r8s, version 1.70 User's Manual

(December 2004)

Michael J. Sanderson

Section of Evolution and Ecology, One Shields Avenue,
University of California, Davis, CA 95616, USA
email:mjsanderson@ucdavis.edu

Table of Contents

Introduction

This is a program for estimating absolute rates ("r8s") of molecular evolution and divergence times on a phylogenetic tree. It implements several methods for estimating these parameters ranging from fairly standard maximum likelihood methods in the context of global or local molecular clocks to more experimental semiparametric and nonparametric methods that relax the stringency of the clock assumption using smoothing methods. Its starting point is a given phylogenetic tree and a given set of estimated branch lengths (numbers of substitutions along each branch). In addition one or more calibration points can be added to permit scaling of rates and times to real units. These calibrations can take one of two forms: assignment of a fixed age to a node, or enforcement of a minimum or maximum age constraint on a node, which is generally a better reflection of the information content of fossil evidence. Terminal nodes are permitted to occur at any point in time, allowing investigation of rate variation in phylogenies such as those obtained from "serial" samples of viral lineages through time. Finally, it is possible to assign all divergence times (perhaps based on outside estimates of divergence times) and examine molecular rate variation under several models of smoothing.

The motivation behind the development of this program is two-fold. First, the abundance of molecular sequence data has renewed interest in estimating divergence times (e.g., Wray et al. 1996; Ayala et al. 1998; Kumar and Hedges 1998; Korber et al. 2000), but there is wide appreciation that such data typically show strong departures from a molecular clock. This has prompted considerable theoretical work aimed at developing methods for estimating rate and time in the absence of a clock (Hasegawa et al. 1989; Takezaki et al. 1995; Rambaut and Bromham 1998; Sanderson 1997; Thorne et al. 1998; Hulesenbeck et al. 2000; Kishino et al. 2001). However, software tools for this are still specialized and generally do not permit the addition of flexible calibration tools.

Second, there are surprisingly few estimates of absolute rates of molecular evolution constructed in a phylogenetic framework. Most estimates are based on pairwise distance methods with simple fixed calibration points (e.g., Wray et al. 1996; Kumar and Hedges 1998). This program is designed to facilitate estimation of rates of molecular evolution, particularly estimation of the *variability* in such rates across a tree.

Powerful tools are available for phylogenetic analysis of molecular data, such as the magnificent PAUP* 4.0 (Swofford 1999), but none of them is designed from the ground up to deal with *time*. The default molecular models are generally atemporal in the sense that they estimate branch lengths that consist of non-decomposable products of rate and time. PAUP* and PAML and other programs can, of course, enforce a molecular clock and reconstruct a tree in which the branch lengths are proportional to time, but the time of the root of such a tree is arbitrary. It can be scaled manually a posteriori by the user given the age of single fossil attached to a single node in the tree, but none of these programs permit the addition of several calibration points or minimum or maximum age constraints. Moreover, they do not consider models that are intermediate between the clock on the hand and completely unconstrained rate variation on the other hand (the default). This has proven extremely useful for tree-building but may be limited for the purposes of understanding rate variation and estimating divergence times.

This program does not implement all or even most methods now available for estimating divergence times. In particular, it does not implement highly parametric Bayesian approaches described by Thorne et al. (1998, or the later Kishino et al. 2001) or Huelsenbeck et al. (2000). It provides, as a benchmark, routines that implement a molecular clock. It also includes methods for analyzing user-specified models with *k* discrete rates (where *k* is small). These are particular useful for looking at correlates of rate variation. For example, one might assign all lineages leading to species of annual plants in a phylogeny one rate, and those leading to the perennial species a second rate.

However, the main thrust of the program is to implement methods that are nonparametric or semiparametric rather than parametric per se. By analogy to smoothing methods in regression analysis, these methods attempt to simultaneously estimate unknown divergence times and smooth the rapidity of rate change along lineages. The latter is done by invoking some function that penalizes rates that change too quickly from branch to neighboring branch. The first, wholly nonparametric, method, nonparametric rate smoothing (NPRS: Sanderson 1997) consists of essentially nothing but the penalty function. Since the function includes unknown times, a suitably constructed optimality criterion based on this penalty can permit estimation of the divergence times. This method, however, tends to overfit the data leading to rapid fluctuations in rate in regions of a tree that have short branches.

Penalized likelihood (Sanderson 2002) is a method that combines likelihood and the nonparametric penalty function used in NPRS. It permits specification of the relative contribution of the rate smoothing and the data-fitting parts of the estimation procedure. The user can set any level of smoothing from severe, which essentially leads to a molecular clock to effectively unconstrained. Cross validation procedures are provided to provide data-driven methods for finding the optimal level of smoothing.

What's new in version 1.7

This version improves the robustness of the previous implementation of divergence time estimation routines and adds some entirely new features.

- The former is achieved by a new and stringent check on solutions via the `checkgradient` command. In conjunction with the use of multiple replicates from different initial conditions (`set num_time_guesses...`), this is the best line of defense against incorrect results.

- A new penalty function that penalizes differences in the logarithm of rates on neighboring branches has been added because of occasional pathological behavior of deep time estimates from shallow calibrations (`set penalty=log`).

- When multiple fossil calibrations or constraints are present, a fossil cross-validation procedure can now be invoked (as an option in the `divtime` command) to undertake model selection using penalized likelihood. This essentially uses the internal consistency among the fossil ages to help select the appropriate level of rate smoothing, and it can also provide a useful estimate of the age errors implied by that level of smoothing.

- A new likelihood ratio based relative rate test (`rrlike` command) has been added to permit inferences about local shifts in rates of evolution. It has all the advantages (or disadvantages) of other aspects of the program, such as taking advantage of multiple calibrations or age constraints.

These new features and options (as well as some minor changes) are highlighted by the tag, "*****New*****" in the text.

What was new in version 1.5

The most significant change is the addition of a new black box optimization routine to carry out truncated Newton optimization with simple bound constraints. The code, called TN, is due to Stephen Nash and is written in fortran. Further information is available at NetLib (http://www.netlib.org), where the code is distributed. In general, this code is much faster and better at finding correct solutions. However, it relies on gradients, which I have still only derived for Langley-Fitch (clock) and Penalized Likelihood methods (see Appendix). It is not yet implemented for NPRS.

A large number of bugs have been fixed, including some serious memory leaks. See the revision history in the `doc` directory of the distribution.

Installation

The program is currently available in several forms, including as a binary executable for Mac OS X and as source code that can be compiled under many UNIX systems, including Linux. I now do all r8s development entirely in the Mac OS X Unix environment but routinely check its portability to at least Linux systems. If you want to compile the source code for some reason and are working in OS X, note that their version of gcc does not come with the necessary FORTRAN compiler needed for a few important functions (you will have to obtain that elsewhere, e.g., from the FINK repository). However, the supplied executable does not require anything special to run.

If you are at all familiar with UNIX then you will have no trouble downloading and uncompressing the code and documentation. For those who are used to point-and-click programs on the Mac, this is not one of them! You have to run r8s in a UNIX shell window. In OS X there is a utility called "Terminal", which opens such a window and presents the standard UNIX command line interface. After downloading the file `r8sXX.dist.tar.Z` (where XX is the version number, such as 1.7) to your machine and learning enough UNIX to stick it in a directory somewhere, uncompress and untar it with the following commands

```
uncompress r8sXX.dist.tar.Z
tar xvf r8sXX.dist.tar
```

This sets up a subdirectory called `dist` which has four subdirectories, `bin`, `sample`, `doc`, `and` `src`. The first has the OS X executable, the second has several sample data files, the third contains various versions of this manual and the fourth contains the source code. You may want to move the executable file, `r8s`, into the path that your account uses to look for binary files. If you have a system administrator, or you have root access yourself, you can install the program in some place like `/usr/local/bin` so that everyone can have access to it. If you have not set the `PATH` variable for your account and are frustrated that you cannot get the program to execute, you may have to type `./r8s` rather than just `r8s`, assuming the program is in your current working directory.

To begin, you can check the version number of the program with by typing

```
r8s –v
```

If you've gotten to this point, you've probably learned nearly enough UNIX to use r8s effectively.

If you are working on another UNIX operating system (so the OS X executable is obviously useless!), or if you are on OS X but wish to compile the program from its source code, a `makefile` is included that works on Linux machines and OS X (assuming you have installed FORTRAN correctly). There are known incompatibilities with some other compiler's libraries or headers, most of which are easy to resolve. You should own a copy of Numerical Recipes in C (Press et al. 1992 or more recent versions) to compile and run from source.

Some definitions

A *cladogram* is a phylogenetic tree in which branch lengths have no meaning. A *phylogram* is a tree in which branch lengths correspond to numbers of substitutions along branches (or numbers of substitutions per site along that branch). A *chronogram* is a phylogenetic tree in which branch lengths correspond to time durations along branches. A *ratogram* is a tree in which branch lengths correspond to absolute **rates** of substitution along branches (apologies for the jargon).

Running the program

The program can be run in either batch or interactive modes. The default is interactive mode, usually executed from the operating system prompt as follows:

`r8s -f datafile <cr>` or `./r8s -f datafile <cr>`

This invokes the program, reads the data file, containing a tree or trees and possibly some `r8s` commands, and presents a prompt for further commands. Typing `q` or `quit` will exit the program.

Batch mode is invoked similarly but with the addition of the `-b` switch:

```
r8s -b -f datafile <cr>
```

This will execute the commands in the data file and return to the operating system prompt. Finally, note that the `-f` switch and `datafile` is optional. The program can be run interactively with just

```
r8s <cr>
```

It is possible to subsequently load a data file from within `r8s` with the `execute` command (see command reference).

The output from a run can be saved to a file using standard UNIX commands, such as

```
r8s -b -f datafile > outfile
```

Notice that if you neglect to include the `-b` switch in this situation, the program will sit there patiently waiting for keyboard input. You may think it has crashed or is taking unusually long to finish (oops).

Data file format

The program reads ASCII text files in the Nexus format (Maddison et al., 1997), a standard that is being used by increasing numbers of phylogenetic programs (despite the grumblings of Nixon et al. 2001). At a minimum the data file should have a `Trees` block in it that contains at least one `tree` command (tree description statement). The tree command contains a tree in parentheses notation, and optionally may include branch length information, which is necessary for the divergence time methods described below. Thus, the following is an example of a nearly minimal datafile named 'Bob' that `r8s` can handle:

```
#nexus
begin trees;
tree bob = (a:12,(b:10,c:2):4);
end;
```

Upon execution of

```
r8s -f Bob <cr>
```

the program will come back with a prompt allowing operations using this tree.

It is usually easiest to imbed these `r8s` commands into a special block in the data file itself and then run the program in batch mode. For example:

```
#nexus
begin trees;
tree bob = (a:12,(b:10,c:2):4);
end;

begin r8s;
blformat lengths=total nsites=100 ultrametric=no;
set smoothing=100;
divtime method=pl;
end;
```

followed by

```
r8s -f bob -b <cr>
```

The meaning of the commands inside the `r8s` block will be discussed below. All commands described in this manual can be executed within the `r8s` block.

Grammar for r8s block and r8s commands

All `r8s` commands must be found within a r8s block:

```
begin r8s;
      command1;
      command2;
      .
      .
      .
end;
```

All `r8s` commands follow a syntax similar to that used in PAUP* and several other Nexus-compliant programs, which will be described used the following conventions:

```
command option1=value1|value2|… option2=value1|…  (etc);
```

Thus any command might have several options, each of which has several alternative values. The vertical bars delimit alternative choices between values. Values may be strings, integers, real numbers, depending on the option. Default values are underlined. There are only a few exceptions to this general grammar.

Required information in the `r8s` block.

The program needs a few pieces of information about the trees and the format of the branch lengths stored in the `Trees` block. This is provided in the `blformat` command, which should be the first command in the `r8s` block. Its format is

```
blformat lengths=total|persite nsites=nnnn ultrametric=no|yes
            round=no|yes;
```

The `lengths` option indicates the units used for branch lengths in the tree descriptions. Depending on the program and analysis used to generate the branch lengths, the units of those lengths may vary. For parsimony analyses, the units are typically *total* numbers of substitutions along the branch, whereas for maximum likelihood analyses, they are typically expected numbers *per site*, which is therefore dependent on sequence length.

The `nsites` option indicates how many sites are in the sequences that were used to estimate the branch lengths in PAUP or some other program. Take care to insure this number is correct in cases in which sites have been excluded in the calculations of branch lengths (e.g., by excluding third positions in PAUP and *then* saving branch lengths). The program does not recognize exclusion sets or other PAUP commands that might have been used in getting the branch lengths.

The `ultrametric` option indicates whether the branch lengths were constructed using a model of molecular evolution that assumes a molecular clock (forcing the tree and branch lengths to be ultrametric). If `ultrametric` is set to `yes` then uncalibrated ages are immediately available for the trees. To calibrate them against absolute age, issue a `calibrate` command, which scales the times to the absolute age of *one specific node* in the tree. This linear stretching or shrinking of the tree is the only freedom allowed on an ultrametric tree. Multiple calibrations might force a violation of the information provided by the ultrametric branch lengths. Do not use the `fixage` command on such trees unless you wish to re-estimate ages completely.

The `round` option is included mainly for analyses of trees that are already ultrametric rather than estimation of divergence times using the `divtime` command. Ordinarily input branch lengths are converted to integer estimates of the number of substitutions on each branch (by multiplying by the number of sites and rounding). This is done because the likelihood calculations in `divtime` use numbers of substitutions as their data. However, when merely calibrating trees that are already ultrametric, rounding can generate branch lengths that are not truly ultrametric, because short branches will tend to be rounded inconsistently. When working with ultrametric, user-supplied chronograms, set `round=no`. Otherwise leave the default setting alone. If you set `round=yes` and estimate divergence times, the results are unpredictable, because calculations of gradients always assume data are integers.

If the program is run in interactive mode, this `blformat` command should be executed before any divergence time/rates analysis, although other commands will work without this.

Naming nodes/clades/taxa: the `mrca` command

Many commands refer to internal nodes in the tree, and it is often useful to assign names to these nodes. For example, it is possible to constrain the age of internal nodes to some minimum or maximum value, and the command to do this must be able to find the correct node. Every node is also the most recent common ancestor ("MRCA") of a clade. Throughout the manual an internal node may be referred to interchangeably as a "node name" , "clade name", or "taxon name."

The program uses two methods to associate names with internal nodes. One is available in the Nexus format for tree descriptions. The Nexus format permits internal nodes to be named by adding that name to the appropriate right parenthesis in a tree description. For example

```
tree betty=(a,(b,c)boop);
```

assigns the name `boop` to the node that is the most recent common ancestor of taxon b and c.

This method is obviously error prone and not very dynamic. Therefore `r8s` has a special command to permit naming of nodes. The command

```
MRCA boop a b;
```

will assign the name boop to the most recent common ancestor of `a` and `b`.

The name of a terminal taxon can also be changed by dropping one of the arguments:

```
MRCA boop a;
```

assigns the name `boop` to terminal taxon `a`.

Fixing and constraining times prior to estimating divergence times

One or more nodes can have its age fixed or constrained prior to estimating the divergence times of all other nodes using the `divtime` command. Time is measured backward from the most recent tip of the tree, which is assumed to have a time of 0 by default. In other words, times might best be thought of as "ages" with respect to the present. **Note**: the following commands are not appropriate if the input tree is already ultrametric—use `calibrate` instead.

Fixed and free nodes. Any node in the tree, terminal or internal, can have its age *fixed* or *free*. A node that is fixed has an age that is set to a specific value by the user. A node that is free has an age that must be estimated by the program. By default, all internal nodes in the tree are assumed to be free, and all terminal nodes are assumed to be fixed, but any node can have its status changed by using the `fixage` or `unfixage` command. The format for both of these commands is similar:

```
fixage taxon=angio age=150;
unfixage taxon=angio;
```

It is possible to fix *all* internal node ages with

```
fixage taxon=all;
```

or to unfix them all with

```
unfixage taxon=all;
```

but note that this applies (by default) only to internal nodes. To unfix the age of a terminal taxon, you must specify each such taxon separately:

```
unfixage taxon=terminal1;
unfixage taxon=terminal2;
```

Fixing the age of nodes generally makes it easier to estimate other ages or rates in the tree—that is it reduces computation time, because it reduces the number of parameters that must be estimated. **Roughly speaking, the divergence time algorithms require that at least one internal node in the tree be fixed**, or that several nodes be constrained as described below. If this is awkward, it is always possible to fix the root (or some other node) to have an arbitrary age of 1.0, and then all rates and times will be in units scaled by that age. Under certain conditions and with certain methods, the age of unfixed nodes cannot be estimated uniquely. See further discussion under "Uniqueness of solutions."

Ages persist on a tree until changed by some command. Sometimes it is desirable to retain an age for a particular node but to fix it so that it is no longer estimated in subsequent analyses. This can be accomplished using `fixage` but not specifying an age:

```
fixage taxon=sometaxon;
```

Constrained nodes. Any node can also be *constrained* by either a minimum age or a maximum age. This is very different from *fixing* a node's age to a specific value, and it only makes sense if the node is *free* to be estimated. Computationally it changes the optimization problem into one with bound constraints, which does not remove any parameters, and is much more difficult to solve. The syntax for these commands is given in the following examples:

```
constrain taxon=node1 min_age=100;
constrain taxon=node2 min_age=200 max_age=300;
```

and to remove constraints on a particular node:

```
constrain taxon=node1 min_age=none;
```

or to remove **all** such constraints across the entire tree (including terminals and internals):

```
constrain remove=all;
```

Once a constraint is imposed on a node, divergence time and rate algorithms act accordingly. No further commands are necessary.

The words `minage` and `maxage` can be used instead of `min_age`, `max_age`.

Estimation of rates and times

The `divtime` command, with numerous options, provides an interface to all the procedures for estimating divergence times and absolute rates of substitution. Given a tree or set of trees with branch lengths, and one or more times provided with the `fixage` or `constrain` commands, the `divtime` command provides calibrated divergence times and absolute rates in substitutions per site per unit time.

The simplified format of the `divtime` command is

```
divtime method=LF|NPRS|PL algorithm=POWELL|TN|QNEWT;
```

The first option describes the method used; the second describes the type of numerical algorithm used to accomplish the optimization.

Methods. The program currently implements four different methods for reconstructing divergence times and absolute rates of substitution. The first two are variations on a parametric molecular clock model; the other two are nonparametric or semiparametric methods.

LF method. The Langley-Fitch method (Langley and Fitch, 1973, 1974) uses maximum likelihood to reconstruct divergence times under the assumption of a molecular clock. It estimates one substitution rate across the entire tree and a set of calibrated divergence times for all unfixed nodes. The optimality criterion is the likelihood of the branch lengths. A chi-squared test of rate constancy is reported. If a gamma distribution of rates across sites is used, the appropriate modification of the expected Poisson distribution to a negative-binomial is used (see under Theory below).

LF "local molecular clock" method. This relaxes the strict assumption of a constant rate across the tree by permitting the user to specify up to k separate rate parameters. The user must then indicate for every branch in the tree which of the parameters are associated with it. This is useful for assigning different rates for different clades or for different collections of branches that may be characterized by some biological feature (such as a life history trait like generation time).

The first step in running this method is to "paint" the branches of the tree with the different parameters. Assume that there will be *k* rate parameters, where *k* > 1. The parameters are labeled as integers from 0,…, *k*-1. The default Langley-Fitch model assigns the rate parameter "0" to every branch of the tree, meaning there is a single rate of evolution. Suppose we wish to assign some clade "bob" a different rate, "1". Use the following command:

```
        localmodel taxon=bob stem=yes rateindex=1;
```

The option `stem=yes|no` is necessary if the taxon refers to a clade. It indicates whether the branch subtending the clade is also assigned rate parameter "1". It is important to use all integers from 0,…, *k*-1, if there are *k* rate parameters. The program does not check, and estimates will be unpredictable if they do not match. Using this command it is possible to specify quite complicated patterns of rate variation with relatively few commands.

The local molecular clock procedure is then executed by

```
        divtime method=lf nrates=k;
```

where *k* must be greater than 1. If *k* = 1, then a conventional LF run is executed. It is important to remember that with this approach it is quite easy to specify a model that will lead to degenerate solutions. **To detect this, multiple starting points should always be run** (see below under search strategies)**.** One robust case is when the two sister groups descended from the root node have different rates.

NPRS method. Nonparametric rate smoothing (Sanderson, 1997) relaxes the assumption of a molecular clock by using a least squares smoothing of local estimates of substitution rates. It estimates divergence times for all unfixed nodes. The optimality criterion is a sum of squared differences in local rate estimates compared from branch to neighboring branch.

PL method. Penalized likelihood (Sanderson, 2002) is a semiparametric approach somewhere between the previous two methods. It combines a parametric model having a different substitution rate on every branch with a nonparametric roughness penalty which costs the model more if rates change too quickly from branch to branch. The optimality criterion is the log likelihood minus the roughness penalty. The relative contribution of the two components is determined by a "smoothing parameter". If the smoothing parameter, `smoothing`, is large, the objective function will be dominated by the roughness penalty. If it is small, the roughness penalty will not contribute much. If `smoothing` is set to be large, then the model is reasonably clocklike; if it is small then much rate variation is permitted. Optimal values of smoothing can be determined by a cross-validation procedure described below.

Penalty functions. [***New***] Previous versions of PL in r8s used an **additive penalty** function that penalized squared differences in rates across neighboring branches in the tree. This seemed appropriate for many data sets in which calibrations were present deep in the tree and most nodes to be estimated were more recent. However, in some data sets in which calibrations are recent and users are attempting to extrapolate far backwards in time, this penalty function has the undesirable property that it can become small just by allowing the age of the root to become very old. This is because the estimated rates deep in the tree will be become small in magnitude as the branch durations get large, and thus the squared rate differences will be small as well. Version 1.7 introduces a second type of penalty function, the **log penalty**, which penalizes the squared difference in the *log* of the rates on neighboring branches. Because $\log x - \log y = \log(x/y)$, this is effectively penalizing fractional changes in rate rather than absolute changes. It is not always clear a priori for what conditions it is appropriate to use one penalty versus the other, but results can always be compared by looking at the best cross-validation scores for each. The log penalty function may also be more consistent with Bayesian relaxed clock methods that rely on modeling the evolution of rates by a distribution in which the log of descendant rates is distributed around the mean of the log of ancestral rates.

To select the penalty function use the `set` command:

`set penalty=log` or `set penalty=add`;

Algorithms. The program uses three different numerical algorithms for finding optima of the various objective functions. In theory, it should not matter which is used, but in practice, of course, it does. The details of how these work will not be described here. Eventually I hope to incorporate all algorithms in all the methods described above, but at the moment only Powell's algorithm is available for all methods. Table 1 indicates which algorithms are available for which methods and under what restrictions. **Unless you are doing NPRS or the local clock model, I recommend the TN algorithm**.

Powell. Multidimensional optimization of the objective function is done using a derivative-free version of Powell's method (Gill et al. 1981; Press et al. 1992). This algorithm is available for each of the three methods. However, it is not as fast, or as reliable as quasi-newton methods (QNEWT), which require explicit calculations of derivatives. For the LF algorithm, its performance is quick and effective at finding optima; for NPRS and PL, it sometimes gets trapped in local optima or converges to flat parts of the objective function surface and terminates prematurely. I recommend liberal restarts and multiple searches from several initial conditions (see under Search Strategies below).

TN. TN implements a truncated Newton method with bound constraints (code due to Stephen Nash). This algorithm is the best and fastest for use with LF or PL. It is not yet implemented for NPRS. It can handle age constraints and uses gradients for better convergence guarantees. It scales well and is useful for large problems.

Qnewt. Quasi-newton methods rely on explicit coding of the gradient of the objective function. This search algorithm is currently available only for LF and PL. The derivative in NPRS is tedious to calculate (stay tuned). In addition to converging faster than Powell's algorithm for this problem, the available of the gradient information permits an explicit check on convergence to a true optimum, where the gradient should be zero. However, one shortcoming of the current implemention of this algorithm is that it only works on unconstrained problems. If any nodes are constrained, QNEWT will almost surely fail, as the derivatives at the solution are no longer expected to be zero. **Powell or TN must be used for constrained problems** (Table 1).

Table 1. Algorithms implemented for various methods.

Method	Constraints	Non-extant terminals	POWELL	TN	QNEWT
LF	no	no	yes	yes	yes
	no	yes	yes	yes	no
	yes	no	yes	yes	no
	yes	yes	yes	yes	no
LF (local)	no	no	yes	no	no
	no	yes	yes	no	no
	yes	no	yes	no	no
	yes	yes	yes	no	no
NPRS	no	no	yes	no	no
	no	yes	yes	no	no
	yes	no	yes	no	no
	yes	yes	yes	no	no
PL	no	no	yes	yes	yes
	no	yes	yes	yes	no
	yes	no	yes*	yes	no
	yes	yes	yes*	yes	no

* but not cross-validation!

Cross-validation. The fidelity with which any of the three methods explain the branch length variation can be explored using a cross-validation option (Sanderson 2002). In brief, this procedure removes each terminal branch in turn, estimates the remaining parameters of the model without that branch, predicts the expected number of substitutions on the pruned branch, and reports the performance of this prediction as a cross-validation score. The cross-validation score comes in two flavors: a raw sum of squared deviations, and a normalized chi-square-like version in which the squared deviations are standardized by the observed. In a Poisson process the mean and variance are the same, so it seems reasonable to divide the squared deviations by an estimate of the variance, which in this case is the observed value.

The procedure is invoked by adding the option `CROSSV=yes` to the `divtime` command. For the PL method we are often interested in checking the cross-validation over a range of values of the `smoothing` parameter. To automate this process, the following options can be added to the `divtime` command, as in:

```
divtime method=pl crossv=yes cvstart=n1 cvinc=n2 cvnum=k;
```

where the range is given by $\{10^{(n1)}, 10^{(n1+n2)}, 10^{(n1+2n2)}, \ldots, 10^{(n1+(k-1)n2)}\}$. In other words the values range on a $\log_{10}$ scale upward from $10^{(n1)}$, with a total number of steps of *k*. The reason this is on a log scale is that many orders of magnitude separate qualitatively different behavior of the smoothing levels.

For most data sets that depart substantially from a molecular clock, some value of smoothing between 1 and 1000 will have a lowest (best) cross-validation score. Once this is found, the data set can be rerun with that level of smoothing selected. Suppose the optimal smoothing value is found to be *S** during the cross-validation analysis, then execute

```
set smoothing=S*;
divtime method=PL algorithm=qnewt;
```

to obtain optimal estimates of divergence times and rates.

Below I describe a new fossil-based version of this procedure, which uses some of the same syntax as described here.

<u>Estimating absolute rates alone</u>. In some problems, divergence times for all nodes of the tree might be given, and the problem is to estimate absolute substitution rates. This can be done by fixing all ages and running divtime (note that it is **not** an option with NPRS, which does not estimate rates as free parameters). For example:

```
fixage taxon=all;
set smoothing=500;
divtime method=pl algorithm=qnewt;
```

will estimate absolute substitution rates, assuming the times have been previously estimated or read in from a tree description statement using the `ultrametric` option.

[*****New*****] Checking the solution: the `checkGradient` command

The least exciting but most significant improvement in version 1.7 is the `checkGradient` command, which supercedes the older `showGradient` command. It provides additional checks on the correctness of solutions found by any of the `divtime` methods and algorithms. Moreover, it must be used carefully because it can falsely imply a correct solution is incorrect. **Nonetheless, I strongly recommend that any solution that you wish to take seriously be checked using this option.** The gradient check is turned on by

```
set checkGradient=yes;
```

To use it properly you must appreciate some of the basic calculus of optimization problems. I recommend the book by Gill et al. (1981) for further details. Things are relatively simple if there are no time constraints. If the objective function (say the penalized likelihood) has a peak, then if the algorithm finds the peak, the gradient (the set of derivatives of the objective function with respect to all the parameters) should be zero at that peak. All of the algorithms try to make this happen either explicitly (TN and QNEWT) or implicitly (POWELL), but it just does not always work. The `checkGradient` option tests the solution's gradient. Since no solution is exact it permits some tolerance in the gradient's deviation from 0 (see Gill et al. 1981), but usually if the gradient is zero it will "look like it" in the output.

When age constraints are present all bets are off. The objective function may still have a peak, but the parameter space is filled with "walls" that may prevent the algorithm from getting to that peak. In fact, the peak might occur in a place that violates some of those input constraints. So instead of finding the peak, the algorithm settles for, hopefully, the best solution it can find without bashing through any walls. If the best solution is nudged right up against a wall, the slope of the function at that point might well *not* be zero. As data sets are including more and more fossil constraints, it is almost always the case that divergence time solutions run up against these walls somewhere in the analysis.

An "active" constraint is one for which the estimated value of the parameter seems to run right up against the constraint (without violating it of course). The gradient at an active constraint can be very non-zero. However, its *sign* must still be correct. For example, if a constraint is a maximum age, and it is active at the solution, the gradient for that age parameter should be positive (the age is below the maximum and the objective function gets better—more positive—as the age continues to increase closer and closer to the bound). If the constraint is a minimum age, the gradient should be negative. The `checkGradient` command checks the sign for each active constraint.

Unfortunately, there is a small technical issue. How do we determine if the constraint is active or not? How close does the solution have to be to the "wall" before we allow it to have a non-zero gradient? In practice we set a small tolerance level with the command

```
	set activeEpsilon=real_number;
```

This number is combined with the age scale of the tree (the age of the root) to determine a distance from the constraint, within which a solution will be considered "close enough". This value is displayed in the `checkGradient` output. Two things can go wrong. First, if the tolerance is too small, `checkGradient` may not realize that the constraint is active, may conclude that a gradient's magnitude is too big, and decide that the gradient check has failed. If the tolerance is too large, then the constraint may be treated as active when it is not. The gradient might then be approximately zero, but because of numerical approximations have a random sign, leading to a failed check. Generally, I try to avoid this by ignoring any check of active parameters if the magnitude of the gradient is small (regardless of sign), but this is subject to numerical imprecision. Finally, one should avoid placing minimum and maximum age constraints that are very close together on the same node. Use `fixage` instead. If these constraints are too close to each other, the program will not know to which wall it is bumped up against, and it cannot predict which is the correct sign. This can be fixed if necessary by reducing the value of `activeEpsilon`.

If this sounds complicated, it is.

And, a final reminder that the gradient can be zero at a saddle point or on a plateau. The only check on this (without calculating the Hessian matrix...) is the unrelenting use of multiple initial starting conditions (`set num_time_guesses`...)

Should you use `checkGradient` during cross-validation? It is not computationally expensive, but I don't necessarily recommend that it be used without inspection of the results. It is conservative and given enough opportunities will probably cause a "failed" optimization warning to be registered if it is called enough times (as it is likely to be during cross-validation).

Showing ages and rates: the `showage` command

Each run of `divtime` produces a large quantity of output (which can be modified by the `set verbose` option), much of which is in a rather cryptic form. To display results in a cleaner form, use the `showage` command which shows a table giving the estimated age of each node in the tree, indicating whether the node is a tip, internal node, or name clade, whether the node is fixed or unfixed and what the age constraints, if any are set to. It also indicates (in square brackets) for every branch what rate index has been assigned to it using the `localmodel` command (the default is 0).

This command also indicates estimated rates of substitution on every branch of the tree. Branches are labeled by the node that they subtend. The estimates are different for the different methods. For example, under LF, there is only a single estimate of the rate of substitution, since the model assumes a clock—one rate across the tree. However, the table also includes a column for the "local rate" of substitution, which is just the observed number of substitutions (branch lengths) divided by the estimated time duration of the branch. For NPRS, this local rate is the only information displayed. For PL, each branch has a parameter estimate itself, which is displayed along with the local rate as described for the other two methods. For PL the parameter estimate should be interpreted as the best local rate estimate; the "local rate" is mainly provided for comparison to NPRS and LF methods. When examining a user supplied chronogram (assumed to be ultrametric), the program provides the single rate of substitution common across all branches.

[***New***] Use the command

```
showage shownamed=yes;
```

to display time information for only nodes in the tree that have been named by the MRCA command. This is useful when sorting through large tables of dates.

Showing trees and tree descriptions: the `describe` command

The describe command is useful for outputting simple text versions of trees either or before or after analyses have been run, and for generating tree description statements that can be imported into more graphically savvy packages, such as PAUP*. The general format is

```
describe plot=cladogram|phylogram|chronogram|ratogram|
       chrono_description|phylo_description|rato_description|node_info;
```

The first four option values generate an ASCII simple text version of a tree. A `cladogram` just indicates branching relationships; a `phylogram` has the lengths of branches proportional to the number of substitutions; a `chronogram` has the lengths of branches proportional to absolute time; a `ratogram` has them proportional to absolute rates.

The three description statements print out Nexus tree description versions of the tree. In the `phylo_description`, the branch lengths are the standard estimated numbers of substitutions (with units the same as the input tree); in `chrono_description` they are scaled such that a branch length corresponds to a temporal duration; in `rato_description`, the branch lengths printed correspond to estimated absolute rates in substitutions per site per unit time.

Option value of `node_info` prints a table of information about each node in the tree.

Other options include

```
            plotwidth=k;
```

where *k* is the width of the window into which the trees will be drawn, in number of characters.

The `set` command

The `set` command controls numerous details about the environment, output, optimization parameters, etc. See the command reference for further information

Scaling tree ages: the `calibrate` command

This command is useful for changing all the times on a tree by a constant factor. For example, if the input tree is scaled so that the distance from root to tip is 1.0, but you would like to define the age of the root at 500 MY and display the “calibrated” ages of the other nodes relative to the root. **The tree must already be ultrametric** (either supplied as such by the user, or reconstructed by the `divtime` command).

```
      calibrate taxon=nodename age=x;
```

where `taxon` refers to some node with a known age, *x*. All other ages will be scaled to this node.

When importing chronograms from other programs (using `blformat ultrametric=yes`), the ages will be set arbitrarily so that the root node has an age of 1.0 and the tips have ages of 0.0. Issue the `calibrate` command to scale these to some node's real absolute age.

Warning: do not mix `calibrate` and `fixage` commands. The former is only to be used on ultrametric trees, in situations where you are not estimating ages with the `divtime` command; the latter should only be used prior to issuing the `divtime` command.

Profiling across trees

Often it is useful to summarize information about a particular node in a tree across a collection of trees that are topologically identical, but which have different age or rate estimates. For example, a set of phylograms can be constructed by bootstrapping sequence data while keeping the tree fixed (Baldwin and Sanderson 1998; Korber et al. 2000; Sanderson and Doyle 2001). This generates a space of phylograms, which when run through `divtime`, yields a confidence set of node times for all nodes on the tree. The following command will collect various types of information for a node across a collection of trees.

```
      profile taxon=nodename parameter=length|age|rate;
```

Relative rates tests: the `rrlike` command [***New***]

New in version 1.7 is a likelihood ratio test of whether the two (or more) clades descended from a particular node are evolving at the same constant rate. It is invoked with

```
rrlike taxon=node_name;
```

The test uses the same modeling assumptions as the Langley-Fitch clock model. It tests the hypothesis that a model with one constant rate for the clade whose MRCA is `node_name` fits the data as well as a model in which each of the (two or more) subclades descended from that node have different constant rates.

The test is versatile in that it can take any fixed or constrained node ages into account while performing the test. However, note carefully that the tree outside of the clade defined by *`node_name`* is ignored entirely. This means that there has to be the usual minimum of one fixed node age *within* the clade of interest! If this is not the case, you will have to set the age of *`node_name`* to 1.0 or some other arbitrary value to proceed. If you ignore this advice, the program will try to set the age of the node to 100.0 units. This can cause other problems if this arbitrary age conflicts with constraints deeper in the tree.

Fossil-based model cross-validation [***New***]

In previous versions of r8s model-selection for penalized likelihood has relied entirely on a cross-validation scheme that involves pruning terminal branches and calculating prediction error. Given the increasing number of studies that include several fossil ages, another possibility is to use these dates to assist in model selection. At the same time, this might provide another criterion to evaluate the quality of the models implemented in r8s. This fossil-based model cross-validation is invoked in r8s by adding an option to the `divtime` command in conjunction with specifying cross-validation with the `crossv` option. There are two different flavors:

```
divtime crossv=yes fossilconstrained=yes ...;
divtime crossv=yes fossilfixed=yes ...;
```

In the first, “constrained” version the program does the following:

For each node, *k*, that has a constraint (fixed nodes are not affected)

(1) unconstrains node *k*

(2) does a full estimation of times and rates on the tree,

(3) if the estimated age of node *k would* have violated the original constraint on node *k*, calculate the magnitude of that violation (note that constraints are usually one sided, so that we consider a constraint “violated” if it is older than a maximum age or younger than a minimum age).

(4) keeps a running total of the magnitude of these violations across nodes examined

If the inference method is LF or NP then one round of cross-validation is invoked. If the method is PL, then the entire analysis is repeated over the usual range of smoothing values specified by the `divtime` options `cvstart`, `cvinc`, and `cvnum` described elsewhere for conventional cross-validation. The analysis reports two kinds of error, a fractional value per constrained node, and a raw value per constrained node in units of absolute time.

The second “fixed age” cross-validation analysis works slightly differently:

For each node, *k*, that is fixed (constrained nodes are not affected) the program

(1) “unfixes” the age of node *k*

(2) does a full estimation of times and rates on the tree,

(3) calculates the absolute value of the deviation of the newly estimated age of node k from its original “fixed” value.

(4) keep a running total of the magnitude of these deviations across nodes examined

The data set should have more than one fixed age for this approach because deletion of a solitary fixed age will preclude estimation of any times.

This method was used in a recent paper to examine divergence times in two data sets with mutliple fossil calibrations (Near and Sanderson 2004), but its utility in general is not yet known.

Substitution models and branch lengths

All the algorithms in `r8s` assume that estimates of branch lengths for a phylogeny are available, and that these lengths (and trees) are stored in a Nexus style tree command statement. Both the trees and the lengths will have been provided by another program, such as PAUP* or PAML. The original sequence data used to estimated these lengths are not used by `r8s` and are ignored even if provided in the data file. This is a deliberate design decision which offers significant advantages as well as certain limitations. The advantages are mainly with respect to computational speed and robustness of the optimization algorithms. Reduction of the character data to branch lengths permits much larger studies to be treated quickly. It also permits analytical calculation of gradients of objective functions for some of the algorithms in the program, which permits use of efficient numerical algorithms *and* checking of optimality conditions at putative solutions, which is harder with character-based approaches.

The risk, of course, is that some information is lost in the data reduction. However, the problem of estimating rates and times is considerably different than the problem of estimating tree topology. The latter is bound to be quite sensitive to the distribution of states among taxa in a single character, since this provides the raw evidential support for clades (even in a likelihood framework, where it is essentially weighted by estimates of rates of character evolution). The estimation of rates and divergence times, on the other hand, is simpler in the sense that one can at least imagine a transformation of branch length to time. For example, with a molecular clock, the expected branch lengths among branches with the same duration of time should be a constant.

Complex models of the substitution process have been described for nucleotide and amino acid sequence data, and it is possible to estimate the parameters of these models for sequence data on a given tree using maximum likelihood (or other estimation procedures) in programs such as PAUP* or PAML. Two important aspects of these models should be distinguished. One is the set of parameters associated with rates between alternative character states at given site in the sequence. For DNA sequences this is described by a 4X4 matrix of instantaneous substitution rates, which might have as many as 12 free parameters or as few as one (Jukes-Cantor). For amino acid data the matrix is 20 X 20, and for codons it might well by 64 X 64. Model acronyms abound for the various submodels possible based on these matrices. These models form the basis of a now standard formulation of molecular evolution as a Markov process (see Swofford et al. 1996, or Page and Holmes, 1998, for a review).

The other aspect of these models is the treatment of rate variation between branches in a phylogeny. By default, for example, PAUP* assumes the so called "unconstrained" model which permits each branch to have a unique parameter, a multiplier, which permits wide variation in absolute rates of substitution. To be more precise, this multiplier confounds rate and time, so that only the product is estimated on each branch. Under this model it is not possible to estimate rate and divergence times separately. Alternatively, PAUP* can enforce a molecular clock, which forces a constant rate parameter on each branch and then also estimates unknown node times. This is a severe assumption, of course, which is rarely satisfied by real data, but it is possible to relax the strict clock in various ways and still estimate divergence times.

The approach taken in `r8s` is to simplify the complexities of the standard Markov formulation, but increase the complexity of models of rate variation between branches. This is an approximation, but all models are. The main ingredients are a Poisson process approximation to more complex substitution models, and inclusion of rate variation between sites, which is modeled by a gamma distribution (Yang 1996). The shape parameter of the gamma distribution can be estimated in PAUP* (or PAML, etc.) at the same time as branch lengths are estimated. It is then invoked with the `set` command using:

```
	set rates=equal|gamma shape=α;
```

where the variable `shape` is the value of the normalized shape parameter, α, of a gamma distribution. The shape parameter controls the possible distributions of rates across sites. If $\alpha < 1.0$ then the distribution is monotonically decreasing with most sites having low rates and a few sites with high rates. If is $\alpha = 1.0$ the distribution is normal, and as $\alpha \rightarrow \infty$ the distribution gets narrower, approaching a spike, which corresponds to the same rate at all sites. The effect of rate variation between sites is much less important in the divergence time problem if branch lengths have already been correctly estimated by PAUP*, than if one were using the raw sequence data themselves. Unless the branch durations are extremely long, the alpha value extremely low or the rate very high, the negative binomial distribution that comes from compounding a Poisson with a gamma distribution is very close to the Poisson distribution. **Consequently, it is usually perfectly safe to `set rates=equal`. For this reason, only `algorithm=powell` takes into account any gamma distributed rate variation; the `qnewt` and `tn` algorithms treat rates as equal for all sites.**

Recommendations for obtaining trees with branch lengths.

PAUP* is a very versatile program for obtaining phylogenetic trees and estimating branch lengths, but a few of its options should be kept in mind. First, save trees with the `ALTNEXUS` format (without translation table), which saves trees with the taxon names as integral parts of the tree description. At the moment `r8s` does not use translation tables. Second, save trees as rooted rather than unrooted. By default, PAUP* saves trees as unrooted, whereas most analyses in `r8s` assume the tree is rooted. The reason this is important is that conventionally unrooted trees are stored with a basal trichotomy, which reflects the ambiguity associated with not having a more distant outgroup (i.e., it is impossible for a phylogenetic analysis to resolve the root node of a tree without more distant outgroups). Upon reading an unrooted tree with a basal trichotomy, `r8s` will assume the trichotomy is "hard" (i.e., an instant three-way speciation event, as it does with all polytomies), and act accordingly.

This requires the user to root the tree perhaps arbitrarily, or at best on the basis of further outgroup information. However, in converting the basal trichotomy to a dichotomy, PAUP* creates a new root node, and it or you must decide how to dissect the former single branch length into two sister branches. Left to its own, PAUP decides this arbitrarily, giving one branch all the length and the other branch zero, which is probably the worst possible assumption as far as the divergence time methods in `r8s` are concerned. The solution I recommend is to make sure every phylogenetic analysis that will eventually be used in `r8s` analyses start out initially with an *extra* outgroup taxon that is distantly related to all the remaining taxa. The place where this outgroup attaches to the rest of the tree will become the root node of the real tree used in r8s, once it is discarded. It will have served the purpose in PAUP of providing estimates of the branch lengths of the branches descended from the eventual root node.

The extra outgroup can be deleted manually from the tree description commands in the PAUP tree file, or it can be removed using the `prune` command in r8s.

Uniqueness of solutions

Theory (e.g. Cutler 2000) and experimentation with the methods described above shows that it is not always possible to estimate divergence times (and absolute rates) uniquely for a given tree, branch lengths, and set of fixed or constrained times. This can happen when the optimum of the objective function is a plateau, meaning that numerous values of the parameters are all equally optimal. Such "weak optima" should be distinguished from cases in which there are multiple "strong" optima, meaning there is some number of multiple solutions, but each one is well-defined, corresponding to a single unique combination of parameter values. Multiple strong optima can often be found by sufficient (if tedious) rerunning of searches from different initial conditions. Necessary and sufficient conditions for the existence of strong optima are not well understood in this problem, but the following variables are important.

- Is any internal node fixed, and if so, is it the root or not?
- Are the terminal nodes all extant, or do some of them have fixed times older than the present (so-called "serial" samples, such as are available for some viral lineages)
- Are minimum and/or maximum age constraints present?
- How much does the reconstruction method permit rates to vary?

The following conclusions appear to hold, but I would not be shocked to learn of counterexamples:

- Under the LF method a sufficient condition for all node times to be unique is that at least one internal node be fixed.
- Under the LF *local* clock method, the previous condition is not generally sufficient, although for *some* specific models it is.
- Under the LF method a sufficient condition for all node times to be unique is that at least one terminal node must be fixed at an age older than the present.
- Under the NP method condition the previous condition is not sufficient.
- Minimum age constraints for internal nodes, by themselves, are not sufficient to allow unique estimates under any method.
- Under LF, maximum age constraints *sometimes* are sufficient to permit the estimation of ages uniquely *when combined with minimum age constraints*. Generally this works when the data want to stretch the tree deeper than the maximum age allows and shallower than the minimum age allows. If the reverse is true, a range of solutions will exist

Bootstrap confidence intervals on parameters

As of version 1.7, there are no longer built in commands to provide immediate information about confidence intervals on parameters. They were unstable and difficult to interpret. Instead I recommend a lengthy bootstrap procedure as follows.

The idea is to generate chronograms from bootstrapped data. This means generating a series of phylograms based on a single tree (meaning the same topology but different sets of branch lengths), reading these into r8s, estimating ages for all trees, and then using the `profile` command to summarize age distributions for a particular node. The central 95% of the age distribution then provides a confidence interval (see Baldwin and Sanderson 1998; Sanderson and Doyle 2001). This can be a fairly tedious procedure unless you are willing to write scripts to automate some of the work, or use some that have been developed by other workers in the systematics community (e.g., Torsten Eriksson's package available at http://www.bergianska.se/ index_kontaktaoss_torsten.html).

Several steps are involved:

a) From the original data matrix, generate *N* bootstrap data matrices using PHYLIP (currently PAUP does not report the actual matrices).
b) Convert these to Nexus formatted files (here is where some programming ability helps)
c) Construct a "target" tree—the single tree topology upon which dates will be estimated. This can be built any way you want, but presumably will be based on the same data as in (a). Make a Nexus tree file of this target tree.
d) Add a PAUP block to each of the *N* bootstrapped Nexus files. The PAUP block should contain PAUP commands to (i) read the target tree file (after the data matrix has been stored, of course), (ii) set the options for the appropriate substitution model (e.g., using likelihood), and (iii) save the target tree **with branch lengths** (i.e., as a phylogram) to a common file for all *N* replicates. Note that r8s does not read Nexus translation tables, so you should save trees with the `ALTNEXUS` format in PAUP.

e) Finally, the tree file with *N* phylograms is read into `r8s` and divergence times estimated. Use the `profile` command to summarize the distribution of ages for a particular node of interest.

Search strategies, optimization methods, and other algorithmic issues

Many things can go wrong with the numerical algorithms used to find the optimum of the various objective functions in the `divtime` command:

1) the algorithms may fail to terminate in the required maximum number of iterations (variable `maxiter`), giving an error message
2) they may terminate some place that is not an optimum even though it may have a gradient of zero or a flat objective function (such as a saddle point, or local plateau, a so-called "weak optimum")
3) they may terminate at a local optimum, but not a global optimum.

Problem 1 can sometimes (though surprisingly rarely) be solved just by increasing the value of `maxiter` parameter (only when using `algorithm=powell`). Values above a few thousand, however, seem adequate for any solution that is going to be found. Above that, the routines will probably bomb for some other reason. This value is not adjustable for the TN algorithm, as it rarely helps.

Problem 2 is more pathological and harder to detect. Assuming that the `checkGradient` option reveals that the gradient for the inactive constraints is actually zero, it is still possible that the solution is on a saddle or a plateau, rather than a true peak. If so, it may be possible to discover this by perturbing the solution and seeing if it returns to the same point. If it is a local optimum, it should. Perturbations of the solutions in random directions are implemented by setting `num_restarts` to something greater than zero, which will rerun the optimization that number of times. The variable `perturb_factor` controls the fractional change in the parameter values that are invoked in the random perturbation. If you see that this tends to frequently find better solutions, you should suspect that the algorithm is having trouble finding a real optimum. Multiple starts of the optimization from different initial conditions may also give provide evidence of the reality of an optimal value. Multiple starts are invoked by setting `num_time_guesses=`*`n`*, where *n*>1. This selects a random set of initial divergence times to begin the optimization, and repeats the entire process *n* times. The results are ideally the same for each replicate. If not there are multiple optima. The command `set seed=`*`k`*, where *`k`* is some integer, should be invoked to initialize the random number generator.

Problem 3 is a matter of finding the all the real optima if more than one exists. Because these algorithms are all hill-climbers, they will at best find only one local optimum in any one search from an initial starting condition. To find multiple optima, it is necessary to start from several different random starting values. As described above, this is implemented by the `set num_time_guesses=`*`n`*, where *n* is greater than 1.

General advice and troubleshooting

Since the last major revision in 2002, I have had the opportunity to look at many data sets on which r8s performs well and many on which it does not. Frankly, I'm always surprised when it works on any data set that has more than about 35 taxa. The multivariate optimization problems handled by r8s are more difficult than some other phylogenetic problems in two respects: first, optimization is subject to constraints, and second, the problem's objective function itself can become "ill-conditioned" (small changes in the data can lead to large changes in the estimated solution)in several ways—for example, by specifying smoothing levels to be too low or too high in penalized likelihood.

In general, I recommend the `algorithm=TN` option for all LF or PL divergence time searches, including anything involving constraints or cross validation or both. It certainly does not hurt to check with `algorithm=powell`. However, this is much slower (especially for cross-validation work). Also be aware that results are more a bit more variable with this algorithm, and it may be necessary to increase the stringency of its search by making the stopping tolerance criterion smaller (e.g., `set ftol=1e-9` which is lower than the default 1e-6.).

Problems can crop up in a variety of contexts. Some involving numerical/algorithmic issues were discussed in the previous section. Others include:

Problems stemming from the data set and inferred phylogram, such as when:

- Sequence variation is very low and branch lengths are very short or zero. The `divtime` command requires that internal branches that are 0-length be collapsed with the `collapse` command. Terminal 0-length branches are also a problem, which I have tried to solve with various hacks, such as putting a lower bound on rates and durations so as to prevent terminal 0-length branches being inferred pathologically to have a rate of zero, which would cause no end of problems in the numerics. See the `set minRateFactor` and `set minDurFactor` options. Some data sets I have seen have large numbers of identical terminal taxa on very short or 0-length branches (especially some population level data or close phylogeographic studies). I suggest reducing all these clades to one exemplar.
- Rate variation is extremely high and fairly chaotically distributed across the tree. This is apparent by observing extreme heterogeneity of branch lengths on the phylogram. NPRS and PL perform best if there is autocorrelation in rates across the tree. The user will probably obtain estimates by assuming a clock that are as accurate as those obtained with NPRS or PL.

Problems in calibration:

- You must have either (i) at least one fixed age, or (ii) *possibly* a combination of one or more each of minimum and maximum age constraints, to have any hope of finding unique solutions. The second case is much more problematic, but even the first can fail if the clock is relaxed too much with low smoothing values.

- The most robust data sets have fixed ages or constraints near the root of the tree
- A number of problems and perceived problems have come up in data sets where the fixed age(s) or constraints are relatively shallowly placed in the tree and the problem is to estimate deeper nodes. I mentioned this case as potentially problematic in the one FAQ in vers. 1.5's manual. The issue is that the additive penalty function can allow nearly equally optimal solutions over a range of possibly old ages deep in the tree as smoothing is relaxed. Some people have referred to this as a "bias" toward older ages. I prefer to see it as a "feature" (!), but have added the log penalty function in version 1.7 for users who want to explore a different penalty function that should behave differently in this situation. Comparison of minimum CV scores between methods can help the user decide if one is better. I am reluctant to conclude that this reflects a bias in the additive penalty until simulation studies are done, but I do think that the log penalty is more intuitively reasonable for deep extrapolation problems. Comparisons with other methods are not compelling, since, for example, some require the imposition of a prior distribution on the root time. In r8s there is no prior penalty assigned to any node's time as long as it does not violate a constraint. Imposition of priors can certainly help with difficult divergence time problems, but it obviously comes at the cost of making additional assumptions.

Problems occurring during cross-validation:

- It may register errors during some of the taxon prunings, causing the entire score for that smoothing value to be invalidated. One workaround is to try multiple initial starts with `set num_time_guesses`. The problem may occur only for particular initial conditions. However, it is often a signal that the smoothing value is too low or too high, causing the problem to be ill-conditioned. You may have to be satisfied with a narrower range of smoothing parameters. You can always estimate the value for very high smoothing values by just running an LF analysis. It is worth playing around with the range of values, because the optimal range depends on the data and whether and additive or log penalty is used.
- Sometimes (though not often), the CV score does not exhibit the expected dip at intermediate values of smoothing. Trivially, this can occur if you have not examined a broad enough range of smoothing parameters (the range of 10^0 to 10^4 is often a good place to start). However, some data monotonically decreases its CV score as smoothing gets large. This implies that a model of a molecular clock does as well or better at predictive cross-validation as a PL model. Use it instead to estimate ages. This can occur even if the data show considerable rate heterogeneity, but the heterogeneity has no pattern to it across the tree.
- Even less often there appears to be no smooth pattern to the CV score as a function of smoothing. Usually this occurs in a case like that just described and the percentage fluctuation in the scores is usually small (<5%) of the CV scores themselves. This again should lead to the conclusion that a clock is just as good a model as a relaxed clock. If there are data sets that have wild fluctuations that are not just due to numerical problems in the optimizations, I would like to hear about it. Note that in

both of these last two cases, estimated divergence times *may change* with different smoothing values under PL. This does not mean that the PL "model" should be accepted over the LF model.

Finally, based on examining many data sets that are reported to have problems, I conclude that the best test of any result is numerous repeated runs from different initial conditions. To this I would add judicious use of the new `checkGradient` feature.

Bug reports (thanks, apologies, etc.)

Thanks to all of you who have sent me bug reports and data sets in the past. I apologize for my very uneven responses to these. Many of them have been very useful and some have led to major changes in the code. Unfortunately, in a few cases, the only answer was, "it doesn't work with your data".

I will continue to look at bug reports and try to fix them. Please send a completely self-contained nexus data file that exhibits the misbehavior. Send them to me at

mjsanderson@ucdavis.edu

with a brief description of the problem. Thankfully I am getting fewer and fewer requests for basic help with UNIX as people become more familiar with this environment, so in the last year or so most bug reports have focused on real r8s specific issues.

"Extra commands and features"

The last three commands listed in the command reference are not discussed in detail in this manual because they are not directly related to divergence times or rates. The `mrp` command constructs an MRP matrix for later supertree construction, based on all the input trees in the input nexus file. The `simulate` command provides a set of functions to generate random trees according to various models. The `bd` command plots the log species richness of the clade as a function of time and reports some statistics. The `mrp` command is quite reliable and I have found it handles large inputs quite well (many trees, many taxa). The `simulate` command has been used by several people in addition to myself, but has not been tested extensively. The `bd` command is very much under construction at this point, and should not be trusted. Other, even less well documented, commands are buried in the junkyard of the source code, and you can feel free to look through the source file `ReadNexusFile2.c` to excavate these.

Command reference

Command	Option	Value	Description	Default value
blformat	`lengths =`	`persite | total`	(input trees have branch lengths in units of numbers of substitutions per site | lengths have units of total numbers of substitutions	`total?`
	`nsites =`	`<integer>`	(number of sites in sequences that branch lengths on input trees were calculated from)	`1`
	`ultrametric =`	`yes | no`	(input trees have ultrametric branch lengths)	`no`
	`round =`	`yes | no`	(round estimated branch lengths to nearest integer)	`yes`
calibrate	`taxon =`	`<nodename>`	(scales the ages of an ultrametric tree based on the node nodename…)	
	`age =`	`<real>`	(…which has this age)	
cleartrees			(removes all trees from memory)	
collapse			(removes all zero-length branches from all trees and converts them to hard polytomies)	
constrain	`taxon =`	`<nodename>`	(enforces a time constraint on node nodename, which may be either…)	
	`minage (min_age) =`	`<real> | none`	(…a minimum age | or "none" removes the constraint)	
	`maxage (max_age) =`	`<real> | none`	(…maximum age, | or removes the constraint)	
	`remove =`	`all`	(removes all constraints on the tree)	
describe	`plot =`	`cladogram | phylogram | ratogram | chrono_description | phylo_description|`		

	plotwidth =	`rato_description` | `node_info` | `<integer>`		
divtime	method =	LF | NPRS | PL	(estimate divergence times and rates using Langley-Fitch | NPRS | penalized likelihood)	
	algorithm =	POWELL | QNEWT TN	(using this numerical algorithm)	
	nrates =	`<integer>`	(number of rate categories across tree for use with local molecular clock)	
	confidence =	YES | NO	(estimate confidence intervals for a node)	
	taxon =	`<taxonname>`	(node for confidence interval)	
	cutoff =	`<real>`	(decrease in log likelihood used to delimit interval)	
	crossv =	yes | no	(turn on cross-validation)	
	cvstart =	`<real>`	(log10 of start value of smoothing)	
	cvinc =	`<real>`	(smoothing increment on log10 scale)	
	cvnum =	`<integer>`	(number of different smoothing values to try)	
	fossilconstrained=	yes | no	(fossil cross validation removing constrained nodes in turn)	no
	fossilfixed=	yes | no	(fossil cross validation removing fixed nodes in turn)	no
	tree =	`<integer>`	(use this tree number)	
execute	`<filename>`		(execute this filename from r8s prompt)	
fixage	taxon =	`<taxonname>`	(fix the age of this node at the following value and no longer estimate it…)	
	age =	`<real>`	(…age to fix it)	
localmodel	taxon =	`<taxonname>`	(assign all the branches descended from this node an index for a local molecular clock model)	
	rateindex =	`<integer>`	(an integer in the range [0,…, *k*-1] where *k* is the number of different allowed local rates)	0
	stem =	yes | no	(if "yes" then include	no

			the branch subtending the node in this rate index category)	
mrca		`cladename terminal1 terminal2 [etc.]`	(assign the name cladename to the most recent common ancestor of terminal1, terminal2, etc.—NB! the syntax here does not include an equals sign. Also, if there is only one terminal specified, the terminal is renamed as cladename)	
profile	`taxon=`	`<nodename>`	(calculates summary statistics across all trees in the profile for this node)	
	`parameter=`	`age|length|rate`	(the statistics are for this selected parameter only)	
prune	`taxon=`	`<taxonname>`	(delete this taxon from the tree)	
quit			(self evident)	
reroot	`taxon=`	`<taxonname>`	(reroot the tree making everything below this node a monophyletic sister group of this taxon)	
rrlike	`taxon=`	`<taxonname>`	(perform a likelihood ratio test for molecular clock in the subclades descended from node taxonname)	
set	`rates =`	`equal | gamma`	(all sites have the same rates | sites are distributed as a gamma distribution)	`equal`
	`shape =`	`<real>`	(shape parameter of the gamma distribution)	`1.0`
	`smoothing=`	`<real>`	(smoothing factor in penalized likelihood)	`1.0`
	`npexp =`	`<real>`	(exponent in NPRS penalty)	`2.0`
	`verbose =`	`0`	(suppress almost all output)	`1`
		`>1`	(display more output)	
	`seed =`	`<integer>`	(random number seed)	`1`
	`num_time_guesses=`	`<integer>`	(number of initial starts	`1`

		in `divtime`, using different random combinations of divergence times)	
`num_restarts =`	`<integer>`	(number of perturbed restarts after initial solution is found)	`1`
`perturb_factor=`	`<real>`	(fractional perturbation of parameters during restart)	`0.05`
`ftol =`	`<real>`	(fractional tolerance in stopping criterion for objective function)	
`maxiter =`	`<integer>`	(maximum allowed number of iterations of Powell or Qnewt routines)	`500`
`barriertol =`	`<real>`	(fractional tolerance in stopping criterion between barrier replicates in constrained optimization)	
`maxbarrieriter =`	`<integer>`	(maximum allowed number of iterations of constrained optimization barrier method)	
`barriermultiplier=`	`<real>`	(fractional decrease in barrier function on each barrier iteration during constrained optimization)	
`initbarrierfactor=`	`<real>`		
`linminoffset=`	`<real>`		
`contractfactor=`	`<real>`		
`showconvergence =`	`yes | no`	(display objective function during approach to optimum)	`no`
`checkgradient =`	`yes | no`	(check if gradient is zero at solution, or has correct signs in constrained problems)	`no`
`showgradient =`	`yes | no`	(display the analytically calculated gradient and norm at the solution, if it is available)	`no`
`trace =`	`yes | no`	(show calculations at every iteration of the objective function—for debugging purposes)	
`minRateFactor =`	`<real>`	(imposed lower bound on the rates in PL as a fraction of the approximate mean rate)	`0.05`
`minDurFactor =`	`<real>`	(imposed lower bound on the durations of 0-length terminal branches	`0.001`

			as a fraction of root's age)	
	`penalty =`	`add|log`	(penalty function for PL method)	`add`
	`activeEpsilon =`	`<real>`	(let ε = activeEpsilon × age of root node; then if a solution is within ε of a constraint, it is considered "active": see `checkGradient`)	`0.001`
`showage`			(display a table with estimated ages and rates of substitution)	
	`shownamed =`	`yes|no`	(only display age information for nodes that have been named using MRCA command)	`no`
`unfixage`	`taxon =`	`all | <taxonname>`	(all node <taxonname> to have its age estimated | all all internal nodes to be estimated)	
`"Extra" commands and features`				
`mrp`	`method=`	`baum|purvis`	(construct an MRP matrix representation for the collection of trees using either method)	`baum`
	`weights=`	`yes|no`	(write a Nexus style 'wts' command to the output file based on the input tree descriptions' specified branch lengths—these should have been save so as to correspond to bootstrap support values)	`no`
`simulate`	`diversemodel=`	`yule | yule_c | bdforward`	(simulate a random tree according to one of three stochastic processes—see text)	
	`T=`	`<real>`	(age of root node, measured backward from the present, which is age 0)	`1.0`
	`speciation=`	`<real>`	(speciation rate)	`1.0`
	`extinction=`	`<real>`	(extinction rate)	`0.0`
	`ntaxa=`	`<integer>`	(conditioned on this	

			final number of taxa)	
	nreps=	<integer>	(number of trees generated)	
	seed=	<integer>	(random number seed)	
	charevol=	yes|no	(generate branch lengths, instead of times only)	no
	ratemodel=	normal|autocorr	(rates are either chosen randomly from a normal distribution with mean of 'startrate' and standard deviation of 'changerate'; or in an autocorrelated fashion starting with 'startrate' and jumping by an amount 'changerate' with probability 'ratetransition'	
	startrate=	<real>	(character evolution rate at root of tree)	
	changerate=	<real>	(either standard deviation of rate or amount that rate changes per jump in rate, depending on model)	
	ratetransition	<real>	(probability that rate changes at a node under AUTOCORR model)	
	minrate=	<real>	(lower bound)	
	maxrate=	<real>	(upper bound)	
	infinite=	yes|no	(default of 'no' means that the branch length is determined as a Poisson deviate based on the rate and time; if 'yes', then branch length is set to the expected value, which is just rate*time)	no
bd	divplot=	yes|no	(takes a chronogram and plots species diversity through time, and estimates some parameters)	

References

Ayala, J. A., A. Rzhetsky, and F. J. Ayala. 1998. Origin of the metazoan phyla: molecular clocks confirm paleontological estimates. Proc. Natl. Acad. Sci. USA **95**:606-611.

Cutler, D. J. 2000. Estimating divergence times in the presence of an overdispersed molecular clock. Mol. Biol. Evol. **17**:1647-1660.

Gill, P. E., W. Murray, and M. H. Wright. 1981. Practical optimization. Academic Press, New York.

Hasegawa, M., H. Kishino, and T. Yano. 1989. Estimation of branching dates among primates by molecular clocks of nuclear DNA which slowed down in Hominoidea. J. Human Evol. **18**:461-476.

Huelsenbeck, J. P., B. Larget, and D. Swofford. 2000. A compound Poisson process for relaxing the molecular clock. Genetics **154**:1879-1892.

Kishino, H., J. L. Thorne, and W. J. Bruno. 2001. Performance of a divergence time estimation methods under a probabilistic model of rate evolution. Mol. Biol. Evol. **18**:352-361.

Korber, B., M. Muldoon, J. Theiler, F. Gao, R. Gupta, A. Lapedes, B. H. Hahn, S. Wolinsky, and T. Bhattacharya. 2000. Timing the ancestor of the HIV-1 pandemic strains. Science **288**:1789-1796.

Kumar, S., and S. B. Hedges. 1998. A molecular timescale for vertebrate evolution. Nature **392**:917-920.

Langley, C. H., and W. Fitch. 1974. An estimation of the constancy of the rate of molecular evolution. J. Mol. Evol. **3**:161-177.

Maddison, D. R., D. L. Swofford, and W. P. maddison 1997. NEXUS: An extensible file format for systematic information. Syst. Biol. 46:590-621.

Near, T. J., and M. J. Sanderson. 2004. Assessing the quality of molecular divergence time estimates by fossil calibrations and fossil-based model selection. Phil. Trans. R. Soc. London B, in press.

Nixon, Kevin C.; Carpenter, James M.; Borgardt, Sandra J. 2001. Beyond NEXUS: Universal cladistic data objects. Cladistics. 17:S53-S59.

Page, R. D. M., and E. C. Holmes. 1998. Molecular evolution: A phylogenetic approach. Blackwell Scientific, New York.

Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. 1992. Numerical recipes in C. Cambridge University Press, New York. 2nd ed.

Rambaut, A., and L. Bromham. 1998. Estimating divergence data from molecular sequences. Mol. Biol. Evol. **15**:442-448.

Sanderson, M. J. 1997. A nonparametric approach to estimating divergence times in the absence of rate constancy. Mol. Biol. Evol. **14**:1218-1231.

Sanderson, M. J. 2002. Estimating absolute rates of molecular evolution and divergence times: a penalized likelihood approach. Mol. Biol. Evol. 19:101-109.

Swofford, D. S. 1999. PAUP * 4.0. Phylogenetic analysis using parsimony (*and other methods), v. 4b2. Sinauer Associates, Sunderland, MA.

Takezaki, N., A. Rzhetsky, and M. Nei. 1995. Phylogenetic test of the molecular clock and linearized trees. Mol. Biol. Evol. **12**:823-833.

Thorne, J. L., H. Kishino, and I. S. Painter. 1998. Estimating the rate of evolution of the rate of evolution. Mol. Biol. Evol. **15**:1647-1657.

Wray, G. A., J. S. Levinton, and L. H. Shapiro. 1996. Molecular evidence for deep precambrian divergences among metazoan phyla. Science **274**:568-573.

Yang, Z. 1996. Among-site rate variation and its impact on phylogenetic analyses. Trends Ecol. Evol. **11**:367-372.

Mathematical Appendix

This appendix describes the objective functions used by r8s to estimate divergence times and rates. It also provides analytical expressions for the gradients of these functions, which are used by some of the optimization algorithms to find solutions.

Label the n+1 nodes of a rooted, possibly non-binary tree, with $\{0,...,n\}$, where 0 represents the root node. Label each branch with the label of the node that it subtends. Let t_i be the time of node i, measure backward from the present; x_i be the observed number of substitutions on branch i (the branch "length"); λ_i be the rate of evolution on this branch. Let k' be the ancestor of node k, t' be the time of k' and $T_i = t_i - t_i$' (the branch "duration"). Let $D(k)$ be the set of nodes descended from node k.

Note that this appendix ignores the fact that some nodes in a tree may be fixed or constrained by the user—in fact, at least one node has to be! The changes to the equations below are obvious.

Langley-Fitch (LF)

This is the simple implementation of a constant rate "molecular clock" model. The objective function is the likelihood , which is calculated based on the assumption that the "observed" number of substitutions on a branch follows a Poisson distribution.

$$L(\lambda, t_0, t_1, ..., t_n \mid x_1, ..., x_n) = \prod_{i=1}^{n} (\lambda T_i)^{x_i} \exp(-\lambda T_i) / x_i!$$

$$\log L = \sum_{i=1}^{n} \left\{ -\lambda T_i + x_i \log(\lambda T_i) - \log(x_i!) \right\}$$

The gradient includes derivatives with respect to the single rate and to all unknown node times. The first is

$$\begin{aligned} \frac{\partial \log L}{\partial \lambda} &= \frac{1}{\lambda} \sum_{k=1}^{n} x_k - \sum_{k=1}^{n} T_k \\ &= \frac{B}{\lambda} - T \end{aligned}$$

where B is the sum of the branch lengths and T is the sum of the branch durations. The gradient with respect to the times is more complicated and terms have to be included depending on whether the node is a root, tip or some other internal node.

$$\frac{\partial \log L}{\partial t_k} = \frac{-x_k}{t_{k'} - t_k} + \lambda \text{ \{this term included whenever } k \text{ not the root\}}$$
$$+ \sum_{j \in D(k)} \left\{ \frac{x_j}{t_k - t_j} - \lambda \right\} \text{\{this term included whenever } k \text{ not a tip\}}$$

The reason this gets complicated is that in general there are terms from neighboring branches that affect the derivative with respect to any one node time (because the likelihood depends explicitly on *durations*, which involve a node and its immediate ancestor or descendant node). Sometimes these neighboring branches are absent if the node is a root or tip. These complications holds for all the objective functions.

Penalized likelihood (PL)

Penalized likelihood combines a likelihood term, which is a generalization of the LF term above, with a penalty term that imposes a cost whenever the difference in rates between neighboring branches is large.

$$\Phi = \log L - \gamma \pi(\lambda_1, ..., \lambda_n)$$

where L is the likelihood, π is a penalty function of the rates and γ is a positive number called the smoothing parameter that controls the relative importance of the penalty and the likelihood.

The model for the likelihood term is a Poisson model, as with the LF approach, but here every branch is allowed to have its own rate:

$$L(\lambda_1, ..., \lambda_n; t_0, t_1, ..., t_n \mid x_1, ..., x_n) = \prod_{i=1}^{n} (\lambda_i T_i)^{x_i} \exp(-\lambda_i T_i) / x_i!$$
$$\log L = \sum_{i=1}^{n} \left\{ -\lambda_i T_i + x_i \log(\lambda_i T_i) - \log(x_i!) \right\}$$

The penalty function is an important control on the degeneracy of the problem. If it were absent there would be more parameters to estimate than there are data. When present it effectively reduces the number of free parameters in the optimization. The penalty function I adopt is basically a least squares penalty. Ignoring for a moment the branches immediately descended from the root, the **additive** penalty function looks like

$$\pi(\lambda_1, ..., \lambda_n) = \sum_{k=1}^{n} \left(\lambda_k - \lambda_{k'} \right)^2$$

but since we cannot compare the basal branches (the ones descended from the root) to their ancestral branches, we penalize the *variance* of these basal rates. Thus, the total penalty is:

$$\pi(\lambda_1,\ldots,\lambda_n) = \mathrm{Var}\big(\lambda_r : r \in D(0)\big) + \sum_{k \notin D(0)} \big(\lambda_k - \lambda_{k'}\big)^2$$

where recall that $D(0)$ are the immediate descendants of the root node.

The **logarithmic** penalty function just replaces rates with log rates:

$$\pi(\lambda_1,\ldots,\lambda_n) = \mathrm{Var}\big(\log\lambda_r : r \in D(0)\big) + \sum_{k \notin D(0)} \big(\log\lambda_k - \log\lambda_{k'}\big)^2$$

Since $\log x - \log y = \log (x/y)$, this penalty function penalizes *fractional* changes in rate rather than absolute changes in rate.

The gradient with respect to node times is quite similar to that for LF above, except that the overall rate is replaced by the individual branch rates.

$$\frac{\partial \log L}{\partial t_k} = \frac{-x_k}{t_{k'} - t_k} + \lambda_k \text{ \{this term included whenever } k \text{ not the root\}}$$

$$+ \sum_{j \in D(k)} \left\{ \frac{x_j}{t_k - t_j} - \lambda_j \right\} \text{\{this term included whenever } k \text{ not a tip\}}$$

The gradient with respect to rates is more complicated. It includes terms from both the likelihood and the penalty portion of the objective function.

$$\frac{\partial \log \Phi}{\partial \lambda_k} = \frac{\partial \log L}{\partial \lambda_k} - \gamma \frac{\partial \pi}{\partial \lambda_k}$$

The first term is easy:

$$\frac{\partial \log L}{\partial \lambda_k} = \frac{x_k}{\lambda_k} - (t_{k'} - t_k)$$

For the second term we can use the following pseudo-code to show how it is calculated. There are two different versions depending on whether the additive or logarithmic penalty is used.

```
[Additive penalty]
For each interior node k
    {
    if (k′ == root)
            {
            $\pi(\lambda_1,...,\lambda_n) = ... + \mathrm{Var}\left(\lambda_r : r \in D(k')\right) + \sum_{j \in D(k)} \left(\lambda_j - \lambda_k\right)^2 + ...$

            $\mathrm{Var}\left(\lambda_r : r \in D(k')\right) = \frac{1}{n_{D(k)}} \sum_{j \in D(k')} \lambda_j^{\ 2} - \left( \frac{1}{n_{D(k)}} \sum_{j \in D(k')} \lambda_j \right)^2$

            $\frac{\partial}{\partial \lambda_k} \pi(\lambda_1,...,\lambda_n) = \frac{2\left(\lambda_k - \overline{\lambda}_{j \in D(k')}\right)}{n_{D(k')}} - 2 \sum_{j \in D(k)} \left(\lambda_j - \lambda_k\right)$

            }
    else (k′ != root)
        {
        if (k == tip)
            {
            $\pi(\lambda_1,...,\lambda_n) = ... + \left(\lambda_k - \lambda_{k'}\right)^2 + ...$

            $\frac{\partial}{\partial \lambda_k} \pi(\lambda_1,...,\lambda_n) = 2\left(\lambda_k - \lambda_{k'}\right)$

            }
        else (k != tip)
            {
            $\pi(\lambda_1,...,\lambda_n) = ... + \left(\lambda_k - \lambda_{k'}\right)^2 + \sum_{j \in D(k)} \left(\lambda_j - \lambda_k\right)^2 + ...$

            $\frac{\partial}{\partial \lambda_k} \pi(\lambda_1,...,\lambda_n) = 2\left(\lambda_k - \lambda_{k'}\right) - 2 \sum_{j \in D(k)} \left(\lambda_j - \lambda_k\right)$

            }
        }
    }
```

```
[logarithmic penalty]
For each interior node k
        {
        if (k' == root)
                {
                $\pi(\lambda_1,...,\lambda_n) = ... + \mathrm{Var}\left(\log\lambda_r : r \in D(k')\right) + \sum_{j\in D(k)}\left(\log\lambda_j - \log\lambda_k\right)^2 + ...$

                $\mathrm{Var}\left(\log\lambda_r : r \in D(k')\right) = \frac{1}{n_{D(k)}}\sum_{j\in D(k')}(\log\lambda_j)^2 - \left(\frac{1}{n_{D(k)}}\sum_{j\in D(k')}\log\lambda_j\right)^2$

        $\frac{\partial}{\partial\lambda_k}\pi(\lambda_1,...,\lambda_n) = +\frac{2\left(\log\lambda_k - \langle\log\lambda\rangle_{j\in D(k')}\right)}{\lambda_k n_{D(k')}} - \frac{2}{\lambda_k}\sum_{j\in D(k)}\left(\log\lambda_j - \log\lambda_k\right)$

                }
        else (k' != root)
                {
                if (k == tip)
                        {
                        $\pi(\lambda_1,...,\lambda_n) = ... + \left(\lambda_k - \lambda_{k'}\right)^2 + ...$

                        $\frac{\partial}{\partial\lambda_k}\pi(\lambda_1,...,\lambda_n) = \frac{2}{\lambda_k}\left(\log\lambda_k - \log\lambda_{k'}\right)$

                        }
                else (k != tip)
                        {
                        $\pi(\lambda_1,...,\lambda_n) = ... + \left(\lambda_k - \lambda_{k'}\right)^2 + \sum_{j\in D(k)}\left(\lambda_j - \lambda_k\right)^2 + ...$

                        $\frac{\partial}{\partial\lambda_k}\pi(\lambda_1,...,\lambda_n) = \frac{2}{\lambda_k}\left(\log\lambda_k - \log\lambda_{k'}\right) - \frac{2}{\lambda_k}\sum_{j\in D(k)}\left(\log\lambda_j - \log\lambda_k\right)$

                        }
                }
        }
```