5**) – The beginning optimization precision.

minoptprec (0.001 to startoptprec, **0.01**) – The minimum allowed value of the optimization precision.

numberofprecreductions (0 to 100, **10**) – Specify the number of steps that it will take for the optimization precision to decrease (linearly) from **startoptprec** to **minoptprec**.

treerejectionthreshold (0 to 500, **50**) – This setting controls which trees have more extensive branch-length optimization applied to them. All trees created by a branch swap receive optimization on a few branches that directly took part in the rearrangement. If the difference in score between the partially optimized tree and the best known tree is greater than

treerejectionthreshold, no further optimization is applied to the branches of that tree. Reducing this value can significantly reduce runtimes, often with little or no effect on results. However, it is possible that a better tree could be missed if this is set too low. In cases in which obtaining the very best tree per search is not critical (e.g., bootstrapping), setting this lower (~20) is probably safe.

Settings controlling the proportions of the mutation types:

Each mutation type is assigned a prior *weight*. These values determine the expected proportions of the various mutation types that are performed. The primary mutation categories are *topology* (t)*, model* (m) and *branch length* (b). Each are assigned a prior weight (P_i) in the config file. Each time that a new best likelihood score is attained, the amount of the increase in score is credited to the mutation type responsible, with the sum of the increases (S_i) maintained over the last **intervallength** x **intervalstostore** generations. The number of times that each mutation is performed (N_i) is also tallied. The total weight of a mutation type is $W_i = P_i + (S_i/N_i)$. The proportion of mutations of type *i* out of all mutations is then

$$P(i) = \frac{W_i}{\sum_{j \in t,m,b} W_j}$$

The proportion of each mutation is thus related to its prior weight and the average increase in score that it has caused over recent generations. The prior weights can be used to control the expected (and starting) proportions of the mutation types, as well as how sensitive the proportions are to the course of events in a run. It is generally a good idea to make the topology prior much larger than the others so that when no mutations are improving the score many topology mutations are still attempted. You can look at the "problog" file to determine what the proportions of the mutations actually were over the course of a run.

topoweight (0 to infinity, **1.0**) The prior weight assigned to the class of topology mutations (NNI, SPR and limSPR).

modweight (0 to infinity, **0.05**) The prior weight assigned to the class of model mutations. Note that setting this at 0.0 fixes the model during the run.

brlenweight ((0 to infinity, **0.2**) The prior weight assigned to branch-length mutations.

The same procedure used above to determine the proportion of Topology:Model:Branch-Length mutations is also used to determine the relative proportions of the three types of topological mutations (NNI:SPR:limSPR), controlled by the following three weights. Note that the proportion of mutations applied to each of the model parameters is not user controlled.

randnniweight (0 to infinity, **0.1**) - The prior weight assigned to NNI mutations.

randsprweight (0 to infinity, **0.3**) - The prior weight assigned to random SPR mutations. For very large datasets it is often best to set this to 0.0, as random SPR mutations essentially never result in score increases.

limsprweight (0 to infinity, **0.6**) - The prior weight assigned to SPR mutations with the reconnection branch limited to being a maximum of **limsprrange** branches away from where the branch was detached.

intervallength (10 to 1000, **100**) – The number of generations in each interval during which the number and benefit of each mutation type are stored.

intervalstostore = (1 to 10, **5**) – The number of intervals to be stored. Thus, records of mutations are kept for the last (**intervallength** x **intervalstostore)** generations. Every **intervallength** generations the probabilities of the mutation types are updated by the scheme described above.

Settings controlling mutation details:

limsprrange (0 to infinity, **6**) – The maximum number of branches away from its original location that a branch may be reattached during a limited SPR move. Setting this too high (> 10) can seriously degrade performance.

meanbrlenmuts (1 to # taxa, **5**) - The mean of the binomial distribution from which the number of branch lengths mutated is drawn during a branch length mutation.

gammashapebrlen (50 to 2000, **1000**) - The shape parameter of the gamma distribution (with a mean of 1.0) from which the branch-length multipliers are drawn for branch-length mutations. Larger numbers cause smaller changes in branch lengths. (Note that this has nothing to do with gamma rate heterogeneity.)

gammashapemodel (50 to 2000, **1000**) - The shape parameter of the gamma distribution (with a mean of 1.0) from which the model mutation multipliers are drawn for model parameters

mutations. Larger numbers cause smaller changes in model parameters. (Note that this has nothing to do with gamma rate heterogeneity.)

uniqueswapbias (0.01 to 1.0, **0.1**) – In version 0.95, GARLI now keeps track of which branch swaps it has attempted on the current best tree. Because swaps are applied randomly, it is possible that some swaps are tried twice before others are tried at all. This option allows the program to bias the swaps applied toward those that have not yet been attempted. Each swap is assigned a relative weight depending on the number of times that it has been attempted on the current best tree. This weight is equal to (**uniqueswapbias**) raised to the (# times swap attempted) power. In other words, a value of 0.5 means that swaps that have already been tried once will be half as likely as those not yet attempted, swaps attempted twice will be ¼ as likely, etc. A value of 1.0 means no biasing. If this value is not equal to 1.0 and the **outputmostlyuseless** files option is on, a file called <**ofprefix>**.swap.log is output. This file shows the total number rearrangements tried and the number of unique ones over the course of a run. Note that this bias is only applied to NNI and limSPR rearrangements. Use of this option may allow the use of somewhat larger values of **limsprrange**.

distanceswapbias (0.1 to 10, **1.0**) – This option is similar to **uniqueswapbias**, except that it biases toward certain swaps based on the topological distance between the initial and rearranged trees. The distance is measured as in the **limsprrange**, and is half the the Robinson-Foulds distance between the trees. As with **uniqueswapbias**, **distanceswapbias** assigns a relative weight to each potential swap. In this case the weight is (**distanceswapbias**) raised to the (reconnection distance - 1) power. Thus, given a value of 0.5, the weight of an NNI is 1.0, the weight of an SPR with distance 2 is 0.5, with distance 3 is 0.25, etc. Note that values less than 1.0 bias toward more localized swaps, while values greater than 1.0 bias toward more extreme swaps. Also note that this bias is only applied to limSPR rearrangements. Be careful in setting this, as extreme values can have a very large effect.

Miscellaneous options:

bootstrapreps (0 to infinity, **0**) - The number of bootstrap reps to perform. If this is greater than 0, normal searching will not be performed. The resulting bootstrap trees (one per rep) will be output to a file named <ofprefix>.boot.tre. To obtain the bootstrap proportions they will then need to be read into PAUP* or a similar program to obtain a majority rule consensus. Note

that it is probably safe to reduce the strictness of the termination conditions during bootstrapping (perhaps halve **genthreshfortopoterm**), which will greatly speed up the bootstrapping process with negligible effects on the results.

resampleproportion (0.1 to 10, **1.0**) – When **bootstrapreps** > 0, this setting allows for bootstrap-like resampling, but with the psuedoreplicate datasets having a different number of alignment columns than the real data. Setting values < 1.0 is akin to jackknifing.

inferinternalstateprobs = (0 or 1, **0**) – Specify 1 to have GARLI infer the marginal posterior probability of each character at each internal node. This is done at the very end of the run, just before termination. The results are output to a file named <ofprefix>.internalstates.log.

References

Adachi, J., and M. Hasegawa. 1996. Model of amino acid substitution in proteins encoded by mitochondrial DNA. J. Mol. Evol. 42:459-468.

Dayhoff, M. O., R. M. Schwartz, and B. C. Orcutt. 1978. A model of evolutionary change in proteins. pp. 345-352. Atlas of protein sequence and structure, Vol 5, Suppl. 3. National Biomedical Research Foundation, Washington D. C.

Goldman, N., J. P. Anderson, and A. G. Rodrigo. 2000. Likelihood-based tests of topologies in phylogenetics. Syst. Biol. 49:652–670.

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

Huelsenbeck, J.P., D. M. Hillis, and R. Nielsen. 1996. A likelihood-ratio test of monophyly. Syst. Biol. 45:546–558.

Jones, D. T., W. R. Taylor, and J. M. Thornton. 1992. The rapid generation of mutation data matrices from protein sequences. CABIOS 8:275-282.

Lewis, P.O. 1998. A genetic algorithm for maximum-likelihood phylogeny inference using nucleotide sequence data. Mol. Biol. Evol. 15(3):277-283.

Whelan, S., and N. Goldman. 2001. A general empirical model of protein evolution derived from multiple protein families using a maximum likelihood approach. Mol. Biol. Evol. 18:691-699.

Yang, Z., R. Nielsen, and M. Hasegawa. 1998. Models of amino acid substitution and applications to mitochondrial protein evolution. Mol. Biol. Evol. 15:1600-1611.

Yang, Z., R. Nielsen, N. Goldman, and A. M. K. Pedersen. 2000. Codon-substitution models for heterogeneous selection pressure at amino acid sites. Genetics 155:431-449.

GARLI v0.96 manual (May 2008)
Derrick J. Zwickl
garli{dot}support{at}gmail{dot}com

Zwickl, D. J., 2006. Genetic algorithm approaches for the phylogenetic analysis of large biological sequence datasets under the maximum likelihood criterion. Ph.D. dissertation, The University of Texas at Austin.