Emmanuel Paradis

Analysis of Phylogenetics and Evolution with R

Springer

Emmanuel Paradis
Institut de Recherche pour le Développement
UR 175 Caviar
GAMET-BP 5095
361 rue Jean François Breton
F-34196 Montpellier cédex 5
France
Emmanuel.Paradis@mpl.ird.fr

Series Editors:
Robert Gentleman
Program in Computational Biology
Division of Public Health Sciences
Fred Hutchinson Cancer Research Center
1100 Fairview Ave. N, M2-B876
Seattle, Washington, 981029-1024
USA

Kurt Hornik
Department für Statistik und Mathematik
Wirtschaftsuniversität Wien Augasse 2-6
A-1090 Wien
Austria

Giovanni Parmigiani
The Sidney Kimmel Comprehensive
Cancer Center at Johns Hopkins University
550 North Broadway
Baltimore, MD, 21205-2011
USA

Library of Congress Control Number: 2006923823

ISBN-0-387-32914-5
ISBN-978-0387-32914-7

Printed on acid-free paper.

Printed in the United States of America. (MVY)

9 8 7 6 5 4 3 2 1

springer.com

Preface

As a result, the inference of phylogenies often seems divorced from any connection to other methods of analysis of scientific data.

Felsenstein

Once calculation became easy, the statistician's energies could be devoted to understanding his or her dataset.

Venables & Ripley

The study of the evolution of life on Earth stands as one of the most complex fields in science. It involves observations from very different sources, and has implications far beyond the domain of basic science. It is concerned with processes occurring on very long time spans, and we now know that it is also important for our daily lives as shown by the rapid evolution of many pathogens.

As a field ecologist, for a long time I was remotely interested in phylogenetics and other approaches to evolution. Most of the work I accomplished during my doctoral studies involved field studies of small mammals and estimation of demographic parameters. Things changed in 1996 when my interest was attracted by the question of the effect of demographic parameters on bird diversification. This was a new issue for me, so I searched for relevant data analysis methods, but I failed to find exactly what I needed. I started to conduct my own research on this problem to propose some, at least partial, solutions. This work made me realize that this kind of research critically depends on the available software, and it was clear to me that what was offered to phylogeneticists at this time was inappropriate.

I first read about R in 1998 while I was working in England: I first tried it on my computer in early 1999 after I got a position in France. I quickly thought that R seemed to be the computing system that is needed for developing phylogenetic methods: versatile, flexible, powerful, with great graphical possibilities, and free.

When I first presented the idea to develop programs written in R for phylogenetic analyses in 2001, the reactions from my colleagues were mixed with enthusiasm and scepticism. The perspective of creating a single environment for phylogenetic analysis was clearly exciting, but some concerns were expressed about the computing performance of R which, it was argued, could not match those of traditional phylogenetic programs. Another criticism was that biologists would be discouraged from using a program with a command-line interface. The first version of the R package ape was eventually released in August 2002. The reactions from some colleagues showed me that related projects were undertaken elsewhere.

The progress accomplished has been much more than I expected, and the perspectives are far reaching. Writing a book on phylogenetics with R is an opportunity to bring together pieces of information from various sources, programs, and packages, as well as discussing a few ideas.

I realize that the scope of the book is large, and the treatment may seem superficial in some places, but it was important to treat the present topics in a concise manner. It was not possible to explore all the potentialities now offered by R and its packages written for phylogenetic analysis. Similarly, I tried to explain the underlying concepts of the methods, sometimes illustrated with R codes, but I meant to keep it short as well.

I must first thank the "R community" of developers and users from whom I learned much about R through numerous exchanges on the Internet: this definitely helped me to find my way and envision the development of ape. Julien Claude has shared the venture of developing programs in R and contributing to ape since he was a doctoral student. A great thank you to those who contributed some codes to ape: Korbinian Strimmer, Gangolf Jobb, Rainer Opgen-Rhein, Julien Dutheil, Yvonnick Noël, and Ben Bolker. I must emphasize that all these authors should have full credit for their contributions. I am grateful to Olivier François and Michael Blum for showing me the possibilities of their package apTreeshape.

Several colleagues kindly read some parts of the manuscript: Lounès Chikki, Julien Claude, Jean Lobry, Jean-François Renno, Christophe Thébaud, Fabienne Thomarat, and several colleagues who chose to remain anonymous. Thanks to all of them! Special thanks to Susan Holmes for encouragement and some critical comments. Thank you to Elizabeth Purdom and Julien Dutheil for discussions about ape and R programming. I am sincerely thankful to John Kimmel at Springer for the opportunity to write this book, and for managing all practical aspects of this project. Finally, many thanks to Diane Sahadeo for handling my manuscript to make it an actual book.

Jakarta Emmanuel Paradis
April 2006

Contents

1

Introduction

Phylogenetics is the science of the evolutionary relationships among species. Recently, the term has come to include broader issues such as estimating rates of evolution, dating divergence among species, reconstructing ancestral characters, or quantifying adaptation, all these using phylogenies as frameworks.

Computers seem to have been used by phylogeneticists as soon they were available in research departments [28]. Since then, progress has been obvious in two parallel directions: biological databases, particularly for molecular sequences, have increased in quantity at an exponential rate and, at the same time, computing power has grown at an expanding pace. These concurrent escalations have resulted in the challenge of analyzing larger and larger data sets using more and more complex methods.

The current complexity of phylogenetic analyses implies some strategic choices. This chapter explains the advantages of R as a system for phylogenetic analyses.

1.1 Strategic Considerations

How data are stored, handled, and analyzed with computers is a critical issue. This is a strategic choice as this conditions what can subsequently be done with more or less efficiency.

R is a language and environment for statistical and graphical analyses [74]. It is flexible, powerful, and can be interfaced with several systems and languages. R has many attractive features: we concentrate on four of them that are critical for phylogenetic analyses.

Integration

Phylogenetics covers a wide area of related issues. Analyzing phylogenetic data often implies doing different analyses such as tree estimation, dating divergence times, and estimating speciation rates. The implementation of these methods in R enhances their integration under a single user interface. It should be pointed out that although the development of phylogenetic methods in R is relatively recent, a remarkable range of methods is already available.

Integration is not new among phylogenetic analysis programs and the most widely used ones cover a wide range of methods. However, this feature combined with those detailed below, has a particular importance not observed in these programs.

A less obvious aspect of integration is the possibility of using different languages and systems from the same user interface. This is called *intersystems interfaces* and has been particularly developed in R [49]. The most commonly used interfaces in R are with programs written in C, C++, or FORTRAN, but there exist interfaces with PERL, Python, and Java.[1] The gain from these interfaces is enormous: developers can use the languages or systems they prefer to implement their new methods, and users do not have to learn a new interface to access the last methodological developments.

Interactivity

Interactivity is critical in the analysis of large data sets with a great variety of methods. Exploratory analyses are crucial for assessing data heterogeneity. Selection of an appropriate model for estimation often needs to interactively fit several models. Examination of model output is also often very useful (e.g., plot of regression diagnostics).

In phylogenetic analyses, the usual computer program strategy follows a "black box" model where some data, stored in files, are read by a specific program, some computations are made, and the results are written into a file on the disk. What happens in the program cannot be accessed by the user. Several program executions can be combined using a scripting language, but such programming tasks are generally limited.

R does not follow this model. In R, the data are read from files and stored in active memory: they can be manipulated, plotted, analyzed, or written into files. The results of analyses are treated exactly in the same way as data. In R's jargon, the data in memory are called *objects*. Considering data as objects makes good sense in phylogenetics because this allows us to manipulate different kinds of data (trees, phenotypical data, geographical data) simultaneously and interactively.

Programmability

Data analyses are almost always made of a series of more or less simple tasks. These analyses need to be repeated for many reasons. The most usual situation is that new data have been collected and previous analyses need to be updated. It is thus very useful to automate such analyses, particularly if they are composed of a long series of smaller analyses.

[1] http://www.omegahat.org/.

R is a flexible and powerful language that can be used for simple tasks as well as combining a series of analyses. The programmability of R can be used at a more complex level to develop new methods (Chapter 7). R is an interpreted language meaning that there is no need to develop a full program to perform an analysis. A simple command may need a single line.

Programmability is important in the context of scientific repeatability. Writing programs that perform data analyses (often called *scripts*) ensures better readability, and improves repeatability by others [49]. In this context, there exist some sophisticated solutions, such as `Sweave` (in the package `utils`) which mixes data analysis commands with R and text processing with LaTeX [91] (see also `?Sweave` in R).

Evolvability

Phylogenetic methods have considerably evolved for several decades, and this is likely to go on in the future. An efficient data analysis system needs to evolve with respect to the new methodological developments. Programs written in R are easy to maintain because programming in this language is very simple. Bugs are much easier to be found and fixed than in a compiled language inasmuch as there is no need to manage memory allocation (one of the main time-consuming tasks of progammers).

R's syntax and function definitions ensure compatibility through time in most cases. For instance, consider a function called `foo` which has a single argument `x`. Thus the user will call this function with something such as:

```
foo(x = mydata)
```

If, for any reason, `foo` changes to include other options that have default values, say `y = TRUE` and `z = FALSE`, then the above command will still work with the new version of `foo`.

In addition, the internal structure and functionalities of R evolve with respect to technological developments. Thus using R as a computing environment eases tracking novelties in this area.

R has other strengths as a computing environment. It is scalable: it can run on a variety of computers with a range of hardware, and can be adapted for the analysis of large data sets. On the lower bound, R can run with as few as 16 Mb of RAM,[2] whereas on the upper bound R can be compiled and run on 64-bit computers and thus use more than 4 Gb of RAM. Furthermore, there are packages to run R on multiprocessor computers.

R has very good computing performance: most of its operations are vectorized, meaning that as little time as possible is spent on the evaluation of commands. The graphical environment of R is flexible and powerful giving many possibilities for graphical analyses (Chapter 4).

R is an environment suitable both for basic users (e.g., biologists) and for developers. This considerably enhances the transfer of new methodological

[2] For instance, R can be run under Linux with 32 Mb of RAM.

developments. R can run on most current operating systems: all commands are fully compatible across systems (they are *portable* in computers jargon).

Finally, R is distributed under the terms of the GNU General Public License, meaning that its distribution is free, it can be freely modified, and it can be redistributed under certain conditions.[3] There have been numerous discussions, particularly on the Internet, about the advantages and inconveniences of free software. The crucial points are not that R is free to download and install (this is true for much industrial software), but that it can be modified by the user, and its development is open to contributions.[4] Although it is hard to assess, it is reasonable to assume that such an open model of software development is more efficient—but not always more attractive to all users—than a proprietary model (see [49] for some views on this issue). All computer programs presented in this book are freely distributed.

1.2 Notations

Commands typed in R are printed with a `fixed-spaced font`, usually on separate lines. The same font is used for the names of objects in R (functions, data, options). Names of packages are printed with a sans-serif font.

When necessary, a command is preceded by the symbol `>`, which is the usual prompt in R, to distinguish what is typed by the user from what is printed (or returned) by R. For instance:

```
> x <- 1
> x
[1] 1
```

In the R language, `#` specifies a comment: everything after this character is ignored until the next line. This is sometimes used in the printed commands:

```
mean(x) # get the mean of x
```

When an output from R is too long, it is cut after "`....`". For instance, if we look at the content of the function `plot`:

```
> plot
function (x, y, ...)
{
    if (is.null(attr(x, "class")) && is.function(x)) {
....
```

Names of files are within 'single quotes'. Their contents are indicated within a frame:

[3] See the file "R_HOME/COPYING" for details.
[4] For obvious practical reasons, a limited number of persons, namely, the members of the R Core Team, can modify the original sources.

```
x y
1 3.5
2 6.9
```

1.3 Preparing the Computer

R is a modular system: a base installation is composed of a few dozen packages for reading/writing data, classical data analyses methods, and computational statistical utilities. Several hundred contributed packages add many specialized methods.[5] Note that in R's terminology, a *package* is a set of files that perform some specific tasks within R, and that include the related documentation and any needed files. An R package requires R to run.

R can be installed on a wide range of operating systems: sources and precompiled versions, as well as the installation instructions, can be found at the Comprehensive R Archive Network (CRAN):

http://cran.r-project.org/

1.3.1 Installations

Phylogenetic analyses in R use, of course, the default R packages, but also a few specialized ones that need to be installed by the user. Table 1.1 lists the packages that are discussed in this book.

Table 1.1. R packages used for phylogenetic analyses. The packages marked with (d) are installed by default with R. The "Requires" column indicates the nondefault R packages that are needed

Name	Title	Requires
base (d)	R base package	—
stats (d)	R stats package	—
graphics (d)	R graphics package	—
nlme (d)	Mixed-effects models	—
lattice (d)	Lattice graphics	—
ape	Analyses of phylogenetics and evolution	gee
apTreeshape	Analyses of phylogenetic tree shape	ape
ade4	Analysis of environmental data	—
seqinr	Exploratory analyses of molecular sequences	—

[5] A complete list of R's packages with descriptions can be found at http://cran.r-project.org/src/contrib/PACKAGES.html.

The installation of R packages depend on the way R was installed, but usually the following command in R will work provided the computer is connected to the Internet:

```
install.packages("ape")
```

and the same for all needed packages. Once the packages are installed, they are available for use after being loaded in memory which is usually done by the user:

```
> library(ape)
Loading required package: gee
Loading required package: nlme
Loading required package: lattice
> library(ade4)
> library(seqinr)
```

ape is dedicated to phylogenetic and evolutionary analyses, thus we concentrate a large part of our attention on this package. apTreeshape deals with the analysis of tree shape and has several functions to query tree databases through the Internet. ade4 is dedicated to the analysis of environmental data, but it has several functionalities that complement ape. seqinr (*sequences in R*) is a package for reading and handling molecular sequences (protein and DNA). It has some functions for graphical and exploratory analyses of this kind of data.

Most R packages include a few data sets to illustrate how the functions can be used. These data are loaded in memory with the function data. We of course use them in our examples.

Additionally to these add-on packages, it is useful to have the computer connected to the Internet because some functions connect to remote databases (e.g., ape and seqinr can read DNA sequences in GenBank).

Other programs may be required in some applications. PHYML is called by ape with its function phymltest; it is available at

http://atgc.lirmm.fr/phyml/

A multiple sequence alignment program is also very useful because this operation is not really feasible in R. Clustal X [153] is widely used and available for most operating systems. There are also several interfaces to the Clustal computing engine, such as the Web-interface ClustalWWW [17]. Clustal X is available at

http://www-igbmc.u-strasbg.fr/BioInfo/ClustalX/

Note the existence of the R package dna by Jim Lindsey which includes a version of Clustal W (the computing engine of Clustal X). It is available at

http://popgen.unimaas.nl/~jlindsey/rcode.html

We use Clustal X because sequence alignments can be graphically visualized.

Additionally to these required programs, a few others are useful when using R. Emacs is a flexible text editor that runs under most operating systems. It can be used to edit R programs. Installing the ESS (*Emacs Speaks Statistics*) package allows syntax highlighting, and other facilities such as running R within Emacs. Emacs and ESS can be downloaded at, respectively

http://www.gnu.org/software/emacs/emacs.html
http://ess.r-project.org/

Ghostscript and GSview are two programs to view and convert files in PostScript and PDF formats: they can be used to view the figures made with R (Chapter 4). They can be downloaded at

http://www.cs.wisc.edu/˜ghost/

Finally, a Web browser is useful to view the R help pages in HTML format.

1.3.2 Configurations

Once all packages and software are installed, the computer is ready. There is no special need for the location of data files: they are accessed in the usual way by R. When R is started, a working directory is set. Under UNIX-like systems, this is usually the directory where R has been launched. Under Windows, this is the directory where the executable is located, or if R is started from a short-cut, the directory specified in the "Start-in" field of this short-cut.[6] On all systems, the working directory is displayed in R with the function `getwd()` (*get working directory*); it can be modified with `setwd`:

```
> setwd("/home/paradis/phylo/data") # Linux
> setwd("D:/data/phylo/data")       # Windows
```

Note the use of the forward slashes (`/`), even under Windows, because the backslashes (\) have a specific meaning in character strings.

If a file is not in the working directory, it can be accessed by adding the full path in the `file` argument, for instance, when reading a tree (see Section 3.2):

```
> tr <- read.tree("/home/paradis/phylo/data/treeb1.txt")
```

The same comment applies when writing into a file: the file is written in the current working directory unless a path is given in the `file` argument exactly in the same way as above.

Emacs and ESS need slightly more configuration if the user wants to run R within Emacs. This is essentially system dependent; the critical step is to

[6] This can be modified by the user by editing the properties of the short-cut, usually by right-clicking on its icon. A standard installation under Windows puts a short-cut of R on the Desktop.

tell Emacs where to find R's executable. ESS is distributed with several documentation files detailing the installation and configuration for the different operating systems.

2

First Steps in R for Phylogeneticists

It is clear that some experience with R greatly helps in handling the materials presented in this book. The goal of this chapter is to give the first steps for new users of R. It is focused on the topics required for the present book, and does not cover all introductory concepts and notions about R.

A generally deterring fact for new users of R is that it is almost impossible to figure out what to do if the user has no notion of languages, commands, or R itself. A learning step must be taken and this obviously has a cost. Progressing in the use of R involves successive learning steps. Of course, there are benefits to taking these steps.

R has spread through the field of computational statistics, and there is now a wide range of packages for many numerical, analytical, and graphical methods. The fields of application of R include analysis of DNA microarray data,[1] genetics (quantitative trait loci, population analyses, etc.), morphometrics, ecological analyses, drawing maps including the use of geographic information systems (GIS) data, and interacting with a variety of other programs such as SQL databases. Thus learning R for a specific task is very likely to be rewarding very rapidly.

If you do not know R, do not have knowledge of computer languages, and do not want to read introductory documents on R[2] (or cannot), then you should read, certainly carefully, this chapter. If you already have an idea of computer programming but not R, reading this chapter should be easy and will point to the particularities of R.

2.1 The Command Line Interface

The user can interact with R in several ways. The most interactive way is to use the command line interface (CLI). R can also be run in batch mode

[1] http://www.bioconductor.org/.

[2] See http://cran.r-project.org/manuals.html and http://cran.r-project.org/other-docs.html.

(i.e., noninteractive) from a system shell. There are several graphical user interfaces (GUIs), but they are restricted to traditional statistical methods (see the Rcmdr package), and so do not cover the wide range of methods available in R. Finally, there exist several Web servers to run R through the Internet. In this book, we concentrate on the CLI because it is interactive, versatile, and portable (i.e., the commands will run on all operating systems).

All actions are done on data stored in the active memory of the computer. These data are stored as *objects*. To characterize some data, and thus analyze them relevantly, it is often necessary to have additional information. For instance, consider a numeric variable taking the values 0 or 1: is it a count (i.e., a quantitative variable) or a code for a qualitative variable? In R the required information is provided by the *attributes* of the objects. We show some examples in the next section.

Commands in R are made of *functions* and / or *operators* (`+`, `-`, `*`, etc). A command returns an object that is either displayed on the screen (and not stored in memory), or stored in memory using the operator “assign” `<-`. The latter requires giving a name to the object. An object may be displayed by typing its name as a command:

```
> 2 + 7
[1] 9
> x <- 2 + 7
> x
[1] 9
```

R has a wide range of functions and operators to create regular and random sequences. There are also several functions to read data from files on the disk: the most useful for us are illustrated in Section 3.6.

The user does not see her data as in a spreadsheet editor because many objects with different structure can be stored and manipulated at the same time, and this cannot be represented as a spreadsheet. There are, of course, several functions to manage the objects in memory. `ls` displays a simple list of the objects currently in memory.

```
> ls()
character(0)
> n <- 5
> ls()
[1] "n"
> x <- "acgt"
> ls()
[1] "n" "x"
```

As we have seen above, typing the name of an object as a command displays its content. In order to display some attributes of the object, one can use the function `str` (*structure*):

```
> str(n)
 num 5
> str(x)
 chr "acgt"
```

This shows that `n` is a numeric object, and `x` is a character one. Both `ls` and `str` can be combined by using the function `ls.str`:

```
> ls.str()
n :  num 5
x :  chr "acgt"
```

To delete an object in memory, the function `rm` must be used:

```
> ls()
[1] "n" "x"
> rm(n)
> ls()
[1] "x"
```

There are one function and one operator that are good to learn very early because they are used very often in R: `c` concatenates several elements to produce a single one, and `:` returns a regular series where two successive elements differ by one. Here are some examples:

```
> x <- c(2, 6.6, 9.6)
> x
[1] 2.0 6.6 9.6
> y <- 2.2:5.2
> y
[1] 2.2 3.2 4.2 5.2
> c(x, y)
[1] 2.0 6.6 9.6 2.2 3.2 4.2 5.2
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
```

2.2 The Data Structures

We show here how data are stored in R, and how to manipulate them.

2.2.1 Vector

Vectors are the basic data structures in R. A vector is a series of elements that are all of the same type. A vector has two attributes: the *mode*, which characterizes the type of data, and the *length*, which is the number of elements. There are four modes: numeric, logical (`TRUE` or `FALSE`), character, and complex. The last mode is seldom used and is not discussed here.

When a vector is created or modified, there is no need to specify its mode and length: this is dealt with by R. It is possible to check these attributes with the functions of the same names:

```
> x <- 1:5
> mode(x)
[1] "numeric"
> length(x)
[1] 5
```

Logical vectors are created by typing “FALSE” or “TRUE”:

```
> y <- c(FALSE, TRUE)
> y
[1] FALSE  TRUE
> mode(y)
[1] "logical"
> length(y)
[1] 2
```

In most cases, a logical vector results from a logical operation, such as the comparison of two values or two objects:

```
> 1 > 0
[1] TRUE
```

A vector of mode character is a series of character strings (and not of single characters):

```
> z <- c("order", "family", "genus", "species")
> mode(z)
[1] "character"
> length(z)
[1] 4
> z
[1] "order"   "family"  "genus"   "species"
```

We have just seen how to create vectors by typing them on the CLI, but it is clear that in the vast majority of cases they will be created by reading data from files.

As already mentioned, a function returns an object that is itself characterized by its mode. From the examples just above, it can be seen that `mode` returns a vector of mode character, whereas `length` returns one of mode numeric. The same applies to the functions introduced above, and in particular `ls` which returns a vector of mode character.

R has a powerful and flexible mechanism to manipulate vectors (and other objects as well): the indexing system. There are three kinds of indexing: numeric, logical, and with names.

The numeric indexing works by giving the indices of the elements that must be selected. Of course, this can be given as a numeric vector:

```
> z[1:2]
[1] "order"   "family"
> i <- c(1, 3)
> z[i]
[1] "order"   "genus"
```

This can be used to repeat a given element:

```
> z[c(1, 1, 1)]
[1] "order" "order" "order"
> z[c(1, 1, 1, 4)]
[1] "order"   "order"   "order"   "species"
```

If the indices are negative, then the corresponding values are removed:

```
> z[-1]
[1] "family"  "genus"   "species"
> j <- -c(1, 4)
> z[j]
[1] "family"  "genus"
```

Positive and negative indices cannot be mixed. If a positive index is out of range, then a missing value (NA, for *not available*) is returned, but if the index is negative, an error occurs:

```
> z[5]
[1] NA
> z[-5]
Error: subscript out of bounds
```

The indices may be used to extract some data, but also to change them:

```
> x[c(1, 4)] <- 10
> x
[1] 10  2  3 10  5
```

The logical indexing works differently than the numeric one. Logical values are given as indices: the elements with an index TRUE are selected, and those with FALSE are removed. If the number of logical indices is shorter than the vector, then the indices are repeated as many times as necessary; for instance, the two commands below are strictly equivalent:

```
> z[c(TRUE, FALSE)]
[1] "order" "genus"
> z[c(TRUE, FALSE, TRUE, FALSE)]
[1] "order" "genus"
```

As with numeric indexing, the logical indices can be given as a logical vector. The logical indexing is a powerful and simple way to select some data from a vector: for instance, if we want to select the values greater than or equal to five in `x`:

```
> x[x >= 5]
[1] 10 10  5
```

The indexing system with names brings us to introduce a new concept: a vector may have an attribute called *names* that is a vector of mode character of the same length, and serves as labels. It is created or extracted with the function `names`. An example could be:

```
> x <- 4:1
> names(x) <- z
> x
  order  family   genus species
      4       3       2       1
> names(x)
[1] "order"   "family"  "genus"   "species"
```

These names can then be used to select some elements of a vector:

```
> x[c("order", "genus")]
order genus
    4     2
```

In some situations it is useful to delete the names of a vector; this is done by giving them the value `NULL`:

```
> names(x) <- NULL
> x
[1] 4 3 2 1
```

2.2.2 Factor

A factor is a data structure derived from a vector, but it is not the same strictly speaking. It can be of mode numeric or character, and has an attribute `"levels"` which is a vector of mode character and specifies the possible values the factor can take. If a factor is created with the function `factor`, then the levels are defined with all values present in the data:

```
> f <- c("Male", "Male", "Male")
> f
[1] "Male" "Male" "Male"
> f <- factor(f)
> f
[1] Male Male Male
Levels: Male
```

To specify that other levels exist although they are not observed in the present data, the option `levels` can be used:

```
> ff <- factor(f, levels = c("Male", "Female"))
> ff
[1] Male Male Male
Levels: Male Female
```

This is a crucial point when analyzing this kind of data, for instance, if we compute the frequencies in each category with the function `table`:

```
> table(f)
f
Male
   3
> table(ff)
ff
  Male Female
     3      0
```

Factors can be indexed and have names exactly in the same way as vectors.

When data are read from a file on the disk with the function `read.table`, the default is to treat all character strings as factors (see Chapter 3.6 for examples). This can be avoided by using the option `as.is = TRUE`.

2.2.3 Matrix

A matrix can be seen as a vector arranged in a tabular way. It is actually a vector with an additional attribute called *dim* (dimensions) which is itself a numeric vector with length 2, and defines the numbers of rows and columns of the matrix.

There are two basic ways to create a matrix: either by using the function `matrix` with the appropriate options `nrow` and `ncol`, or by setting the attribute dim of a vector:

```
> matrix(1:9, 3, 3)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> x <- 1:9
> dim(x) <- c(3, 3)
> x
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

The numeric and logical indexing systems work in exactly the same way as for vectors. Because a matrix has two dimensions, it can be indexed with two integers separated by a comma:

```
> x[3, 2]
[1] 6
```

If one wants to extract only a row or a column, then the appropriate index must be omitted (without forgetting the comma):

```
> x[3, ] # extract the 3rd row
[1] 3 6 9
> x[, 2] # extract the 2nd column
[1] 4 5 6
```

In contrast to vectors, a subscript out of range results in an error.

Matrices do not have names in the same way as vectors, but have rownames, colnames, or both:

```
> rownames(x) <- c("A", "B", "C")
> colnames(x) <- c("v1", "v2", "v3")
> x
  v1 v2 v3
A  1  4  7
B  2  5  8
C  3  6  9
```

Selection of rows and / or columns follows in nearly the same ways as seen before:

```
> x[, "v1"]
A B C
1 2 3
> x["A", ]
v1 v2 v3
 1  4  7
> x[c("A", "C"), ]
  v1 v2 v3
A  1  4  7
C  3  6  9
```

2.2.4 Data Frame

A data frame is superficially similar to a matrix in the sense that it is a tabular representation of data. The distinction is that a data frame is a set of distinct vectors and / or factors all of the same length, but possibly of different modes.

Data frames are the main way to represent data sets in R because this corresponds roughly to a spreadsheet data structure. This is the type of objects returned by the function `read.table` (see Section 3.6 for examples). The other way to create data frames is with the function `data.frame`:

```
> DF <- data.frame(z, y = 0:3, 4:1)
> DF
        z y X4.1
1   order 0    4
2  family 1    3
3   genus 2    2
4 species 3    1
> rownames(DF)
[1] "1" "2" "3" "4"
> colnames(DF)
[1] "z"    "y"    "X4.1"
```

This example shows how colnames are created in different cases. By default, the rownames `"1"`, `"2"`, ... are given, but this can be changed with the option `row.names` of `data.frame`, or modified subsequently as seen above for matrices.

If one of the vectors is shorter, then it is recycled along the data frame but this must be an integer number of times:

```
> data.frame(1:4, 9:10)
  X1.4 X9.10
1    1     9
2    2    10
3    3     9
4    4    10
> data.frame(1:4, 9:11)
Error in data.frame(1:4, 9:11) :
  arguments imply differing number of rows: 4, 3
```

All we have seen about indexing, colnames, and rownames for matrices apply in exactly the same way to data frames with the difference that colnames and rownames are mandatory for data frames. An additional feature of data frames is the possibility of extracting a column selectively with the operator `$`:

```
> DF$y
[1] 0 1 2 3
```

2.2.5 List

Lists are the most general data structure in R: they can contain any kind of objects, even lists. They can be seen as vectors where the elements can be any kind of object. They are built with the function `list`:

```
> L <- list(z = z, 1:2, DF)
> L
$z
[1] "order"   "family"  "genus"   "species"

[[2]]
[1] 1 2

[[3]]
        z y X4.1
1   order 0    4
2  family 1    3
3   genus 2    2
4 species 3    1

> length(L)
[1] 3
> names(L)
[1] "z" ""  ""
```

Most of the concepts we have seen on indexing vectors apply also to lists. Additionally, an element of a list may be extracted either with its index within double square brackets, or with the operator `$`:

```
> L[[1]]
[1] "order"   "family"  "genus"   "species"
> L$z
[1] "order"   "family"  "genus"   "species"
```

2.3 The Help System

Every function in R is documented through a system of help pages available in different formats:

- Simple text that can be displayed from the CLI;
- HTML that can be browsed with a Web browser (with hyperlinks between pages where available);
- PDF that constitutes the manual of the package.

The contents of these different documents are the same.

Through the CLI a help page may be displayed with the function `help` or the operator `?` (the latter does not work with special characters such as the operators):

```
help("ls")
?ls
```

By default, `help` only searches in the packages already loaded in memory. The option `try.all.packages = TRUE` allows us to search in all packages installed on the computer.

If one does not know the name of the function that is needed, a search with keywords is possible with the function `help.search`. This looks for a specified topic, given as a character string, in the help pages of all installed packages. For instance:

```
help.search("tree")
```

will display a list of the functions where help pages mention "tree". If some packages have been recently installed, it may be necessary to refresh the database used by `help.search` using the option `rebuild = TRUE`.

Another way to look for a function is to browse the help pages in HTML format. This can be launched from R with the command:

```
help.start()
```

This loads in the local Web browser a page with links to all the documentation installed on the computer, including general documents on R, an FAQ, links to Internet resources, and the list of the installed packages. This list has itself links to the list of all functions with their help pages.

2.4 Creating Graphics

The graphical functions in R need a special mention because they work somewhat differently from the others. A graphical function does not return an object (though there are a few exceptions), but sends its results to a *graphical device* which is either a graphical window (by default) or a graphical file. The graphical formats depend on the operating systems, but mostly the following are available: encapsulated PostScript (EPS), PDF, JPEG, PNG, and bitmap (BMP). Additionally, xfig is possible under Linux, and EMF under Windows.

There are two ways to write graphics into a file. The most general and flexible way is to open the appropriate device explicitly, for instance, if we write into an EPS file:

```
postscript("plot.eps")
```

then all subsequent graphical commands will be written in the file 'plot.eps'. The operation is terminated (i.e., the file is closed and written on the disk) with the command:

```
dev.off()
```

The function `postscript` has many options to set the EPS files. All the figures of this book have been produced with this function.

The second way is to copy the content of the window device into a file using the function `dev.copy` where the user must specify the target device. Two variants of this function are `dev.print` which prints into an EPS file, and `dev.copy2eps` which does the same by setting the page in portrait format.

2.5 Saving and Restoring R Data

R uses two basic formats to save data: ASCII (simple text) and XDR (external data representation[3]). They are both cross-platform. The ASCII format is used to save a single object (vector, matrix, or data frame) into a file. Two functions can be used: `write` (for vectors and matrices) and `write.table` (for data frames). The XDR format can be used to save any kind and any number of objects. It is used with the function `save`, for instance, to save three objects:

```
save(x, y, z, file = "xyz.RData")
```

These data can then be restored with:

```
load("xyz.RData")
```

2.6 Using R Functions

Now that we have seen a few instances of R function uses, we can draw some general conclusions on this point.

To execute a function, the parentheses are always needed, even if there is no argument inside (typing the name of a function without parentheses prints its contents). The arguments are separated with commas. There are two ways to specify arguments to a function: by their positions or by their names (also called *tagged arguments*). For example, let us consider a hypothetical function with three arguments:

```
fcn(arg1, arg2, arg3)
```

`fcn` can be executed without using the names `arg1`, ..., if the corresponding objects are placed in the correct position, for instance, `fcn(x, y, z)`. However, the position has no importance if the names of the arguments are used, for example, `fcn(arg3 = z, arg2 = y, arg1 = x)`. Another feature of R's functions is the possibility of using default values (also called *options*), for instance, a function defined as:

```
fcn(arg1, arg2 = 5, arg3 = FALSE)
```

[3] http://www.faqs.org/rfcs/rfc1832.html.

Both commands `fcn(x)` and `fcn(x, 5, FALSE)` will have exactly the same result. Of course, tagged arguments can be used to change only some options (e.g., `fcn(x, arg3 = TRUE)`).

Many functions in R act differently with respect to the type of object given as arguments: these are called *generic* functions. They act with respect to an optional object attribute: the *class*. The main generic functions in R are `print`, `summary`, and `plot`. In R's terminology, `summary` is a generic function, whereas the functions that are effectively used (e.g., `summary.phylo`, `summary.default`, `summary.lm`, etc.) are called *methods*.

In practice, the use of classes and generics is implicit, but we show in the next chapter that different ways to code a tree in R correspond to different classes. The advantage of the generic functions here is that the same command is used for the different classes (e.g., `plot(tr)` to draw a tree).

2.7 Repeating Commands

When it comes to repeating some analyses, several strategies can be used. The simplest way is to write the required commands in a file, and read them in R with the function `source`. It is usual to name such files with the extension '.R'. For instance, the file 'mytreeplot.R' could be:

```
tree1 <- read.tree("tree1.tre")
postscript("tree1.eps")
plot(tree1)
dev.off()
```

These commands will be executed by typing `source("mytreeplot.R")` in R.

2.7.1 Loops

As with any language, R has control and programming structures to execute a series of commands. The most often-used one is the `for`[4] statement, whose general syntax is:

```
for (x in y) <command>
```

where `y` is an object, and `x` successively takes the different values of `y`. It is not required to use these values in `<command>` (e.g., `for (i in 1:5) print("done")`). A `for` loop may encompass more than one command in which case it is necessary to group them within braces:

[4] The following words are reserved to the R language and cannot be used to name objects: `for`, `in`, `if`, `else`, `while`, `next`, `break`, `repeat`, `function`, `NULL`, `NA`, `NaN`, `Inf`, `TRUE`, and `FALSE`.

```
for (x in y) {
     .....
     .....
}
```

`y` may be a vector of any mode, a factor (in which case the numerical coding will be used), a matrix (treated as a vector), a data frame (`x` will be substituted by the different columns of `y`), or a list (`x` will be substituted by the different elements of `y`).

Two commands may be useful here: `next` stops the current iteration and moves to the next value of `x`, and `break` aborts the loop. They are usually combined with an `if` statement which takes a single logical value as argument, for example:

```
for (i in 1:10) {
     if (x[i] < 0) break
     .....
}
```

2.7.2 *Apply*-Like Functions

In many situations, there is an easier and more efficient alternative to the use of loops and control statements: the *apply*-like functions. `apply` applies a function to all columns and / or rows of a matrix or a data frame. Its syntax is:

```
apply(X, MARGIN, FUN, ...)
```

where `X` is a matrix or a data frame; the second argument indicates whether to apply the function on the rows (1), the columns (2), or both (`c(1, 2)`); `FUN` is the function to be used; and `...` any argument that may be needed for `FUN`.

`lapply` does the same as `apply` but on different elements of a list. Its syntax is:

```
lapply(x, FUN, ...)
```

This function returns a list. `sapply` has nearly the same action as `lapply` but it returns its results as a more friendly way as a vector or a matrix with rownames and colnames.

`tapply` acts on a vector and applies a function on subsets defined by an additional argument `INDEX`:

```
tapply(X, INDEX, FUN = NULL, ...)
```

Typically, `INDEX` defines groups, and the function `FUN` is applied to each group. By default, the indices of the groups defined by `INDEX` are returned.

Finally, `replicate` replicates a command a given number of times, returning the results as a vector, a matrix, or a list; for example,

```
> replicate(5, rnorm(1))
[1] -1.424699824  0.695066367  0.958153028  0.002594864
[5] -0.879007194
> replicate(4, rnorm(3))
           [,1]       [,2]       [,3]        [,4]
[1,]  0.7743082 -0.7689951 -0.4332675  1.58177859
[2,] -0.7495421 -0.5846179 -1.0581448  0.03818309
[3,]  0.1632760  0.8818927  0.6218508 -1.37648467
```

2.8 Exercises

1. Start R and print the current working directory. Suppose you want to read some data in three different files located in three different directories on your computer: find two ways to do this.

2. Create a matrix with three columns and 1000 rows where each column contains a random variable that follows a Poisson distribution with rates 1, 5, and 10, respectively (see `?Poisson` for how to generate random Poisson values). Find two ways to compute the means of each column of this matrix.

3. Create a vector of 10 random normal values using the three following methods.
   (a) Create and concatenate successively the 10 random values with `c`.
   (b) Create a numeric vector of length 10 and change its values successively.
   (c) Use the most direct method.

   Compare the timings of these three methods (see `?system.time`) and explain the differences.
   Repeat this exercise with 10,000 values.

4. Create the following file:

```
Mus_musculus                    10
Homo_sapiens                 70000
Balaenoptera_musculus    120000000
```

   (a) Read this file with `read.table` using the default options. Look at the structure of the data frame and explain what happened. What option should have been used?
   (b) From this file, create a data structure with the numeric values that you could then index with the species names, for example,

```
> x["Mus_musculus"]
[1] 10
```

   Find two ways to do this, and explain the differences in the final result.

5. Create these two vectors (source: [5]):

```
Archaea <- c("Crenarchaea", "Euryarchaea")
Bacteria <- c("Cyanobacteria", "Spirochaetes",
              "Acidobacteria")
```

   (a) Create a list named `TreeOfLife` so that we can do `TreeOfLife$Archaea` to print the corresponding group.
   (b) Update `TreeOfLife` by adding the following vector:

```
Eukaryotes <- c("Alveolates", "Cercozoa", "Plants",
                "Opisthokonts")
```

   It should appear at the same level as `Archaea` and `Bacteria`.
   (c) Update `Archaea` by adding `"Actinobacteria"`.
   (d) Print all the lowest-level taxa.

3

Phylogenetic Data in R

This chapter details how phylogenetic data are handled in R. The issues discussed here will interest all users. Issues relative to implementation and programming are discussed in Chapter 7.

3.1 Phylogenetic Data as R Objects

One strength of R is the flexibility of its data structures. In most phylogenetic programs, the data structures are completely opaque to the user. This is because complex data structures in low-level languages (such as C or C++) need a lot of programming work. This is not the case in R where the *list* data structure provides an efficient and flexible way to build complex data structures using any kind of element. For a tree coded with a list, the critical advantage is that the user can easily access its components, and manipulate or analyze them with R's functions and operators.

As a simple example, consider a tree read in R with `ape`: this will be stored in R as an object of class `"phylo"`. If this object is named `tr`, then its branch lengths will be accessed simply with `tr$edge.length`. Any subsequent analysis can be conducted with the usual R functions; as illustrations, the following commands will compute the mean, some summary statistics, plot a frequency histogram, and finally copy these branch lengths into an object named `x`.

```
mean(tr$edge.length)
summary(tr$edge.length)
hist(tr$edge.length)
x <- tr$edge.length
```

Trees can be coded in different ways in R which reflects the choices of the authors who designed these different classes. The class of an object is the attribute that signs its particularities. Some functions treat objects differently with respect to their class (Section 2.6).

ape uses a class called "phylo" to describe phylogenetic trees. The principle of its design is to store in different elements a description of its hierarchical structure, the names of the taxa, the branch lengths, and other information that may be necessary. The structure of an object of class "phylo" is detailed below. ape has another class called "matching", but its use is restricted to a few situations; it is described below.

ade4 uses another class called "phylog". It has the same goal as "phylo" but its design is radically different. An object of class "phylog" stores more information than the object of class "phylo" representing the same phylogeny, and thus it requires more memory. The class "phylog" is outlined below. apTreeshape uses a class called "treeshape" that codes dichotomous trees with no branch lengths.

We may recall that in R, all actions are done on objects stored in the active memory of the computer. Consequently, the classes described above are not designed to be new tree file formats, but rather to handle and analyze phylogenetic data efficiently.

The package stats has two classes worth mentioning here: "hclust" and "dendrogram". These classes are designed to code hierarchical clusters, and thus contain less information than the two classes described above (they may be appropriate to code ultrametric trees). However, because objects of class "hclust" and "dendrogram" are produced by clustering analyses in R, it may be useful to convert them in objects of class "phylo" which is what can be done by some functions as shown in Section 3.4.

3.1.1 The Class "phylo" (ape)

An object of class "phylo" is a list with the following components.

`edge` a two-column matrix where each row represents a branch (or edge) of the tree; the nodes and the tips are symbolized with numbers; the nodes are represented with negative numbers (the root being "-1"), and the tips are represented with positive numbers. For each row, the first column gives the ancestor. This representation allows an easy manipulation of the tree.

`edge.length` (optional) a numeric vector giving the lengths of the branches given by `edge`.

`tip.label` a vector of mode character giving the names of the tips; the order of the names in this vector corresponds to the (positive) numbers in edge.

`node.label` (optional) a vector of mode character giving the names of the nodes.

`root.edge` (optional) a numeric value giving the length of the branch at the root if it exists.

A class in R can be easily extended to include other elements, providing the names already defined are not reused. For instance, a "phylo" object could include a numeric vector `tip.date` giving the dates of the tips if they are not all contemporary (e.g., for viruses); this will not change the way other elements are accessed or modified. Another potential extension is to code networks or reticulograms because this would require simply adding the appropriate rows in the matrix `edge`.

The class `"phylo"` is a somehow minimalist representation of a phylogenetic tree. Other information that may be needed in some analyses (e.g., branching times, number of descendants for each node, etc.) must be computed by the functions that need them.

3.1.2 The Class `"phylog"` (ade4)

The class `"phylog"` takes a different approach than the `"phylo"` one: in addition to the basic structure of the tree, other information is stored. This has the advantage that some computations are faster, but the overhead is that more memory is needed to store a `"phylog"` object than a `"phylo"` one.

A `"phylog"` object is a list with 20 elements. The structure of the tree is stored in three of them: `tre` is a character string representing the tree in Newick format without the branch lengths, `leaves` is a named numeric vector where the values are the terminal branch lengths and the names are the tip labels, and `nodes` is similar to `leaves` but for the internal branches. The 17 other elements store various information which is needed by some functions in `ade4` (the details may be found on the help pages of this package, e.g., `?phylog`).

3.1.3 The Class `"matching"` (ape)

Matchings have been introduced by Diaconis and Holmes [25] as a representation of binary phylogenetic trees. The idea is to assign to each tip and node a positive number, and then to represent the topology as a series of pairs of these numbers that are siblings (the matchings). Interestingly, if some conventions are given, this results in a unique representation between a given tree and a given matching [25]. An object of class `"matching"` is a list with the following components.

`matching` a three-column numeric matrix where the first two columns represent the sibling pairs (the matching), and the third one the corresponding ancestor.

`edge.length` (optional) a numeric vector representing the branch lengths where the ith element is the length of the branch below the element numbered i in `matching`.

`tip.label` (optional) a character vector giving the tip labels where the ith element is the label of the tip numbered i in `matching`.

`node.label` (optional) a character vector giving the node labels in the same order as in `matching` (i.e., the ith element is the label of the node numbered $i + n$ in `matching`, with n the number of tips).

An object of class `"matching"` is not a matching in Diaconis and Holmes's [25] sense because it includes extra information. The latter can be printed from the former, say `x`, with `x$matching[, 1:2]`.

The class `"matching"` is used essentially in the estimation of phylogenies because this is an efficient representation for binary trees (Chapter 5).

3.1.4 The Class `"treeshape"` (apTreeshape)

The class `"treeshape"` is derived from the `"hclust"` one. An object of this class is a list with two elements:

`merge` a two-column numeric matrix where each row represents a pairing: a negative number represents a tip, and a positive number represents a group of tips as identified by the line number of this matrix. For instance, a row with `(-8, 1)` means that the eighth tip is paired with the group of tips defined by the first row of this matrix.
`names` (optional) a vector of mode character giving the names of the tips.

An object of class `"treeshape"` can be built with the function `treeshape` which takes as arguments these two elements.

3.2 Reading Phylogenetic Data

3.2.1 Phylogenies

Treelike data structures are very common in computer science, and there are many ways to store them in files. Fortunately, biologists, systematists, and phylogeneticists seem to agree on the use of a single data format for trees: the nested parentheses format, known as the Newick or New Hampshire format.[1] This format has many advantages: it is flexible, can be interpreted directly by humans (if not too long), has a close link with the hierarchical nature of evolutionary relationships, and can store large trees using little resources on a computer disk.

A common extension of the Newick format is the NEXUS format which can also include other data (usually matrices of species characters), and system commands such as calls to other programs [95].

`ape` has two functions to read trees in Newick and NEXUS formats: `read.tree` and `read.nexus`. Both functions have a file name (given as a character string or a variable of mode character) as main argument:

```
tr <- read.tree("treefile.tre")
trx <- read.nexus("treefile.nex")
```

[1] http://evolution.genetics.washington.edu/phylip/newicktree.html.

These functions ignore all white spaces and new lines in the tree file. The latter may contain several trees that are all read: the returned object is of class `c("multi.tree", "phylo")`, and is a list of objects of class `"phylo"`.

If no file name is given, `read.tree` reads the tree in Newick format from the standard input, so that the user can type the parenthetic tree directly on the keyboard (the input is terminated by a blank line). For instance, if we just type `tr <- read.tree()`, R then prompts the user to enter the tree (this can be copied/pasted from a text file). Each line of text is numbered `1:`, `2:`, and so on.

```
> tr <- read.tree()
1: (a:1,b:1);
2:
> ls()
[1] "tr"
```

Alternatively, it is possible to store the Newick tree in a variable of mode character and then use the option `text`:

```
> a <- "(a:1,b:1);"
> tr <- read.tree(text = a)
```

Both `read.tree` and `read.nexus` create an object of class `"phylo"`. Additionally, `read.nexus` keeps track of the original file in an attribute named `origin`.

ade4 has the function `newick2phylog` that creates an object of class `"phylog"` from a Newick tree stored in a character variable.

```
> b <- "((a:1,b:1):1,c:2);"
> trg <- newick2phylog(b)
```

The Newick tree can be read in a file using the function `scan` with the appropriate option.

```
> trh <- newick2phylog(scan("treefile.tre", what = ""))
```

Note that `newick2phylog` cannot read starlike trees, whereas `read.tree` cannot read trees only specified as a “skeleton” made of parentheses and commas.

```
> trg <- newick2phylog("((((,,),,(,)),),(,));")
```

However, in that case arbitrary values of one are given to the branch lengths, as well as “Ext1”, “Ext2”, . . . as tip labels.[2] Both functions can read a tree with no branch lengths such as `"((a,b),c);"`.

[2] `newick2phylog` also gives arbitrary labels to the nodes if they are not in the Newick tree: “I1”, “I2”, and so on.

3.2.2 Reading Internet Tree Databases

apTreeshape can read trees from the PANDIT[3] and TreeBASE[4] Internet databases with the functions `pandit` and `treebase`, respectively. These functions require knowledge of the numbers of the tree in their respective databases: the trees are then returned in R as objects of class `"phylo"` (the default) or `"treeshape"` if the option `class = "treeshape"` is used. These two functions are thus useful for reading trees for further analyses in R. As a simple example, we can read the second tree in Pandit, and plot it directly (Fig. 3.1):

```
  plot(pandit(2), font = 1)
```

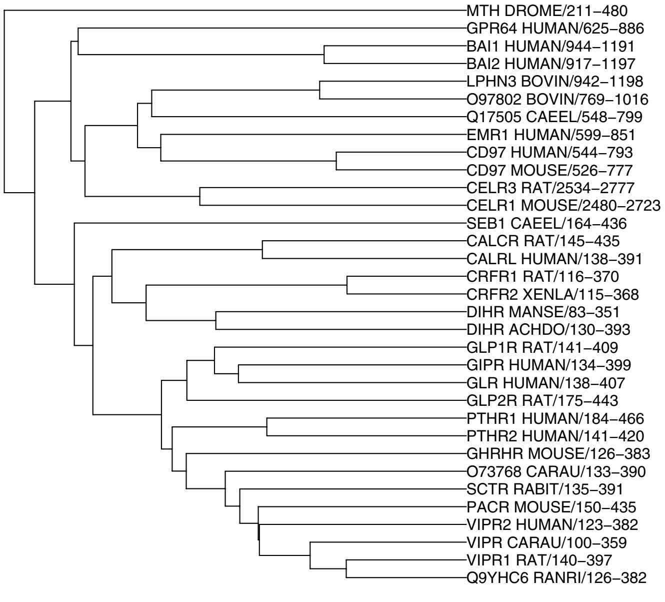


Fig. 3.1. The tree #2 in the Pandit database

3.2.3 Molecular Sequences

DNA sequences can be read with the ape function `read.dna` which reads files in FASTA, interleaved, or sequential format (these formats are described in the help page of `read.dna`). The function `read.GenBank` can read sequences

[3] http://www.ebi.ac.uk/goldman-srv/pandit/.
[4] http://www.treebase.org/.

in the GenBank databases via the Internet: its main argument is a vector of mode character giving the accession numbers of the nucleotide sequences. These accession numbers are used, by default, as names for the individual sequences. If the option `species.names = TRUE` is used, which is the default, then the species names (as read in the "ORGANISM" field in the GenBank data) are returned in an attribute called `"species"`.

These two functions return a list of vectors of single characters giving the nucleotide at each position of the sequence, thus, for instance, the twentieth nucleotide of the second sequence will be accessed with `x[[2]][20]`. All tricks of R's indexing systems (p. 12) can be used here.

`seqinr` has more flexibility than `ape` for reading molecular sequences (proteins and DNA). Two functions can read sequences stored in local files. `read.fasta` reads sequences in FASTA format. It has two arguments: `File` to specify the name of the data file, and `seqtype` to specify the type of the sequence which is either `"DNA"` (the default) or `"AA"` (for proteins). As with `read.dna`, `read.fasta` returns a list of sequences but there are a few additional attributes including a class `"SeqFastadna"` or `"SeqFastaAA"` depending on the type of the sequence.

`read.alignment` reads aligned sequences. There are two arguments: `File` and `format` which can be `"mase"`, `"clustal"`, `"phylip"`, `"fasta"`, or `"msf"`. If `format = "phylip"`, the function detects whether the format is sequential or interleaved. The sequences are stored in a different way than `read.fasta` and `read.dna` do: each sequence is stored as a single character string, whereas for the latter each sequence is a vector of strings made of single characters (each being a position in the sequence). The data returned by `read.alignment` are of class `"alignment"`.

`seqinr` has an elaborate mechanism for retrieving sequences from molecular databanks. This works through the ACNUC repository.[5] The databanks available are listed in R with the function `choosebank` used without argument (this works only if the computer is connected to the Internet):

```
> choosebank()
 [1] "genbank"    "embl"       "emblwgs"    "swissprot"
 [5] "ensembl"    "emglib"     "nrsub"      "nbrf"
 [9] "hobacnucl"  "hobacprot"  "hovernucl"  "hoverprot"
[13] "hogennucl"  "hogenprot"  "hoverclnu"  "hoverclpr"
[17] "HAMAPnucl"  "HAMAPprot"  "hoppsigen"  "nurebnucl"
[21] "nurebprot"  "taxobacgen" "greview"
```

These databanks are mirrored on the PBIL server in Lyon. The user selects one of these banks with the same function:

```
> s <- choosebank("genbank")
```

[5] http://pbil.univ-lyon1.fr.

It is then possible to query the bank for the available sequences. For instance, to get the list of the sequences of the bird genus *Ramphocelus* [59]:

```
> query(s$socket, "rampho", "sp=Ramphocelus@")


$socket:                    description                                  class
"->pbil.univ-lyon1.fr:5558"                                  "socket"
                       mode                                      text
                       "a+"                                    "text"
                     opened                                  can read
                   "opened"                                     "yes"
                  can write
                      "yes"

$banque: genbank
$call: query(socket = s$socket, listname = "rampho",
             query = "sp=Ramphocelus@")
$name: [1] "rampho"

  list length mode      content
1 $req 20     character sequences
```

This command returns an object named `"rampho"` which lists the sequences meeting the selection criteria.[6] The special character `"@"` meets any set of characters (see `?query` for the details of the syntax of this function). The result displayed by `query` shows that 20 sequences were found. `"rampho"` is a list with the accession numbers and the connection details (server name, port number, etc.) to retrieve the sequences effectively:

```
> rampho$req[[1]]
[1] "AF310048"
attr(,"class")
[1] "SeqAcnucWeb"
attr(,"socket")
                description                                     class
"->pbil.univ-lyon1.fr:5558"                                  "socket"
                       mode                                      text
                       "a+"                                    "text"
                     opened                                  can read
                   "opened"                                     "yes"
                  can write
                      "yes"
```

[6] The syntax is unusual in R where objects are often created with the assign operator `<-`.

The sequences are then extracted from ACNUC with the generic function `getSequence`:

```
> x <- getSequence(rampho$req[[1]])
> length(x)
[1] 921
> x[1:20]
 [1] "g" "g" "a" "t" "c" "c" "t" "t" "a" "c" "t" "a" "g"
[14] "g" "c" "c" "t" "a" "t" "g"
```

As a comparison, the same sequence can be extracted with the `ape` function `read.GenBank`:

```
> y <- read.GenBank("AF310048")
> length(y[[1]])
[1] 921
> attr(y, "species")
[1] "Ramphocelus_carbo"
> identical(x, y[[1]])
[1] TRUE
```

3.3 Writing Data

We have seen that R works on data stored in the active memory of the computer. It is obviously necessary to be able to write data, at least for two reasons. The user may want at any time to save all the objects present in memory to prevent data loss from a computer crash, or because he wants to quit R and continue his analyses later. The other reason is that the user wants to analyze some data stored in R with other programs which in most cases need to read the data from files (unless there is a link between the software and R; see Chapter 7).

Any kind of data type in R can be saved in a binary file using the `save` function; the objects to be saved are simply listed as arguments separated by commas.

```
save(x, y, tr, file = "mydata.RData")
```

The ".RData" suffix is a convention and is associated with R on some operating systems (e.g., Windows). The binary files created this way are portable across platforms. The command `save.image()` (used without options) is a short-cut to save all objects in memory (the *workspace* is R's jargon) in a file called '.RData'. It is eventually called by R when the user quits the system and chooses to save an image of the workspace.

`ape` has several functions that write trees and DNA sequences in formats suitable for other systems. `write.tree` writes a tree in Newick format. It takes as main argument a `"phylo"` object. By default the Newick tree is returned as a character string, and thus can be used as a variable itself:

```
> tr <- read.tree(text = "(a:1,b:1);")
> write.tree(tr)
[1] "(a:1,b:1);"
> x <- write.tree(tr)
> x
[1] "(a:1,b:1);"
```

To save the tree in a file, one needs to use the option `file`:

```
> write.tree(tr, file = "treefile.tre")
```

The option `append` (`FALSE` by default) controls whether to delete any previous data in the file. For larger trees, the character string is split with line breaks. This behavior can be avoided with the option `multi.line = FALSE`.

One or several `"phylo"` objects can also be written in a NEXUS file using `write.nexus`. This function behaves similarly to `write.tree` in that it prints by default the tree on the console (but this cannot be reused as a variable).

```
> write.nexus(tr)
#NEXUS
[R-package APE, Mon Dec 20 11:18:23 2004]

BEGIN TAXA;
        DIMENSIONS NTAX = 2;
        TAXLABELS
                a
                b
        ;
END;
BEGIN TREES;
        TRANSLATE
                1       a,
                2       b
        ;
        TREE * UNTITLED = [&R] (1:1,2:1);
END;
```

The options of `write.nexus` are `translate` (default `TRUE`) which replaces the tip labels in the parenthetic representation with numbers, and `original.data` (default `TRUE`) to write the original data in the NEXUS file (in agreement with the NEXUS standard [95]). If several trees are written, they must have the same tip labels, and must be given either as a series, or as a list:

```
> write.nexus(tr1, tr2, tr3, file = "treefile.nex")
> L <- list(tr1, tr2, tr3)
> write.nexus(L, file = "treefile.nex")
```

DNA sequences are written into files with `write.dna`. Its option `format` can take the values `"interleaved"` (the default), `"sequential"`, or `"fasta"`. There are several options to customize the formatting of the output sequences (see `?write.dna` for details).

3.4 Manipulating Data

Manipulating phylogenetic trees is difficult because of the complexity of such data structures. This may be one of the reasons why so few programs offer this possibility. Another reason may be that once a phylogeny has been obtained, sometimes after a long process of various analyses, the user is not willing to change it.

There are good reasons for making such manipulations possible, though, for instance if some comparative analyses are to be done (see Section 6.1). It is also sometimes needed to "arrange" a tree before plotting it, such as rotating a branch or dropping a tip. Other less trivial manipulations include extracting branch lengths, computing branching times or coalescent intervals, (un)rooting a tree, testing whether two trees are identical, and so on.

3.4.1 Basic Tree Manipulation

`ape` has several functions to manipulate `"phylo"` objects. They are listed below. In the examples, the trees are written as Newick strings for convenience; the results could also be visualized with `plot` instead of `write.tree`.

`drop.tip` removes one or several tips from a tree. The former are specified either by their labels or their positions (indices) in the vector `tip.label`. By default, the terminal branches and the corresponding internal ones are removed. This has the effect of keeping the tree ultrametric in the case it was beforehand. This behavior can be altered by setting the option `trim.internal = FALSE`.

```
> tr <- read.tree(text = "((a:1,b:1):1,(c:1,d:1):1);")
> write.tree(drop.tip(tr, c("a", "b")))
[1] "(c:1,d:1);"
> write.tree(drop.tip(tr, 1:2)) # same as above
[1] "(c:1,d:1);"
> write.tree(drop.tip(tr, 1:2, trim.internal = FALSE))
[1] "(NA:1,(c:1,d:1):1);"
```

`bind.tree` is used to build a tree from two trees. The arguments are two `"phylo"` objects. By default, the second tree is bound on the root of the first one; a different node may be specified using the option `node`. If the second tree has a `root.edge` this will be used. Thus the binding of two binary (dichotomous) trees will result in a trichotomy or a tetrachotomy (if there is no root edge) in the returned tree. This may be avoided by using the option `branch` instead of `node`. The syntax is nearly the same: the distinction being that the second tree is bound below the node given in `branch`. The further argument `position` specifies where on the branch the tree is to be bound.

```
> t1 <- read.tree(text = "(a:1,b:1):1;")
> t2 <- read.tree(text = "(c:1,d:1):1;")
> write.tree(bind.tree(t1, t2))
[1] "(a:1,b:1,(c:1,d:1):1):1;"
> write.tree(bind.tree(t1, t2, branch = -1, position = 1))
[1] "((a:1,b:1):1,(c:1,d:1):1):0;
```

`rotate` rotates the internal branch below the most recent common ancestor of a monophyletic group given by the argument `group`. The resulting tree is exactly equivalent to the original one. This function is convenient when plotting a tree if it is needed to change the order of the tips on the plot. On the other hand, the modification is not apparent when writing the tree in Newick format because the tips are written according to the numbers in the `"phylo"` object.

`compute.brlen` modifies or creates the branch lengths of a tree with respect to the second argument, `method`, which may be one of the following.
- A character string specifying the method to be used (e.g., `"Grafen"`).
- An R function used to generate random branch lengths (e.g., `runif`).
- One or several numeric values (recycled if necessary).

For instance, if we want to set all branch lengths equal to one [48, 54]: `tr <- compute.brlen(tr, 1)`. This is likely to be useful in comparative analyses when a phylogeny with no branch lengths is available.

3.4.2 Rooted *Versus* Unrooted Trees

The Newick parenthetic format can represent both rooted and unrooted trees. In the latter, all nodes have at least three connecting branches. Thus, in the Newick representation of an unrooted tree, it is necessary that the basal grouping has (at least) three sibling groups:

((...),(...),(...));

Such a tree read in R with `read.tree` would result in an object of class `"phylo"` whose root has three descendants. In this case, the root has no biological interpretation: it does not represent a common ancestor of all tips.

The function `is.rooted` tests whether an object of class `"phylo"` represents a rooted tree. It returns `TRUE` if either only two branches connect to the root, or if there is a `root.edge` element.

```
> ta <- read.tree(text = "(a,b,c);")
> tb <- read.tree(text = "(a,b,c):1;")
> tc <- read.tree(text = "((a,b),c);")
> is.rooted(ta)
[1] FALSE
> is.rooted(tb)
[1] TRUE
> is.rooted(tc)
[1] TRUE
```

The presence of a zero `root.edge` allows us to have a rooted tree with a basal trichotomy:

```
> td <- read.tree(text = "(a,b,c):0;")
> is.rooted(td)
[1] TRUE
```

Both objects `ta` and `td` are graphically similar; the difference between them is that the root of `td` can be interpreted biologically as a common ancestor of a, b, and c.

The function `root` reroots a tree given an outgroup, made of one or several tips; as the argument `outgroup`. If the tree is rooted, it is unrooted before being rerooted, so that if `outgroup` is already an outgroup, then the returned tree is not the same as the original one. The specified outgroup must be monophyletic, otherwise the operation fails and an error message is printed.

The function `unroot` transforms a rooted tree into its unrooted counterpart. If the tree is already unrooted, it is returned unchanged.

3.4.3 Dichotomous *Versus* Multichotomous Trees

The Newick format represents multichotomies by having more than two sibling groups:

(A,B,C);

This is represented explicitly in the class `"phylo"` by letting a node have several descendants in the element `edge`, for instance:

```
...
-2 1
-2 2
-2 3
...
```

where 1, 2, 3 would be the numbers of the tips A, B, C.

As shown in the next chapters, some methods deal only with dichotomous (i.e., binary) trees, thus it may be useful to resolve multichotomies into dichotomies with internal branches of length zero. On the other hand, when a dichotomous tree has internal branches of length zero it may be needed to collapse them in a multichotomy. These two operations may be performed with the functions `multi2di` and `di2multi`, respectively. They both take an object of class `"phylo"` as main argument; `di2multi` has a second argument `tol` that specifies the tolerance to consider branch lengths significantly greater than zero (10^{-8} by default).

There are several ways to solve a multichotomy resulting in different topologies. The number of possibilities grows very fast with the number of branches, n, involved in the multichotomy: it is given by $n!/2$ (`factorial(n)/2` in R). For only three possibilities with $n = 3$, there are 60 with $n = 5$, and 1,814,400 with $n = 10$. `multi2di` has a second argument, `random`, which specifies whether to solve the multichotomies in a random order (the default), or in an arbitrary order if `random = FALSE`. Repeating the use of `multi2di` on a tree with the default option will likely yield different topologies. Specifying `random = FALSE` may be preferred if the operation is repeated and it is necessary always to have the same topology.

apTreeshape has a different mechanism to solve multichotomies randomly. It is used when reading trees from databases (Section 3.2.2), or when converting trees of class `"phylo"` with multichotomies (Section 3.4.5). The functions `pandit`, `treebase`, and `as.treeshape` have the option `model` that can take the following values: `"biased"`, `"pda"`, or `"yule"`. This specifies the model used to resolve the multichotomies. These models are explained in Section 3.5.

In a rooted dichotomous tree the number of tips is equal to the number of nodes minus one, whereas this is minus two for an unrooted tree (because the root node has been removed). The function `is.binary.tree` tests whether a tree, either rooted or unrooted, is dichotomous, and returns a logical value.

3.4.4 Summarizing and Comparing Trees

There is a `summary` method for `"phylo"` objects. This function prints a brief summary of the tree including the numbers of nodes and tips.

`is.ultrametric` tests if a tree is ultrametric (all tips equally distant from the root), and returns a logical value. This is done taking the numerical precision of the computer into account.

`balance` returns, for a fully binary tree, the number of descendants of both sister-lineages from each node (see Section 6.3.5 for analyses of tree shape).

Once the branch lengths of a `"phylo"` object have been extracted as shown above, any computation can be done on them. There are special functions to perform some particular operations. `branching.times` returns, for an ultrametric tree, the distances from the nodes to the tips using its branch lengths. `coalescent.intervals` computes the coalescence times for an ultrametric tree and returns, in the form of a list, a summary of these computations with the number of lineages present at each interval, the lengths of the intervals, the total number of intervals, and the depth of the tree.

It is often necessary to compare two phylogenetic trees because there could be, for a given format, several representations of the same tree. This is the case with the Newick format, and also for the "phylo" class of objects. The generic function all.equal tests whether two objects are “approximately equal”. For instance, for simple numeric data the comparison is done considering the numerical precision of the computer. For "phylo" objects, only the labeled topologies are compared: if both represention are the same, TRUE is returned, otherwise a summary of the comparison is printed. Here is an example with a simple case of two representations for the same rooted tree:

```
> t1 <- read.tree(text = "((a:1,b:1):1,c:2);")
> t2 <- read.tree(text = "(c:2,(a:1,b:1):1);")
> all.equal(t1, t2)
[1] TRUE
```

If both trees have similar labeled topologies, their branch lengths can be compared with the same generic function:

```
> all.equal(t1$edge.length, t2$edge.length)
[1] "Mean relative  difference: 0.6666667"
> all.equal(sort(t1$edge.length), sort(t2$edge.length))
[1] TRUE
```

Two objects of class "treeshape" can also be compared with all.equal. Because all.equal does not always return a logical value, it should not be used in programming a conditional execution. The following should be used instead:

```
identical(all.equal(t1, t2), TRUE)
```

3.4.5 Converting Objects

We have seen that a tree may be coded in different ways in R that correspond to different classes of objects. It is obviously useful to be able to convert among these different classes because some operations can be done on some classes but not others. Table 3.1 gives details on how to convert among the six classes discussed here.

The entries marked *nd* in this table indicate that the conversion cannot be done directly, and it must be done in two (or more) steps. For instance, to convert a "phylo" object in a "dendrogram" one, we will do:

```
as.dendrogram(as.hclust(x))
```

There is currently no way to convert a "dendrogram" object to another class. This class has been recently introduced in R and is still under development.

Table 3.1. Conversion among the different classes of tree objects in R (`x` is the object of the original class). *nd* means there is no direct way to do the conversion, and it must be done via another class

From	To	Command
phylo	phylog	`newick2phylog(write.tree(x))`[a]
	matching	`as.matching(x)`
	treeshape	`as.treeshape(x)`
cindas.treeshape	hclust	`as.hclust(x)`
	dendrogram	*nd*
phylog	phylo	`as.phylo(x)`
	matching	*nd*
	treeshape	*nd*
	hclust	*nd*
	dendrogram	*nd*
matching	phylo	`as.phylo(x)`
	phylog	*nd*
	treeshape	*nd*
	hclust	*nd*
	dendrogram	*nd*
treeshape	phylo	`as.phylo(x)`
	phylog	*nd*
	matching	*nd*
	hclust	*nd*
	dendrogram	*nd*
hclust	phylo	`as.phylo(x)`
	phylog	`hclust2phylog(x)`
	matching	*nd*
	treeshape	*nd*
	dendrogram	`as.dendrogram(x)`

[a]It may be necessary to use the option `multi.line=FALSE`

3.4.6 Manipulating DNA Data

A DNA sequence read with `read.dna` is stored in R as a vector where each element is a single (lowercase) letter representing a nucleotide site. This allows an easy manipulation of DNA data with little programming overhead. Here are a few examples.

- Reads a sequence in FASTA format, stores it in `x`, and reverts it:

```
x <- read.dna("dnafile.fas", format = "fasta")
rev(x)
```

- Extracts the third position (assuming that the reading frame of the sequence is correct):

```
x[seq(3, length(x), by = 3)]
```

- Creates a vector z of the same sequence but with nucleotides grouped by codon (assuming that the reading frame of the sequence is correct):

```
   z <- character(length(x) %/% 3)
   for (i in 1:length(z))
     z[i] <- paste(x[(3 * i - 2):(3 * i)], collapse = "")
```

A set of sequences can be stored as a list, a matrix, or a data frame. The last two kinds of structures are appropriate only for aligned sequences because all rows must have the same number of elements. To apply the above operations on such sets of sequences, one can use the functions apply or lapply. For instance, in the case of a matrix X, the following will revert all rows:

```
apply(X, 1, rev)
```

and for a list:

```
lapply(X, rev)
```

For more complex operations, one may first create a function that encloses all the needed commands, and then use the appropriate apply-like function:

```
foo <- function(x)
{
    z <- character(length(x) %/% 3)
    for (i in 1:length(z))
      z[i] <- paste(x[(3 * i - 2):(3 * i)], collapse = "")
    z # needed to return the vector
}
lapply(X, foo)
```

seqinr has more sophisticated functions for manipulating molecular sequences. invers reverts a sequence in the same way as rev above. comp returns the complement of a DNA sequence:

```
> x <- scan(what = "")
1: a c g t g g t c a t
11:
Read 10 items
> x
 [1] "a" "c" "g" "t" "g" "g" "t" "c" "a" "t"
> comp(x)
 [1] "t" "g" "c" "a" "c" "c" "a" "g" "t" "a"
```

The functions c2s and s2c transform a vector of single characters into a string, and vice versa:

```
> c2s(x)
[1] "acgtggtcat"
> s2c(c2s(x))
 [1] "a" "c" "g" "t" "g" "g" "t" "c" "a" "t"
```

`splitseq` splits a sequence into portions with respect to two options: `frame` specifying how many sites to skip before starting to read the sequence (default is 0), and `word` giving the length of the portions (default is 3, i.e., a codon for a DNA sequence):

```
> splitseq(x)
[1] "acg" "tgg" "tca"
> splitseq(x, frame = 1)
[1] "cgt" "ggt" "cat"
> splitseq(x, word = 5)
[1] "acgtg" "gtcat"
```

`translate` translates a DNA sequence into an amino acid (AA) one. The option `frame` may be used as above. Two other options are `sens`, which can be `"F"` (forward, the default) or `"R"` (reverse) specifying the direction of the translation, and `numcode` which takes a numeric value specifying the genetic code to be used (by default the universal code is used):

```
> translate(x)
[1] "T" "W" "S"
> translate(x, frame = 1)
[1] "R" "G" "H"
> translate(x, frame = 2)
[1] "V" "V"
> translate(x, frame = 3)
[1] "W" "S"
> translate(x, frame = 4)
[1] "G" "H"
```

The functions `aaa` and `a` convert AA sequences from the one-letter coding to the three-letter one, and vice versa:

```
> aaa(translate(x))
[1] "Thr" "Trp" "Ser"
> a(aaa(translate(x)))
[1] "T" "W" "S"
```

`ape` has a few functions for summarizing information from a set of DNA sequences.

- `base.freq` computes the proportions of each of the four bases; the results are returned as a table (i.e., a table with names `"A"`, `"C"`, `"G"`, and `"T"`).

- `GC.content` is based on the previous function, and computes the proportion of guanine and cytosine; a single numeric value is returned.
- `seg.sites` returns the indices of the segregating sites, that is, the sites that are polymorphic.

`seqinr` has several functions for summarizing molecular sequences. `count` computes the frequencies of all possible combinations of n nucleotides, where n is specified with the argument `word` (there is also an option `frame` used in the same way as above):

```
> count(x, word = 1)

a c g t
2 2 3 3
> count(x, word = 2)

aa ac ag at ca cc cg ct ga gc gg gt ta tc tg tt
 0  1  0  1  1  0  1  0  0  0  1  2  0  1  1  0
> count(x, word = 3)

aaa aac aag aat aca acc acg act aga agc agg agt ata atc atg
  0   0   0   0   0   0   1   0   0   0   0   0   0   0   0
att caa cac cag cat cca ccc ccg cct cga cgc cgg cgt cta ctc
  0   0   0   0   1   0   0   0   0   0   0   0   1   0   0
ctg ctt gaa gac gag gat gca gcc gcg gct gga ggc ggg ggt gta
  0   0   0   0   0   0   0   0   0   0   0   0   0   1   0
gtc gtg gtt taa tac tag tat tca tcc tcg tct tga tgc tgg tgt
  1   1   0   0   0   0   0   1   0   0   0   0   0   1   0
tta ttc ttg ttt
  0   0   0   0
```

The three functions `GC`, `GC2`, and `GC3` compute the proportion of guanine and cytosine over the whole sequence, over the second positions, and over the third ones, respectively:

```
> GC(x)
[1] 0.5
> GC2(x)
[1] 0.9999
> GC3(x)
[1] 0.6666
```

There are two summary methods for the classes `"SeqFastaAA"` and `"SeqFastadna"`: they print a summary of the frequencies of the different amino acids or bases, and other information such as the lengths of the sequences.

`AAstat` has the same effect as `summary.SeqFastaAA`, but additionally a graph is plotted of the position of the different categories of AAs. For instance, taking a protein sequence distributed with `seqinr` (Fig. 3.2):

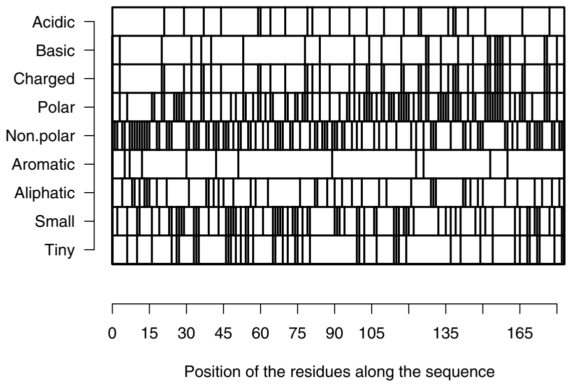


Fig. 3.2. Plot of the distribution of amino acid categories along the sequence of a protein

```
> ss <- read.fasta(system.file("sequences/seqAA.fasta",
+                                    package = "seqinr"),
+                        seqtype = "AA")
> AAstat(ss[[1]])
$Compo

 *  A  C  D  E  F  G  H  I  K  L  M  N  P  Q  R  S  T  V  W
 1  8  6  6 18  6  8  1  9 14 29  5  7 10  9 13 16  7  6  3
 Y
 1
....
```

3.5 Generating Random Trees

`ape` has two functions to generate random trees under assumptions. `rtree` generates a tree by random splitting; its interface is:

```
rtree(n, rooted = TRUE, tip.label = NULL, br = runif, ...)
```

where `n` specifies the number of tips. The tree is rooted by default. If `tip.label` is left `NULL`, the labels `"t1"`, `"t2"`, . . . , are given to the tips. `br` specifies the function to generate random branch lengths: further arguments for this function are given in place of the "dot-dot-dot" (`...`). By default, a uniform distribution between 0 and 1 is used. Use `br = NULL` for a tree with no branch length.

`rcoal` generates a "coalescent" tree by random clustering of tips; its interface is:

```
rcoal(n, tip.label = NULL, br = rexp, ...)
```

where the options are similar to `rtree`. Note that the default for `br` is the exponential distribution: this is used to generate node heights (branch lengths are computed from these heights).

Both `rtree` and `rcoal` generate a single tree: they must be called repeatedly to generate a sample of random trees.

apTreeshape has the function `rtreeshape` that generates tree topologies under various models. Its interface is:

```
rtreeshape(n, tip.number, p = 0.3, model = "", FUN = "")
```

where `n` is the number of generated trees (unlike the above two functions), `tip.number` is the number of tips, `p` is a parameter used if `model = "biased"` (see below), `model` specifies the model to be used, and `FUN` gives a function to generate trees according to Aldous's Markov branching model [2]. Either `model` or `FUN` must be specified, but not both. Note that the arguments are not recycled in R's usual way: for instance, `rtreeshape(2, c(5, 10), model = "yule")` will generate four trees (two with five tips, and two with ten).

The three models that can be specified with the argument `model` are:

- The Yule model (`model = "yule"`) where each species has the same probability of splitting in two species;
- The PDA (proportional to distinguishable arrangements) model (`model = "pda"`) where each topology is equiprobable;
- The biased model (`model = "biased"`) where a species with splitting probability r gives, if it splits, two daughter-species with splitting probability pr and $1 - pr$, respectively [83]. The value of p is given by the argument `p`.

In Aldous's [2] model, the splitting probabilities are specified through a function denoted $Q_n(i)$ which gives the probability that a clade with n tips is made of two sibling groups with i and $n - i$ tips, respectively. We specify these probabilities with the argument `Q` which is an R function of the form `Q(n, i)`. For instance, for a completely unbalanced tree we use the following:

```
Q <- function(n, i) if (i == 1) 1 else 0
rtreeshape(1, 10, FUN = Q)
```

which says that a clade of size n is certain to be made of two subclades with one and $n-1$ tips, respectively. The probabilities given in `FUN` do not need to sum to one, so that it is easy to specify a given model. An interesting model may be to have splitting probabilities proportional to the size of the clade:

```
Q <- function(n, i) if (i > 0 && i < n) n else 0
rtreeshape(1, 30, FUN = Q)
```

An example is shown in Fig. 3.3.

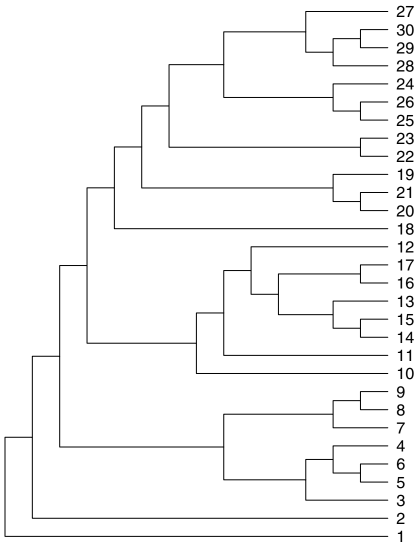


Fig. 3.3. Plot of a random tree generated with `rtreeshape`

3.6 Case Studies

The case studies show examples of workflow when using R for phylogenetic analyses. All operations are detailed and explained so readers can repeat them, and eventually adapt them to their needs. In this chapter, we consider preparing several data sets from the literature. Some of these data are analyzed further in the next chapters.

3.6.1 *Sylvia* Warblers

Böhning-Gaese et al. [10] studied the evolution of ecological niches in 26 species of warblers of the genus *Sylvia*. They also sequenced the gene of the cytochrome *b* for these species; the sequences were deposited in GenBank and have accession numbers AJ534526–AJ534549 and Z73494. We consider these molecular data as well the ecological data in their Table 1. The goal of this application is to get the sequence data from GenBank, prepare (align) them, and read the ecological data from a file.

Because the DNA data are accessible through GenBank, we get them with `read.GenBank`. We first create a vector of mode character with the accession numbers: the operation is straightforward with the function `paste`:

```
> x <- paste("AJ5345", 26:49, sep = "")
> x <- c("Z73494", x)
> x
 [1] "Z73494"   "AJ534526" "AJ534527" "AJ534528" "AJ534529"
 [6] "AJ534530" "AJ534531" "AJ534532" "AJ534533" "AJ534534"
[11] "AJ534535" "AJ534536" "AJ534537" "AJ534538" "AJ534539"
[16] "AJ534540" "AJ534541" "AJ534542" "AJ534543" "AJ534544"
[21] "AJ534545" "AJ534546" "AJ534547" "AJ534548" "AJ534549"
```

We then read the sequences. Of course, the computer must be connected to the Internet:

```
sylvia.seq <- read.GenBank(x)
```

We check that the data have been correctly downloaded by looking at the structure of the returned object:

```
> str(sylvia.seq)
List of 25
 $ Z73494  : chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534526: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534527: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534528: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534529: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534530: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534531: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534532: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534533: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534534: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534535: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534536: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534537: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534538: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534539: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534540: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534541: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534542: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534543: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534544: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534545: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534546: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534547: chr [1:1143] "a" "t" "g" "g" ...
 $ AJ534548: chr [1:1041] "g" "g" "a" "t" ...
 $ AJ534549: chr [1:1041] "g" "g" "a" "t" ...
 - attr(*, "species")= chr [1:25] "Sylvia_atricapilla_atricapilla"
    "Chamaea_fasciata" "Sylvia_nisoria" "Sylvia_layardi" ...
```

We have effectively a list with 25 sequences: 23 of them have 1143 nucleotides, and 2 have 1041. This necessitates an alignment operation with Clustal X. To do this we first write the data in a file in FASTA format:

```
write.dna(sylvia.seq, "sylviaseq.fas", format = "fasta")
```

The first three lines of the file 'sylviaseq.fas' are:

```
> Z73494
atggctctca atcttcgaaa aaaccaccct atcctaaaag tcatcaacga cgccctaatc
gacctaccaa cgccgtctaa catctctact tgatgaaact tcggctcact cctaggtctt
....
```

The alignment operation shows that there are 102 missing nucleotides in the last two sequences. The alignment made by Clustal X is saved in "Phylip" format which is actually the interleaved format of Phylip [38]. The data are read back into R using `read.dna`:

```
sylvia.seq.ali <- read.dna("sylviaseq.phy")
```

Note that we kept the original (unaligned) sequences from GenBank because they have the species names. To save some memory, we can keep them in a separate vector whose names are the accession numbers,[7] and erase the original sequences:

```
> taxa.sylvia <- attr(sylvia.seq, "species")
> names(taxa.sylvia) <- names(sylvia.seq)
> rm(sylvia.seq)
```

We then see that two of these names have to be fixed:

```
> sylvia.seq[c(1, 24)]
                          Z73494
"Sylvia_atricapilla_atricapilla"
                        AJ534548
         "Illadopsis_abyssinica"
```

Böhning-Gaese et al. [10] wrote that *Illadopsis abyssinica* had a different generic status, but they considered it as belonging to *Sylvia*: we change this accordingly for consistency. We also remove the subspecies name of the first sequence, and print all the species names:

```
> taxa.sylvia[1] <- "Sylvia_atricapilla"
> taxa.sylvia[24] <- "Sylvia_abyssinica"
> taxa.sylvia
               Z73494                AJ534526
  "Sylvia_atricapilla"     "Chamaea_fasciata"
             AJ534527                AJ534528
      "Sylvia_nisoria"       "Sylvia_layardi"
             AJ534529                AJ534530
```

[7] We show later the advantage of using this structure.

```
   "Sylvia_subcaeruleum"        "Sylvia_boehmi"
               AJ534531                AJ534532
          "Sylvia_buryi"        "Sylvia_lugens"
               AJ534533                AJ534534
   "Sylvia_leucomelaena"     "Sylvia_hortensis"
               AJ534535                AJ534536
  "Sylvia_crassirostris"       "Sylvia_curruca"
               AJ534537                AJ534538
           "Sylvia_nana"      "Sylvia_communis"
               AJ534539                AJ534540
  "Sylvia_conspicillata"   "Sylvia_deserticola"
               AJ534541                AJ534542
      "Sylvia_balearica"        "Sylvia_undata"
               AJ534543                AJ534544
     "Sylvia_cantillans" "Sylvia_melanocephala"
               AJ534545                AJ534546
       "Sylvia_mystacea"  "Sylvia_melanothorax"
               AJ534547                AJ534548
      "Sylvia_rueppelli"    "Sylvia_abyssinica"
               AJ534549
          "Sylvia_borin"
```

The ecological data are in a file 'sylvia_data.txt' whose first three lines are:

```
                   mig.dist  mig.behav  geo.range
Sylvia_abyssinica         0      resid       trop
Sylvia_atricapilla     5000      short   temptrop
....
```

We read these data simply with `read.table`, and check the returned object:

```
> sylvia.eco <- read.table("sylvia_data.txt")
> str(sylvia.eco)
'data.frame':   26 obs. of  3 variables:
 $ mig.dist : int  0 5000 7500 5900 5500 3400 2600 0 0 0 ...
 $ mig.behav: Factor w/ 3 levels "long","resid",..
 $ geo.range: Factor w/ 3 levels "temp","temptrop",..
....
```

Note that the species names are used as rownames in this data frame:

```
> rownames(sylvia.eco)
 [1] "Sylvia_abyssinica"    "Sylvia_atricapilla"
 [3] "Sylvia_borin"         "Sylvia_nisoria"
 [5] "Sylvia_curruca"       "Sylvia_hortensis"
 [7] "Sylvia_crassirostris" "Sylvia_leucomelaena"
```

```
  [9] "Sylvia_buryi"         "Sylvia_lugens"
 [11] "Sylvia_layardi"       "Sylvia_subcaeruleum"
 [13] "Sylvia_boehmi"        "Sylvia_nana"
 [15] "Sylvia_deserti"       "Sylvia_communis"
 [17] "Sylvia_conspicillata" "Sylvia_deserticola"
 [19] "Sylvia_undata"        "Sylvia_sarda"
 [21] "Sylvia_balearica"     "Sylvia_cantillans"
 [23] "Sylvia_mystacea"      "Sylvia_melanocephala"
 [25] "Sylvia_rueppelli"     "Sylvia_melanothorax"
```

The data are ready and can be saved in an R workspace before being analyzed:

```
save(sylvia.seq.ali, taxa.sylvia, sylvia.eco,
     file = "sylvia.RData")
```

3.6.2 Phylogeny of the Felidae

Johnson and O'Brien [75] studied the phylogenetic relationships of all extant species of felids and cats using sequences from two mitochondrial genes: 16S rRNA and NADH-5. For simplicity, we use only the first set of sequences. The procedure of getting and preparing these data follows the same lines as with the *Sylvia* case.

The accession numbers in GenBank range from AF006387 to AF006459 with only the odd numbers:

```
x <- paste("AF006", seq(387, 459, 2), sep = "")
felidseq16S <- read.GenBank(x)
```

The sequences are not of the same lengths (some insertions / deletions have been reported in [75]):

```
> table(unlist(lapply(felidseq16S, length)))

372 373 374 375 376
  3   9  15   9   1
> str(felidseq16S[1:5])
List of 5
 $ AF006387: chr [1:374] "t" "t" "t" "g" ...
 $ AF006389: chr [1:375] "t" "t" "t" "g" ...
 $ AF006391: chr [1:372] "c" "t" "t" "g" ...
 $ AF006393: chr [1:375] "t" "t" "t" "g" ...
 $ AF006395: chr [1:374] "t" "t" "t" "g" ...
```

We write the sequences in a file in FASTA format to align them with Clustal_X:

```
write.dna(felidseq16S, "felidseq16S.fas", format = "fasta")
```

We also save the names of the species with the accession numbers:

```
taxa.felid <- attr(felidseq16S, "species")
names(taxa.felid) <- names(felidseq16S)
```

The aligned sequences are read back in R:

```
felidseq16Sali <- read.dna("felidseq16S.phy")
```

And we may check that they have all the same length:

```
> table(unlist(lapply(felidseq16Sali, length)))

382
 37
```

In addition to the sequence data, we use data on body mass (source [143]). The first three lines from the file 'felid_bodymass.txt' are:

```
Acinonyx_jubatus            50000
Caracal_caracal           13749.9
Catopuma_badia               2500
....
```

We read this tree with `read.table`:

```
DF <- read.table("felid_bodymass.txt")
```

Because there is only one variable, it is simpler to keep it as a vector with names set as the species names:

```
> felid.body.mass <- DF$V2
> names(felid.body.mass) <- DF$V1
> felid.body.mass
        Acinonyx_jubatus          Caracal_caracal
                50000.00                 13749.90
          Catopuma_badia      Catopuma_temminckii
                 2500.00                 11500.00
....
```

We save the aligned sequences, the species names, and the body mass data for further analyses:

```
save(felidseq16Sali, taxa.felid, felid.body.mass,
     file = "felid.RData")
```

3.6.3 Snake Venom Proteome

Fry [41] made an extensive analysis of the relationships among snake venom proteins and related nontoxic proteins. We limit ourselves to a single data set: the pseutarin C of the Eastern brown snake (*Pseudonaja textilis*) and the related mammalian coagulation factor V [41, Fig. 3B]. The goal of the present application is to get the protein sequence data.

The original data come from the SWISSPROT database of protein sequences. The 22 accession numbers and the corresponding species names are stored in a file called 'venom_factorV.txt' the first three lines of which are:

```
No     species
Q9BQS7 Homo_sapiens
Q9Z0Z4 Mus_musculus
....
```

We read them with `read.table` setting `as.is = TRUE` to avoid these character strings being treated as factors, and `header = TRUE` to specify that the first line contains the names of the columns; we then display the first two rows of the data frame:

```
> venom.no <- read.table("venom_factorV.txt",
                          as.is = TRUE, header = TRUE)
> venom.no[1:2, ]
      No      species
1 Q9BQS7 Homo_sapiens
2 Q9Z0Z4 Mus_musculus
```

We read these data with `seqinr` which we load in memory, and then we select the SWISSPROT database.

```
library(seqinr)
s <- choosebank("swissprot")
```

We can now query the database using the accession numbers. This is done with the "ac" keyword of the function `query`. For instance, if we want to retrieve the sequence of the Eastern brown snake (no. Q7SZN0), we do:

```
> query(s$socket, "venom", "ac=Q7SZN0")


$socket:            description                              class
"->pbil.univ-lyon1.fr:5558"                               "socket"
                           mode                               text
                           "a+"                             "text"
                         opened                           can read
                       "opened"                              "yes"
                       can write
                           "yes"

$banque: swissprot
$call: query(socket = s$socket, listname = "venom",
             query = "ac=Q7SZN0")
$name: [1] "venom"

  list length mode      content
1 $req 1      character sequences
```

We then retrieve the sequence itself with getSequence (we print the first twenty amino acids to check):

```
> X <- getSequence(venom$req[[1]])
> X[1:20]
 [1] "M" "G" "R" "Y" "S" "V" "S" "P" "V" "P" "K" "C" "L" "L"
[15] "L" "M" "F" "L" "G" "W"
```

To retrieve several sequences at the same time with their accession numbers, we need to use the keyword “OU”; for instance, to get the first two sequences we are looking for we could do:

```
query(s$socket, "venom", "ac=Q9BQS7 OU ac=Q9Z0Z4")
```

We again use the function paste to put the 21 numbers together:

```
> paste("ac", venom.no$No, sep = "=")
 [1] "ac=Q9BQS7" "ac=Q9Z0Z4" "ac=Q7ZU12" "ac=Q61147"
 [5] "ac=P00450" "ac=Q804W6" "ac=Q804X3" "ac=Q7TN96"
 [9] "ac=Q06194" "ac=P00451" "ac=P12263" "ac=O62730"
[13] "ac=Q804W5" "ac=Q90X47" "ac=Q7SZN0" "ac=Q804X4"
[17] "ac=P12259" "ac=Q28107" "ac=Q9GLP1" "ac=Q7TPK2"
[21] "ac=O88783"
```

But this time we need to have all these numbers in a single character string. This is done with the option collapse. To see the result, let us do it with the first four numbers:

```
> paste("ac", venom.no$No[1:4], sep = "=", collapse = " OU ")
[1] "ac=Q9BQS7 OU ac=Q9Z0Z4 OU ac=Q7ZU12 OU ac=Q61147"
```

Because there is no need to print the whole string with the 21 numbers, we store it in an object called no4query, and use it as an argument to query:

```
> no4query <- paste("ac=", venom.no$No, sep = "",
+                    collapse = " OU ")
> query(s$socket, "venom", no4query)
```

```
$socket:          description                           class
"->pbil.univ-lyon1.fr:5558"                          "socket"
                         mode                            text
                         "a+"                          "text"
                       opened                        can read
                     "opened"                           "yes"
                    can write
                        "yes"

$banque: swissprot
$call: query(socket = s$socket, listname = "venom",
            query = no4query)
$name: [1] "venom"

  list length mode      content
1 $req 21     character sequences
```

The last line of the printed output shows that the 21 sequences have been found in ACNUC. We are now ready to download them. We do it by applying the function `getSequence` to each element of the list `venom$req`:

```
venom.seq <- lapply(venom$req, getSequence)
```

We check the results by looking at the structure of `venom.seq`:

```
> str(venom.seq)
List of 21
 $ : chr [1:1065] "M" "K" "I" "L" ...
 $ : chr [1:1062] "M" "K" "F" "L" ...
 $ : chr [1:2211] "M" "F" "L" "A" ...
 $ : chr [1:2224] "M" "F" "P" "G" ...
 $ : chr [1:2258] "M" "F" "P" "A" ...
 $ : chr [1:2351] "M" "Q" "I" "E" ...
 $ : chr [1:2319] "M" "Q" "I" "A" ...
 $ : chr [1:2133] "M" "Q" "L" "E" ...
 $ : chr [1:1158] "M" "E" "S" "G" ...
 $ : chr [1:1157] "M" "K" "A" "G" ...
 $ : chr [1:2343] "M" "Q" "V" "E" ...
 $ : chr [1:2183] "M" "L" "L" "V" ...
 $ : chr [1:1460] "M" "G" "R" "Y" ...
 $ : chr [1:2258] "M" "R" "A" "A" ...
 $ : chr [1:2102] "M" "Q" "S" "S" ...
 $ : chr [1:1087] "M" "K" "G" "L" ...
 $ : chr [1:1802] "F" "S" "P" "T" ...
 $ : chr [1:1639] "M" "R" "T" "D" ...
 $ : chr [1:1377] "V" "W" "T" "L" ...
 $ : chr [1:745] "C" "F" "Q" "V" ...
 $ : chr [1:2119] "M" "K" "L" "R" ...
```

Because the retrieval operation lost the accession numbers, we assign them as names to the list `venom.seq`:

```
names(venom.seq) <- venom.no$No
```

3.6.4 Mammalian Mitochondrial Genomes

Gibson et al. [51] made a comprehensive analysis of the mitochondrial genomes of 69 species of mammals. They explored the variations in base composition in different regions of this genome. We limit ourselves to simpler analyses. The goal is to show how to read heterogeneous data in a big file, and manipulate and prepare them in R.

The original data come from the OGRe (Organellar Genome Retrieval system) database.[8] All mammalian mtGenomes available in the database were downloaded in April 2005. This represents 109 species. The data were saved in a single file called 'mammal_mtGenome.fasta'.

The first six lines of this file show how the data are presented:

```
################################################
# OGRe sequences                               #
################################################

#DASNOVMIT : _Dasypus novemcinctus_ (nine-banded armadillo)...
#TAMTETMIT : _Tamandua tetradactyla_ (southern tamandua) : ...
....
```

After the 109 species names and codes, the sequences are printed in FASTA format. For instance, the lines 116–118 are:

```
>DASNOVMIT(ATP6)
atgaacgaaaacctatttgcctcattcgctacccctaccataataggcct...
caagtattcttttccctacccctaaacggataattaccaaccgagtggta...
....
```

Thus the species codes used in the first part of the file are used for the sequence names together with the names of the genes in parentheses. Consequently we need to get the correspondence between these species codes and the species names. Thanks to the flexibility of `read.table` we do this relatively straightforwardly. If we examine the first lines from the file above, we notice that the command needed to read the species names and codes will need to:

[8] http://ogre.mcmaster.ca/.

- Skip the first four lines,
- Read only 109 lines,
- Use the underscore "_" as the character separating the two columns,
- Ignore what comes after the scientific name on each line.

The corresponding command is (we again use the `as.is = TRUE` option for the same reason):

```
mtgen.taxa <- read.table("mammal_mtGenome.fasta", skip = 4,
  nrows = 109, sep = "_", comment.char = "(", as.is = TRUE)
```

Note that we take advantage of the fact that the common names are within parentheses: this is done with the option `comment.char` (whose default value is `"#"`). We look at the first five rows:

```
> mtgen.taxa[1:5, ]
             V1                      V2 V3
1 #DASNOVMIT :   Dasypus novemcinctus NA
2 #TAMTETMIT :  Tamandua tetradactyla NA
3 #ORYCUNMIT :  Oryctolagus cuniculus NA
4 #OCHCOLMIT :      Ochotona collaris NA
5 #LEPEURMIT :        Lepus europaeus NA
```

There are a few undesirable side-effects to our command, but this is easily solved. The fact that we set `sep = "_"` resulted in the space after the second underscore being read as a variable. We can delete it with:

```
mtgen.taxa$V3 <- NULL
```

The first column containing the species codes have a few extra characters that we wish to remove. We can do this operation with `gsub`.

```
mtgen.taxa$V1 <- gsub("#", "", mtgen.taxa$V1)
mtgen.taxa$V1 <- gsub(" : ", "", mtgen.taxa$V1)
```

Finally we change the names of the columns and check the results:

```
> colnames(mtgen.taxa) <- c("code", "species")
> mtgen.taxa[1:5, ]
       code               species
1 DASNOVMIT  Dasypus novemcinctus
2 TAMTETMIT Tamandua tetradactyla
3 ORYCUNMIT Oryctolagus cuniculus
4 OCHCOLMIT     Ochotona collaris
5 LEPEURMIT       Lepus europaeus
```

After this small string manipulation, we can read the sequences with `read.dna`. This function also has an option `skip` that we use here. We then check the number of sequences read:

```
> mtgen <- read.dna("mammal_mtGenome.fasta",
        format = "fasta", skip = 115)
> length(mtgen)
[1] 4033
```

We also check the names of the first ten sequences:

```
> names(mtgen)[1:10]
 [1] "DASNOVMIT(ATP6)" "TAMTETMIT(ATP6)" "ORYCUNMIT(ATP6)"
 [4] "OCHCOLMIT(ATP6)" "LEPEURMIT(ATP6)" "OCHPRIMIT(ATP6)"
 [7] "BERBAIMIT(ATP6)" "BALMUSMIT(ATP6)" "PONBLAMIT(ATP6)"
[10] "CAPHIRMIT(ATP6)"
```

It would be interesting now to get only the name of the gene for each sequence in a separate vector. Again we can use gsub for this, but the command is slightly more complicated because we want to remove all characters outside the parentheses, and the latter as well. We use the fact that gsub can treat regular expressions. For instance, we can do this:

```
genes <- gsub("^[[:alnum:]]{1,}\\(", "", names(mtgen))
```

where `"^[[:alnum:]]{1,}\\("` means “a character string starting with one or more alphanumeric character(s) and followed by a left parenthesis”. We need to call gsub a second time to remove the trailing right parenthesis:

```
genes <- gsub("\\)$", "", genes)
```

Note in these two examples how the caret ^ and the dollar $ are used to specify that the characters we are looking for start or end the string, respectively.[9]

After this operation it appears that some values in genes indicate that the sequence is actually empty:

```
> unique(genes)[11]
[1] "ND4Sequence does not exist"
```

To remove these missing sequences, we find them using grep:

```
> i <- grep("Sequence does not exist", names(mtgen))
> i
[1]  998 3371 3375
```

There are thus three missing sequences in the data set. We remove them with:

```
> mtgen <- mtgen[-i]
```

And we repeat the operation of extracting the sequence names:

```
genes <- gsub("^[[:alnum:]]{1,}\\(", "", names(mtgen))
genes <- gsub("\\)$", "", genes)
```

[9] The syntax of regular expressions used by R is detailed in a help page: ?regexp.

We can now look at how many sequences there are for each gene:

```
> table(genes)
genes
          ATP6           ATP8           COX1           COX2
           109            109            109            109
          COX3           CYTB            ND1            ND2
           109            109            109            109
           ND3            ND4           ND4L            ND5
           109            108            109            109
           ND6            RNL            RNS      tRNA-Ala
           109            109            109            109
      tRNA-Arg       tRNA-Asn       tRNA-Asp       tRNA-Cys
           109            109            109            109
      tRNA-Gln       tRNA-Glu       tRNA-Gly       tRNA-His
           109            109            109            109
      tRNA-Ile tRNA-Leu(CUN) tRNA-Leu(UUR)       tRNA-Lys
           109            109            109            109
      tRNA-Met       tRNA-Phe       tRNA-Pro tRNA-Ser(AGY)
           109            109            107            109
tRNA-Ser(UCN)       tRNA-Thr       tRNA-Trp       tRNA-Tyr
           109            109            109            109
      tRNA-Val
           109
```

We see that we miss one sequence of "ND4" and two of "tRNA-Pro" (this can be seen more clearly with `sort(table(genes))`). We are now ready to do all sorts of analyses with this data set. We see how to analyze base frequencies at three levels of variation:

- Between species (all genes pooled);
- Between genes (all species pooled);
- Between sites for a single protein-coding gene (all species pooled).

To calculate the base frequencies for each species, we first create a matrix with 109 rows and 4 columns that will store the results:

```
BF.sp <- matrix(NA, nrow = 109, ncol = 4)
```

We set its rownames with the species names, and the colnames with the four base symbols:

```
rownames(BF.sp) <- mtgen.taxa$species
colnames(BF.sp) <- c("A", "C", "G", "T")
```

We put in each row of this matrix the frequency of each base. This involves:

1. Selecting only the sequences with the corresponding species code using `grep`;

2. Computing the base frequencies for the selected sequences with the function `base.freq`;
3. Repeating these two operations for all 109 species.

A simple approach is to use a `for` loop where a variable, say `i`, will vary from 1 to 109: this will be used as index for both `BF.sp` and `mtgen.taxa$code`. The commands are relatively straightforward and use some elements seen above. For clarity, we write two separate commands within the loop (the indices of the selected genes are stored in `x`):

```
for (i in 1:109) {
  x <- grep(mtgen.taxa$code[i], names(mtgen))
  BF.sp[i, ] <- base.freq(mtgen[x])
}
```

To visualize the results, we use the graphical function `matplot` which plots the columns of a matrix. We add the options `type = "l"` to have lines (the default is points), and `col = 1` to avoid colors. We further add a legend (Fig. 3.4):

```
matplot(BF.sp, type = "l", col = 1, xlab = "Species",
        ylab = "Base frequency")
legend(0, 0.23, c("A", "C", "G", "T"), lty = 1:4, bty = "n")
```

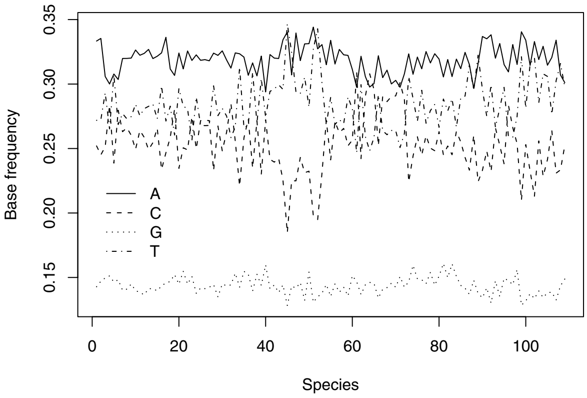


Fig. 3.4. Plot of the base frequencies of the mitochondrial genome of 109 species of mammals

The second analysis—between genes for all species pooled—will follow the same lines as the previous one. The matrix used to store the results will have 37 rows, and its rownames will be the names of the genes.

A subtlety here is the need to use the option `fixed = TRUE` in `grep`: the reason is that some gene names contain parentheses and these characters have a special meaning in regular expressions. The option used here forces `grep` to treat its first argument as a simple character string, and thus avoids this annoyance. The full set of commands is:

```
BF.gene <- matrix(NA, nrow = 37, ncol = 4)
rownames(BF.gene) <- unique(genes)
colnames(BF.gene) <- c("A", "C", "G", "T")
for (i in 1:37) {
    x <- grep(rownames(BF.gene)[i], names(mtgen), fixed = TRUE)
    BF.gene[i, ] <- base.freq(mtgen[x])
}
```

We represent the results in a different way by using the function `barplot` which, by default, makes a stacked barplot of the rows for each column: we thus need to transpose the matrix `BF.gene` first. Because some gene names are somewhat long, we modify the margins; we also use the options `las = 2` to force the labels on the x-axis to be vertical, and `legend = TRUE` to add a legend (Fig. 3.5):

```
par(mar = c(8, 3, 3, 2))
barplot(t(BF.gene), las = 2, legend = TRUE)
```

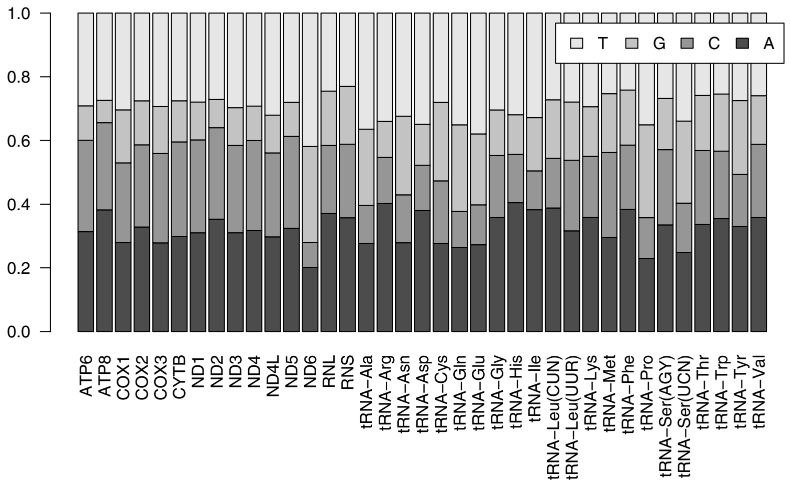


Fig. 3.5. Plot of the base frequencies of the mitochondrial genome of 109 species of mammals for each gene

For the third analysis—between sites for a single gene—we focus on the genes of the cytochrome *b* whose code is `CYTB`. We first extract the sequences of this gene by taking the appropriate indices in the way seen above:

```
cytb <- mtgen[grep("CYTB", names(mtgen))]
```

We now look at the length of each sequence using `lapply` and `length`, and summarize the results with `table`:

```
> table(unlist(lapply(cytb, length)))

1135 1137 1138 1139 1140 1141 1143 1144 1146 1149
   1    4    3    1   78   10    2    1    7    2
```

The majority of these sequences has 1140 sites and thus it is likely that they are properly aligned. Furthermore a look at the first few nucleotides (which can be done with `str(cytb)`) suggests this is true for the whole 109 sequences. For simplicity we assume this to be correct, although a more rigorous check of the alignment, as done for the other cases above, is possible.

To extract the first, second, or third codon position we need to do the operation within each sequence. Thus we need a command such as (for the first position) `cytb[[1]][c(TRUE, FALSE, FALSE)]` repeated for each sequence in `cytb`. There are several solutions for this: we choose one where the extraction using logical indexing is included in a function that we apply to each element of `cytb`. This is done three times with moving the position of `TRUE`:

```
cytb1 <- lapply(cytb, function(x) x[c(TRUE, FALSE, FALSE)])
cytb2 <- lapply(cytb, function(x) x[c(FALSE, TRUE, FALSE)])
cytb3 <- lapply(cytb, function(x) x[c(FALSE, FALSE, TRUE)])
```

`cytb1`, `cytb2`, and `cytb3` are three lists containing the first, second, and third positions, respectively. We can now proceed in a similar way as done above:

```
> BF.cytb <- matrix(NA, 3, 4)
rownames(BF.cytb) <- c("1st codon position",
                       "2nd codon position",
                       "3rd codon position")
colnames(BF.cytb) <- c("A", "C", "G", "T")
BF.cytb[1, ] <- base.freq(cytb1)
BF.cytb[2, ] <- base.freq(cytb2)
BF.cytb[3, ] <- base.freq(cytb3)
BF.cytb
                           A         C          G         T
1st codon position 0.2902603 0.2628290 0.21503534 0.2318753
2nd codon position 0.2018149 0.2491915 0.13636144 0.4126321
3rd codon position 0.4043395 0.3778631 0.03600994 0.1817875
```

We plot the results again using `barplot` but adding a few annotations to present the figure (Fig. 3.6):

```
barplot(t(BF.cytb), main = "Cytochrome b",
        ylab = "Base frequency")
text(0.7, BF.cytb[1, 1]/2, "A", cex = 2)
text(0.7, BF.cytb[1, 1] + BF.cytb[1, 2]/2, "C", cex = 2)
text(0.7, sum(BF.cytb[1, 1:2])+BF.cytb[1, 3]/2, "G", cex = 2)
text(0.7, sum(BF.cytb[1, 1:3])+BF.cytb[1, 4]/2, "T", cex = 2)
```

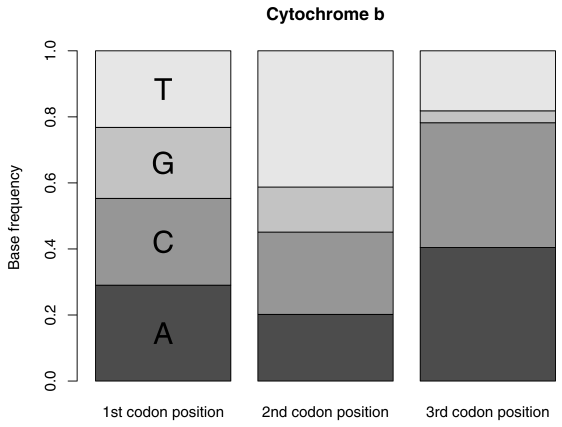


Fig. 3.6. Plot of the base frequencies at the three codon positions of the gene of the cytochrome *b* for 109 species of mammals for each gene

3.6.5 Butterfly DNA Barcodes

Hebert et al. [67] analyzed the molecular variation in the neotropical skipper butterfly *Astraptes fulgerator* in order to assess the species limits among different forms known to have larval stages feeding on distinct host plants. They sequenced a portion of the mitochondrial gene cytochrome oxydase I (COI) of 466 individuals belonging to 12 larval forms. The goal of this application is to prepare a large data set of DNA sequences, and align them for further analyses (Chapter 5).

The GenBank accession numbers are AY666597–AY667060, AY724411, and AY724412 (there is a printing error in [67] for these last two numbers). We read the sequences with `read.GenBank` in the same way as seen for the *Sylvia* or Felidae data.

```
x <- paste("AY66", 6597:7060, sep = "")
```

```
x <- c(x, "AY724411", "AY724412")
astraptes.seq <- read.GenBank(x)
```

We then look at how the sequence lengths are distributed:

```
> table(unlist(lapply(astraptes.seq, length)))

208 219 227 244 297 370 373 413 440 548 555 573 582 599 600
  1   1   1   1   1   1   1   1   1   1   3   1   1   1   1
601 603 608 609 616 619 620 623 626 627 628 629 630 631 632
  2   2   1   1   2   1   1   1   4   1   5   3   4   3   7
633 634 635 636 638 639
  1   1  12   6   2 389
```

The sequences clearly need to be aligned. We resort to Clustal X once more by first saving the sequences in FASTA format:

```
write.dna(astraptes.seq, "astraptesseq.fas", format = "fasta")
```

As before, the alignment made by Clustal X is saved in “Phylip” (interleaved) format, and are read back into R:

```
astraptes.seq.ali <- read.dna("astraptesseq.phy")
```

We check the species names of the sequences downloaded from GenBank:

```
> table(attr(astraptes.seq, "species"))

Astraptes_sp._BYTTNER    Astraptes_sp._CELT
                    4                    23
  Astraptes_sp._FABOV  Astraptes_sp._HIHAMP
                   31                    16
 Astraptes_sp._INGCUP  Astraptes_sp._LOHAMP
                   65                    47
 Astraptes_sp._LONCHO    Astraptes_sp._MYST
                   41                     3
   Astraptes_sp._NUMT  Astraptes_sp._SENNOV
                    4                   102
  Astraptes_sp._TRIGO  Astraptes_sp._YESENN
                   51                    79
```

All specimens were thus attributed to *Astraptes* sp. with further information given as a code (explained in [67]). We do the same operation as above to store the taxon names with the accession numbers:

```
taxa.astraptes <- attr(astraptes.seq, "species")
names(taxa.astraptes) <- names(astraptes.seq)
```

We finally save the data for further analyses:

```
save(astraptes.seq.ali, taxa.astraptes,
     file = "astraptes.RData")
```

3.7 Exercises

Exercises 1–3 aim at familiarizing the reader with tree data structures in R; Exercises 4–6 give more concrete applications of the concepts from this chapter.

1. Create a random tree with 10 tips.
   (a) Extract the branch lengths, and store them in a vector.
   (b) Delete the branch lengths, and plot the tree.
   (c) Give new, random branch lengths from a uniform distribution $U[0, 10]$. Do this in a way that works for any number of tips.
   (d) Restore the original branch lengths of the tree.

2. Create a random tree with 5 tips, print it, and plot it. Find the way to delete the class of this object, and print it again. Try to print it again: comment on what happens. Find a way to force the plot of the tree as before.

3. Generate three random trees with 10 tips. Write them in a file. Read this file in R. Print a summary of each tree. Write a small program that will do these operations for any number of trees (say `N`) and any number of tips (`n`).

4. Extract the tree #1000 in TreeBASE. Make three copies of this tree, and give them branch lengths (i) all equal to one, (ii) so that the node heights are proportional to the number of species, and (iii) randomly extracted from a uniform distribution $U[0, 0.1]$.

5. Extract the sequences of the cytochrome *b* gene with the accession numbers U15717–U15724 (source: [59]).
   (a) Print the species names of each sequence.
   (b) Print, with a single command, the length of each sequence.
   (c) Arrange the data in a matrix.
   (d) Extract and store in three matrices the first, the second, and the third codon positions of all sequences. Compute their base frequencies. What do you conclude?
   (e) Save the three matrices in three different files. Read these files, and concatenate the three sets of sequences.

6. Get the following sequences from GenBank:
   - AF518328–AF51837 (source: [89]),
   - AF141220, AF004572, AF141219, AF004586, AF141217, AF004587, AB033713, AB033699, AB032853, AB033695.

   Prepare them along the same lines as in Section 3.6.

4

Plotting Phylogenies

Drawing phylogenetic trees has been important for a long time in the study of biological evolution, as illustrated by Darwin's only figure in his *Origin of Species* [22]. A plotted phylogeny is the usual way to summarize the results of a phylogenetic anlysis. This also gives the essence of the evolutionary processes and patterns.

Quite surprisingly, graphical tools have been somewhat neglected in the analysis of phylogenetic data. There is a very limited treatment on graphics in recent phylogenetics textbooks [39, 60, 106]. On the other hand, an important area of statistical research has been developed on the graphical analysis and exploration of data. Some of these developments have been implemented in R (e.g., see the `lattice` package). R also has a flexible and programmable graphical environment.

There are undoubtedly values in the graphical exploration of phylogenetic data. Character mapping has been done for some time in some issues, and it will be valuable to have a more general approach for graphical analysis and exploration of phylogenetic data. In this chapter, I explore some of these ideas, as well as explaining how to plot phylogenetic trees in simple ways. Inasmuch as there are many illustrations throughout the chapter, there are no case studies.

4.1 Simple Tree Drawing

`plot.phylo` in `ape` can draw four kinds of trees: phylograms (also called rectangular cladograms), cladograms (triangular cladograms), unrooted trees (dendrograms), and radial (circular) trees. This function is a method: it uses R's syntax of the generic function `plot`, and acts specifically on `"phylo"` objects. It has several options; all of them are defined with default values. In its most direct use (i.e., `plot(tr)`) a phylogram is plotted on the current graphical device from left (root) to right (tips) with some space around (as defined by the current margins). The branch lengths, if available, are used. The tip

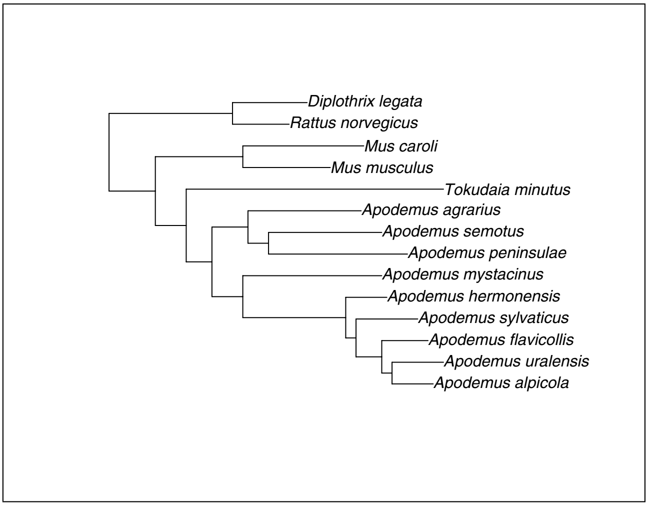


Fig. 4.1. A simple use of `plot(tr)`

labels are printed in italics, left-justified from the tips of their respective terminal branches. The node labels and the root edge, if available, are ignored. As an example, Fig. 4.1 shows a tree named `tr` showing the relationships among some species of woodmice (*Apodemus*) and a few closely related species of rodents published by Michaux et al. [100]. The tree was plotted, after being read with `tr <- read.tree("rodent.tre")` (Section 3.2), by simply typing `plot(tr)`.[1]

The options alter these settings. They are described in Table 4.1. Most of these options have intuitive effects (e.g., `type`, `font`, etc.), whereas some have a `NULL` value by default. This means that, unless the user gives a specific value, it is determined with respect to other arguments. We have seen an illustration of this mechanism above with the simple command `plot(tr)`.

An obvious case where one option alters the default value of another is when the tree is plotted leftwards using `direction = "l"`: the labels are now right-justified, which seems an obvious consequence of the change in direction. For instance, a leftwards cladogram of the same tree may be obtained with (the resulting plot is in Fig. 4.2):

```
plot(tr, type = "c", use.edge.length = FALSE,
     direction = "l")
```

If the user wants to keep the labels left-justified, then the option `adj` must be used (Fig. 4.3):

```
plot(tr, type = "c", use.edge.length = FALSE,
```

[1] In this chapter, the box delimiting the figures indicates the presence of margins around the tree.

Table 4.1. The options of `plot.phylo`. The values marked with (d) are the default ones

Option	Effect	Possible values
`type`	Type of tree	`"p"` (d), `"c"`, `"u"`, `"r"`
`use.edge.length`	Whether to use branch lengths	`TRUE` (d), `FALSE`
`node.pos`	Vertical position of the nodes with respect to the positions of the tips	`NULL` (d), 1, 2
`show.tip.label`	Whether to show tip labels	`TRUE` (d), `FALSE`
`show.node.label`	Whether to show node labels	`FALSE` (d), `TRUE`
`edge.color`	The line colors of the edges	`NULL` (d), a vector of strings giving the colors
`edge.width`	The line thickness of the edges	`NULL` (d), a vector of numeric values
`font`	The font of the labels	1 (normal), 2 (bold) 3 (italics) (d), 4 (bold italics)
`cex`	Relative character size	A numeric value (default: 1)
`adj`	Horizontal and vertical adjustment of the labels	`NULL` (d), one or two numeric values
`srt`	Rotation of the labels	A numeric value (default: 0)
`no.margin`	Leave some space around the tree	`FALSE` (d), `TRUE`
`root.edge`	Draw the root edge	`FALSE` (d), `TRUE`
`label.offset`	Space between the tips and the labels	0 (d), a numeric value
`underscore`	Display the underscores in tip labels	`FALSE` (d), `TRUE`
`x.lim`	Limits on the horizontal axis	`NULL` (d), two numeric values
`y.lim`	Limits on the vertical axis	`NULL` (d), two numeric values
`direction`	Direction of the tree	`"r"` (d), `"l"`, `"u"`, `"d"`
`lab4ut`	Style of labels for unrooted trees	`"horizontal"` (d), `"radial"`

```
        direction = "l", adj = 0)
```

Many publishers of journals or books prefer to receive figures in Encapsulated PostScript (EPS) format. The function `postscript` in R may be used to produce such files. Note that when the tree is plotted in a PostScript file, the default is to print in landscape format so that the tree will be vertical if the page is viewed in portrait format. To set the page in portrait format, you must set `horizontal = FALSE` in the function `postscript`.

In R, it is possible to add further graphical elements to an existing plot using the *low-level plotting commands* (see, e.g., [154, Chap. 4], for further details on how R graphics work). `plot.phylo` exploits this by letting the user manage the space around the tree. This can be accomplished in two non-exclusive ways: either through setting the margins, or by changing the scales of the axes.

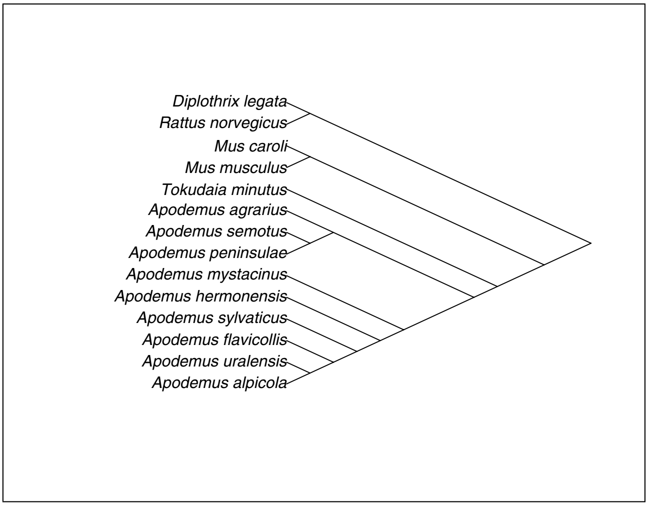


Fig. 4.2. A leftwards cladogram with default label justification

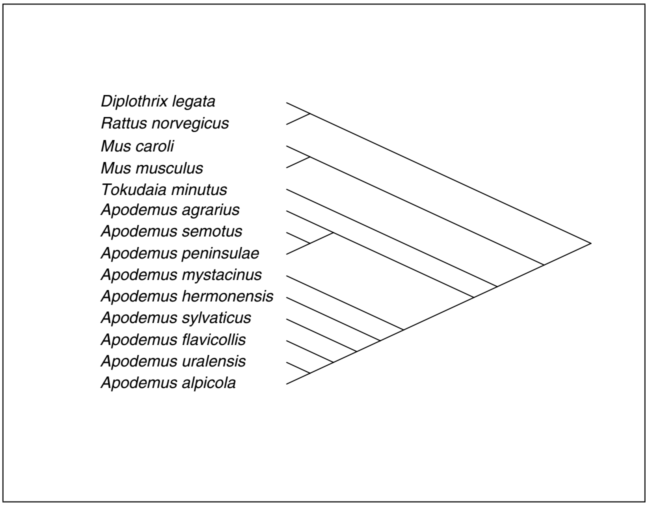


Fig. 4.3. A leftwards cladogram with left-justified labels

When plotting a tree, the current margins are used. The size of the latter, in number of lines, can be found by querying the graphical parameters with the command `par("mar")`. By default, this gives:

```
> par("mar")
[1] 5.1 4.1 4.1 2.1
```

These can be changed with, for instance:

```
par(mar = rep(1, 4))
```

The option `no.margin = TRUE` in `plot.phylo` has the same effect as doing:

```
par(mar = rep(0, 4))
```

The margins of a graphic are usually used to add text around a plot: this is done with the function `mtext` (*marginal text*). The axes can also be drawn with the function `axis`, but this is likely to be informative only for the axis parallel to the branches. Also the default display of the tick marks may not be appropriate for the tree (see the functions `axisPhylo` and `add.scale.bar`, Section 4.1.1). Finally, the function `box` adds a box delimiting the margins from the plot region where the tree is drawn.

The other way to manage space around the tree is to alter the scales of the plotting region itself. `plot.phylo` draws the edges using the lengths of the `"phylo"` object directly, then computes how much space is needed for the labels, and sets the axes so that the plotting region is optimally used. Unless the axes are displayed explicitly with the `axis` function, the user does not know the size of the plotting region. However, `plot.phylo` invisibly returns (meaning that it is not normally displayed) a list with the option values when it was called. This list can be accessed by assigning the call; its elements are then extracted in the usual way:

```
> tr.sett <- plot(tr)
> names(tr.sett)
 [1] "type"             "use.edge.length" "node.pos"
 [4] "show.tip.label"   "show.node.label" "edge.color"
 [7] "edge.width"       "font"            "cex"
[10] "adj"              "srt"             "no.margin"
[13] "label.offset"     "x.lim"           "y.lim"
[16] "direction"
> tr.sett$x.lim
[1] 0.0000000 0.1229417
> tr.sett$y.lim
[1]  1 14
```

This shows that the horizontal axis of the plot in Fig. 4.1 ranges from 0 to 0.123. To draw the same tree but leaving about half the space of the plot region free either on the right-hand side, or on the left-hand side, one can do:

```
> plot(tr, x.lim = c(0, 0.246))
> plot(tr, x.lim = c(-0.123, 0.123))
```

Drawing unrooted trees is a difficult task because the optimal positions of the tips and nodes cannot be found in a straightforward way. `plot.phylo` uses a simple algorithm, inspired by the program drawtree in Phylip, where clades are allocated angles with respect to their number of species [39]. With this scheme, edges should never cross. The option `lab4ut` (*labels for unrooted trees*) allows two positions for the tip labels: `"horizontal"` (the default) or

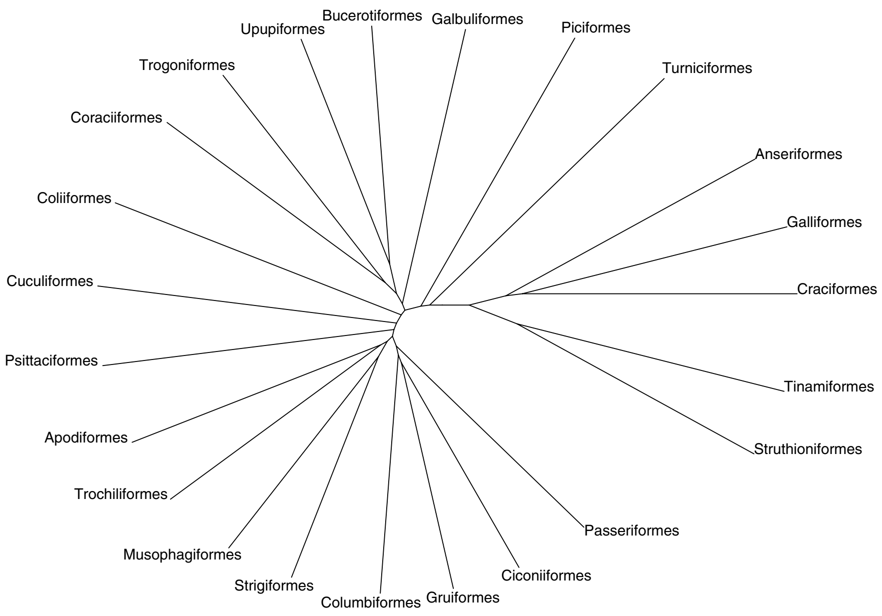


Fig. 4.4. An unrooted tree of the bird families

`"radial"`. Using the latter and adjusting the font size with `"cex"` is likely to give readable trees in most situations, even if they are quite large. Figure 4.4 shows an unrooted tree of the recent bird orders [140]. The command used is:

```
plot(bird.orders, type = "u", font = 1, no.margin = TRUE)
```

Circular trees are drawn with `ade4` using the function `radial.phylog`. After converting our original tree as explained in Section 3.4.5, a simple call to this function results in Fig. 4.5. In `ade4`, the tip labels are drawn on the same level; the tips themselves are marked (by default) with black circles.

`ape` can also plot circular trees by using the option `type = "radial"` in `plot.phylo` but this does not take branch lengths into account. All tips are placed equispaced on a circle, the root being at the center of this circle. The nodes are then placed on concentric circles with distances from the outer circle depending on the number of descendant tips. Figure 4.6 shows an example with the families of birds [140]. This representation can be used for rooted and unrooted trees. It has the advantages of being easily computed; the lines have no chance to cross and the tips are equally spaced.

`apTreeshape` has its own `plot` method for trees of class `"treeshape"` (`plot.tresshape`): it results in a simple plot of a tree similar to the default behavior of `plot.phylo` (Fig. 3.1). An original feature of this method is the possibility of directly plotting two trees on the same graphical device

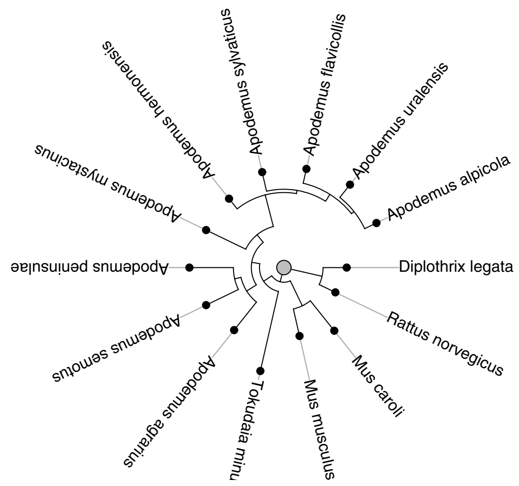


Fig. 4.5. A circular tree with `radial.phylog`

with `plot(t1, t2)`. Section 4.2 explains how to do similar plots with objects of class `"phylo"`.

4.1.1 Annotating Trees

`plot.phylo` allows us to display node labels with the option `show.node.label`: this simply prints the labels using the same font and justification as for the tips. This option is very limited, and it is often needed to have a more flexible mechanism to display clade names, bootstrap values, estimated divergence dates, and so on. Furthermore, the character strings displayed with `show.node.label = TRUE` are from the `node.label` element of the `"phylo"` object, whereas it may be needed to display values coming from some other data.

Node Annotation

The function `nodelabels` offers a flexible way to add labels on a tree. It is a low-level plotting function: the labels are added on a previously plotted tree. It can print text (like the function `text`), plotting symbols (like `points`), or "thermometers" (like `symbols`) on all or some selected nodes. The formatting allows us to place the labels exactly on the node, or at a point around it, thus giving the possibility of adding information. The text can be framed with rectangles or circles, and colors can be used.

The number of options of `nodelabels` is quite small (Table 4.2), but it takes advantage of the `...` (pronounced "dot-dot-dot") argument of R's methods. This "mysterious" argument means that all arguments that are not predefined (i.e., those not in Table 4.2 in the present case) are passed internally

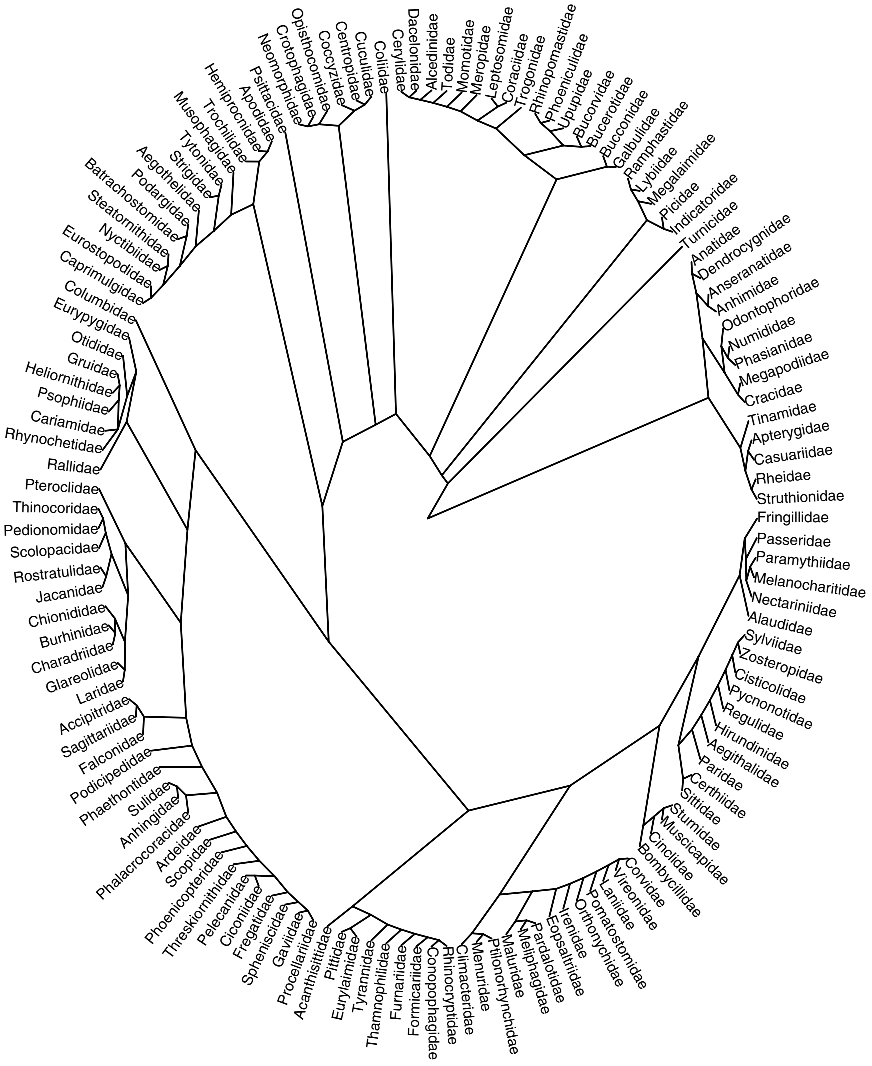


Fig. 4.6. A circular tree using `type = "radial"` in `plot.phylo`

to another function, in the present case either `text` or `points` (see below). Particularly, `text` has a few options to define font, character expansion, and position of the text (some examples are given in Table 4.2) which thus may be used in `nodelabels`.

Table 4.2. The options of `nodelabels`. The values marked with (d) are the default ones

Option	Effect	Possible values
`text`	Text to be printed	A vector of strings; can be left missing (d)
`node`	Nodes where to print	A vector of numerics or strings; can be left missing (d)
`adj`	Position with respect to the node	One or two numeric values
`frame`	Type of frame around text	`"r"` (d), `"c"`, `"n"`
`pch`	The type of plotting symbol	An integer between 1 and 25, or a character string
`thermo`	Draw filled thermometers with one or two levels	A numeric vector or matrix
`col`	Color for text or symbol	A character string or a color code
`bg`	Color for the background of the frame or the symbol	id. (default: `"lightblue"`)
`...`	Further arguments	`cex =` , `font =` , `vfont =` `offset =` , `pos =`

The option `pch` is defined as `NULL` by default, meaning that some text will be printed by default; if `pch` is given a value, then `text` is ignored. The nodes where the labels are printed are specified with `node`: this is done using the numbers of the `edge` element of the `"phylo"` object. The numbers specified can be either positive (1, 2, ...) or negative (−1, −2, ...), and can also be given as character strings (`"1"`, `"2"`, ..., or `"-1"`, `"-2"`, ...). Obviously, it seems necessary to know these node numbers to use `nodelabels`, but this is not a difficulty: they can be displayed on the screen using this function with no argument (i.e., `nodelabels()`; Fig. 4.7).

Another way to proceed is to assume that the vector of labels (or symbols to plot) is already ordered along the nodes: they will be displayed on the nodes in the correct order.

For a very simple operational example, consider plotting a tree showing the estimated divergence dates among gorillas, chimpanzees, and humans. We take the dates estimated by Stauffer et al. [145]:

```
trape <- read.tree(text = "((Homo,Pan),Gorilla);")
plot(trape, x.lim = c(-0.1, 2.2))
nodelabels("6.4 Ma", 1, frame = "c", bg = "white")
nodelabels("5.4 Ma", 2, frame = "c", bg = "white")
```

Because the labels need some space, we have to leave a little extra space between the root and the left-hand side margin, hence the use of the `x.lim` option (Fig. 4.8). We know that the root is numbered −1, so the first date is printed by simply giving 1 as second argument. Similarly, the second node

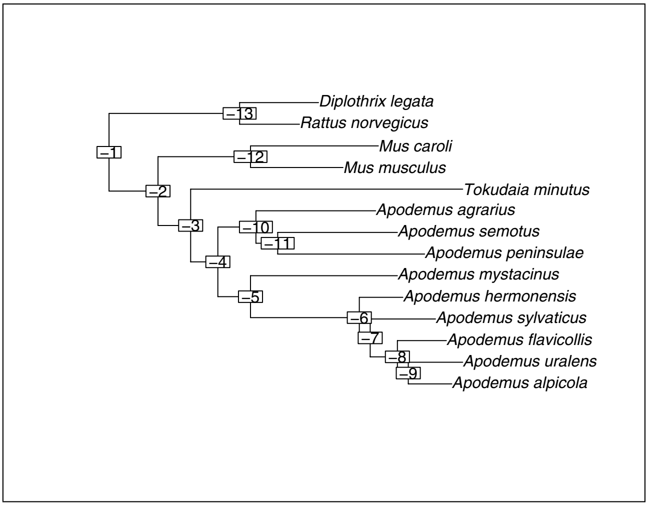


Fig. 4.7. Display of node numbers with `nodelabels()`

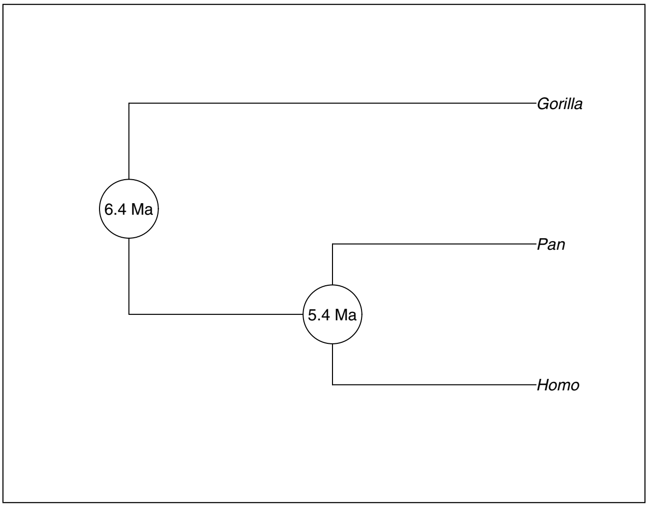


Fig. 4.8. Adding dates with `nodelabels`

is obviously numbered −2. If the node numbers are omitted, the labels are printed successively on all nodes. Thus, the same figure could have been obtained with:

```
plot(trape, x.lim = c(-0.1, 2.2))
nodelabels(c("6.4 Ma", "5.4 Ma"), frame = "c", bg = "white")
```

This is clearly useful if one has a large number of values to add on the tree. It is also often needed to print numeric values close to, but not exactly on, the nodes, for instance, bootstrap values. Usually, such values are arranged in a vector (say `bs`) and ordered along the node numbers, because this is the interface of the `"phylo"` objects. It is common to print the bootstrap values right to the nodes and without frames which can be done simply with:

```
plot(tr)
nodelabels(bs, adj = 0, frame = "n")
```

In some cases, this may need to be tuned slightly because the labels will be stuck to the nodes and the font size may be too large (or too small): the former can be moved slightly rightwards by giving a small negative value to `adj` (e.g., `adj = -0.2`), and the font size can be set by using the option `cex`.

Note that here a single value has been given to `adj`: this sets the horizontal justification only, and this conforms to standard R's graphical functions (see `?par` in R for details).

If a program outputs bootstrap values as node labels in a Newick tree, then this can be handled easily because once the tree has been read with `read.tree` these values are stored in the `node.label` element of the `"phylo"` object (see Section 3.1.1). They can be plotted with something like:

```
plot(tr)
nodelabels(tr$node.label, adj = 0, frame = "n")
```

It is also usual to plot several values around a node. Michaux et al. [100] showed on their tree bootstrap values from the different phylogeny reconstruction methods they used: parsimony, neighbor-joining, and maximum likelihood. This can be done by successive calls to `nodelabels` with different values for `adj`. The option `font` can be used to distinguish the different values. We first input the bootstrap values on the keyboard simply using `scan`:

```
> bs.pars <- scan()
1: NA 76 34 54 74 100 56 91 74 60 63 100 100
14:
Read 13 items
> bs.nj <- scan()
1: NA 74 48 68 75 100 NA 91 67 82 52 100 100
14:
Read 13 items
> bs.ml <- scan()
1: NA 88 76 73 71 100 45 81 72 67 63 100 100
14:
Read 13 items
```

There are of course many other ways to input these values. Note that we have given a missing value to the first node, because this is the root and the tree was rooted with an outgroup. We then plot the tree without the margins to leave more space for the bootstrap values, and add successively the latter with three calls to nodelabels (Fig. 4.9):

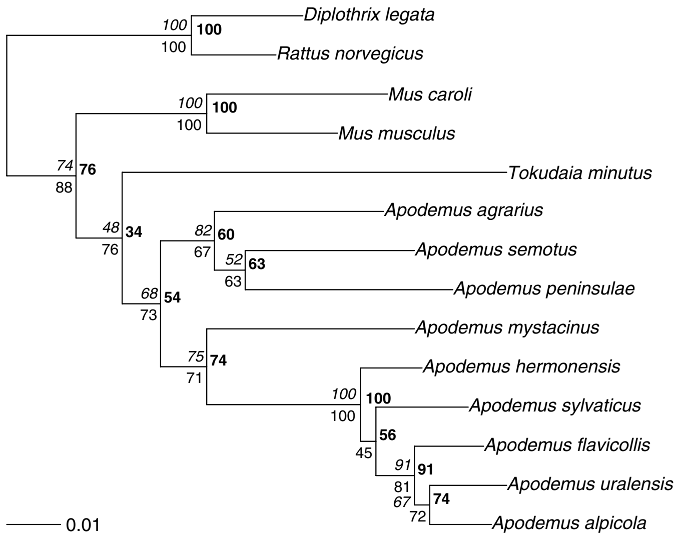


Fig. 4.9. Adding bootstrap values

```
plot(tr, no.margin = TRUE)
nodelabels(bs.pars, adj = c(-0.2, -0.1), frame = "n",
           cex = 0.8, font = 2)
nodelabels(bs.nj, adj = c(1.2, -0.5), frame = "n",
           cex = 0.8, font = 3)
nodelabels(bs.ml, adj = c(1.2, 1.5), frame = "n", cex = 0.8)
add.scale.bar(length = 0.01)
```

The last command adds a scale bar (see below for explanation of this function).

To graphically display the different levels of a single proportion, say `bs.ml`, we can use the option `thermo`. It represents the proportions of two or more categories as a filled thermometer. This representation is less usual than circular symbols such as piecharts, but the latter are less intelligible, particularly with more than three proportions. The commands are (Fig. 4.10):

```
plot(tr, no.margin = TRUE)
nodelabels(thermo = bs.ml/100, col = "grey", bg = "white")
```

We now illustrate the use of the `pch` option by plotting symbols instead of the raw numeric values. For this, we consider again the bootstrap values of the maximum likelihood method (`bs.ml`). Suppose we want to plot a filled circle for a bootstrap value greater than or equal to 90, a grey circle for a value between 70 and 90, and an open circle for a value less than 70. We first create a vector of mode character and assign strings with respect to the original bootstrap values according to the rules defined above.

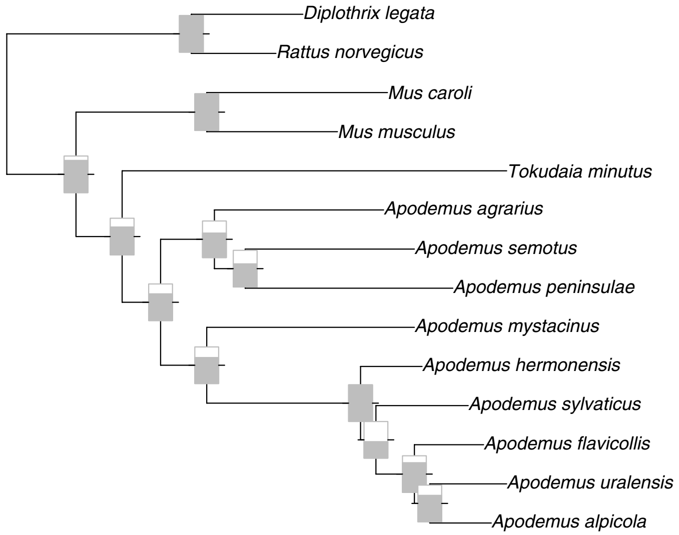


Fig. 4.10. Plotting proportions on nodes with thermometers

```
p <- character(length(bs.ml))
p[bs.ml >= 90] <- "black"
p[bs.ml < 90 & bs.ml >= 70] <- "grey"
p[bs.ml < 70] <- "white"
```

We can now plot the tree, then call `nodelabels` giving `p` as value for the option `bg`. We also specify `pch = 21` which uses a color-filled circle.

```
plot(tr, no.margin = TRUE)
nodelabels(node = 2:13, pch = 21, bg = p[-1], cex = 2)
```

Here we must use `node` to avoid a symbol being plotted at the root. Also we have to tell the option `bg` to ignore the first value of `p` (which is actually an empty string). To finish the figure, we further add a legend by two calls to `points` and `text` (Fig. 4.11):

```
points(rep(0.005, 3), 1:3, pch = 21, cex = 2,
       bg = c("black", "grey", "white"))
text(rep(0.01, 3), 1:3, adj = 0,
     c("90 <= BP", "70 <= BP < 90", "BP < 70"))
```

The function `tiplabels` plots labels at the tips of the tree, and has exactly the same syntax as `nodelabels` except that the argument `node` is replaced by `tip`.

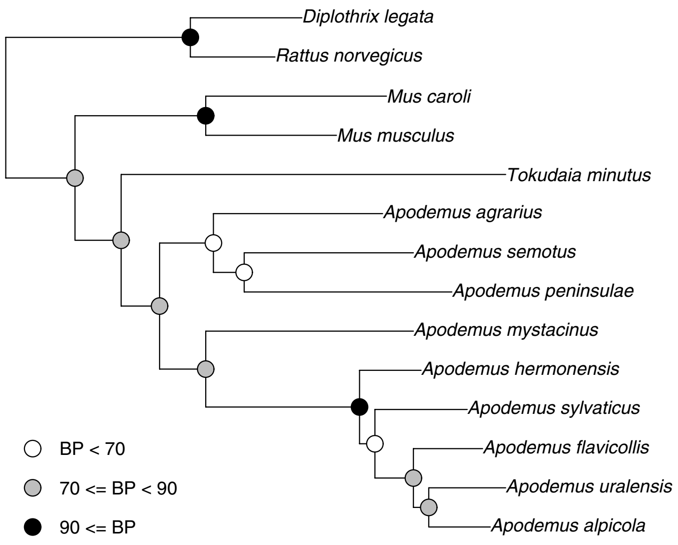


Fig. 4.11. Plotting symbols on nodes

Axes and Scales

`ape` has two low-level plotting functions that add an indication of the scale of the branches on a phylogeny plot.

`add.scale.bar()` adds a short bar at the bottom left corner of the plotting region. If this default location is not suitable, it can be modified with the arguments `x` and `y`. The length of the bar is calculated from the lengths of the plotted tree (so this works even if the tree has no branch lengths); this can be modified too with the `length` option (see Fig. 4.9).

`axisPhylo()` adds a scale on the bottom side of the plot which scales from zero on the rightmost tip to increasing values leftwards (see Figs. 4.17 and 4.18). If the tree is ultrametric, this may represent a time scale. The option `side` allows us to draw the scale on different sides of the plot: `side = 1` (the default) draws it below, `2` on the left, `3` above, and `4` on the right. Note that either `2` or `4` should be used if the tree is vertical.

Manual Annotation

R's low-level plotting commands can be used to annotate tree manually a once it has been plotted. The useful functions in this context are `text`, `segments`, `arrows` (all have explicit names), and `mtext` (*marginal* text). Except for the last one, the coordinates must be given by the user.

A simple, but hopefully didactic example, plots a four-taxon tree, and add various annotations (Fig. 4.12):

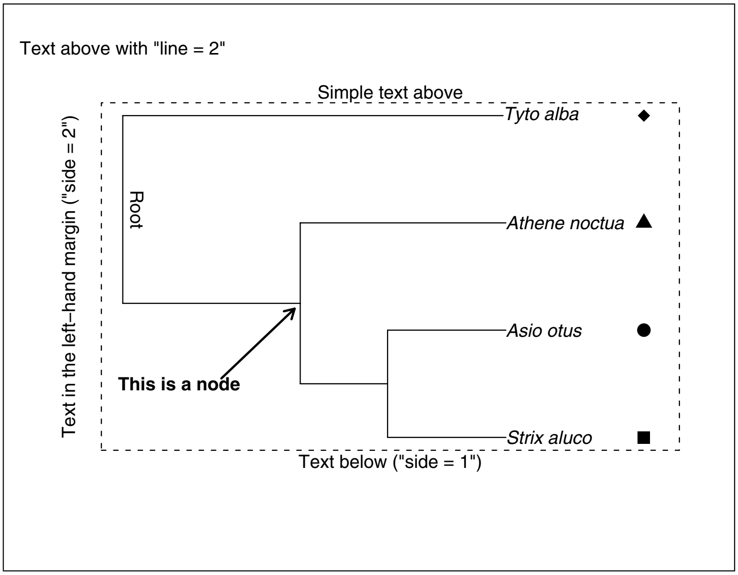


Fig. 4.12. Manual annotation of a tree

```
tree.owls <- read.tree(text = "(((Strix_aluco:4.2,
  Asio_otus:4.2):3.1,Athene_noctua:7.3):6.3,
  Tyto_alba:13.5);")
plot(tree.owls, x.lim = 19)
box(lty = 2)
text(2, 1.5, "This is a node", font = 2)
arrows(3.5, 1.55, 6.1, 2.2, length = 0.1, lwd = 2)
text(0.5, 3.125, "Root", srt = 270)
points(rep(18.5, 4), 1:4, pch = 15:18, cex = 1.5)
mtext("Simple text above")
mtext("Text above with \"line = 2\"", at = 0, line = 2)
mtext("Text below (\"side = 1\")", side = 1)
mtext("Text in the left-hand margin (\"side = 2\")",
      side = 2, line = 1)
```

The call to `box` helps to visualize the limit between the plotting region and the margins. Note the use of the option `x.lim` to leave a little extra space for the symbols plotted by `points`. By default, `mtext` prints the text at the center of the closest line to the plotting region: this is altered by the options `at` and `line`, respectively, as illustrated above. Note how double quotes are specified inside a character string: a backslash is needed to escape them.

Colors (which are not used here) can be specified in all of these functions with the `col` options.

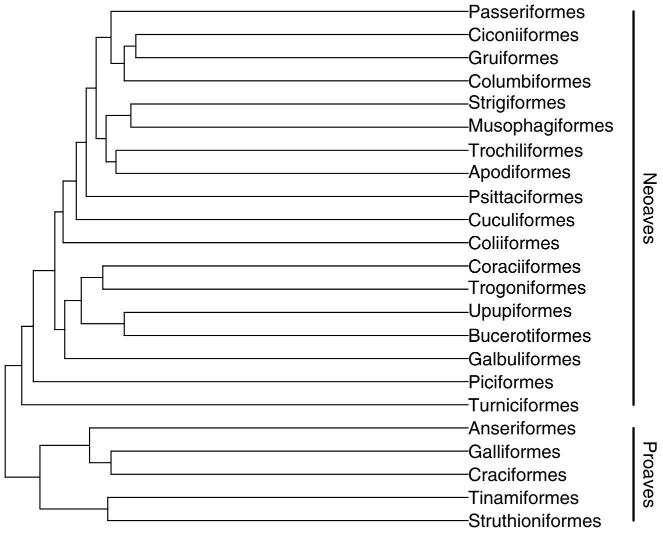


Fig. 4.13. Simple bars

4.1.2 Showing Clades

Trees are statistical tools for classification of observations, and it is obvious that in some situations clades (monophyletic groups) need to be identified in a plotted phylogeny. This may be for simple illustrative purpose, for instance, to show how different groups segregate on a phylogeny, or for exploratory reasons. In the latter case, an automated approach is clearly required.

I have found four ways commonly used in the literature to show clades on a phylogeny:

- Drawing bars in the face of the tips of the clade;
- Labeling the node corresponding to the most recent common ancestor of the clade;
- Coloring the branches of the clade;
- Drawing an ellipse or a rectangle over the branches and tips belonging to the clade.

The second approach is covered in Section 4.1.1. The first and fourth approaches are mostly appropriate for illustrative purposes, whereas the second and third ones are the best suited for exploratory analyses.

Bars can be added easily on the side of a tree with the low-level plotting command `segments`. The options of this function that are useful in this context are `lwd` for the line width and `col` for its color. When drawing such bars, it will be necessary to leave some space on the appropriate side of the plot.

It is useful to know that the tips of the tree are drawn in the same order as in the element `tip.label` in the `"phylo"` object, and their coordinates on the

y-axis are 1, 2, and so on. This may be helpful in specifying the coordinates of the vertical bars. Figure 4.13 shows a simple example with a phylogeny of bird orders; the commands used were:

```
plot(bird.orders, font = 1, x.lim = 40,
     no.margin = TRUE)
segments(38, 1, 38, 5, lwd = 2)
text(39, 3, "Proaves", srt = 270)
segments(38, 6, 38, 23, lwd = 2)
text(39, 14.5, "Neoaves", srt = 270)
```

Some arguments are obviously repeated in the successive calls to `segments` and `text`: they are the coordinates of the plotted objects. These calls may be grouped in a single one (e.g., `text(rep(39, 1), c(3, 14.5), c("Proaves", "Neoaves"), srt = 270)`; they were kept distinct for clarity.

Colors are interesting for showing clades, because this can be somewhat automated in R, and thus used for exploratory graphical analyses. In `plot.phylo`, the options `edge.color` and `edge.width` allow us to specify the color and width of each branch of the tree. For instance, `edge.color = "blue"` will color all edges in blue. As many colors as the number of branches may be specified, the values being possibly recycled: `edge.color = c("blue", "red")` will color the first, third, ..., branches in blue, and the second, fourth, ..., in red. The problem is to know the numbers of the branches. This may be easy with a small `"phylo"` object by printing it and then visually finding the number of each branch. However, this may be more difficult with large trees. The function `which.edge` may be used here because it returns the indices of the branches that belong to a specified group. The latter may be not monophyletic in which case the indices will include branches up to the most recent common ancestor of the group. For instance, using the same bird phylogeny:

```
> wh <- which.edge(bird.orders, 19:23)
> wh
 [1] 31 35 37 38 39 40 41 42 43 44
```

It is now easy to define a vector of colors to be used in `plot.phylo`. We first repeat a default color (say black) with as many branches as in the tree:

```
colo <- rep("black", dim(bird.orders$edge)[1])
```

The command `dim(...)[1]` extracts the number of rows in the element `edge` of the tree:[2] we now have a vector with 45 repetitions of `"black"`. The colors of the clades defined above (tips 19–23) are simply modified with:

```
colo[wh] <- "grey"
```

[2] This could be done with `length(bird.orders$edge.length)`, but this will not work if the tree has no branch length.

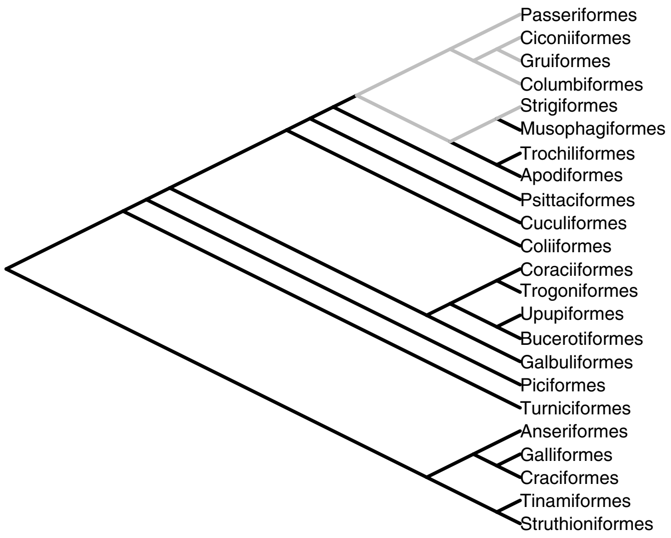


Fig. 4.14. Simple edge colors

The tree can now be drawn. We use wider lines to display the difference in colors better (Fig. 4.14):

```
plot(bird.orders, "c", FALSE,  font = 1, edge.color = colo,
     edge.width = 3, no.margin = TRUE)
```

Showing a clade with a frame or an ellipse is not so easy because if the contour is added after the tree is plotted, it will overlap the latter and hide a portion of it if a colored background is chosen. An obvious solution is to plot a contour without background (which is the default in most functions in R). For instance, with the bird phylogeny, if we want a rectangle showing the clade of the first five orders, we could do:

```
plot(bird.orders, font = 1)
rect(1.2, 0.5, 36, 5.4, lty = 2)
```

By default, the lines of the rectangle are the same as those of the tree edges, hence it may be good to distinguish them with the usual options (`lty = 2` specifies dashed lines). The numeric arguments to `rect` give the position of the leftmost, lower, rightmost, and upper sides of the rectangle. Those can be obtained with the `locator` function which returns the coordinates on the current R plot of points indicated by the user with a pointer (usually the mouse of the computer).

A less straightforward, but maybe more efficient, solution is to edit the code of `plot.phylo`, and add the above call to `rect` just after the call to `plot`. This will draw the rectangle before the tree. A possible set of commands may be (Fig. 4.15):

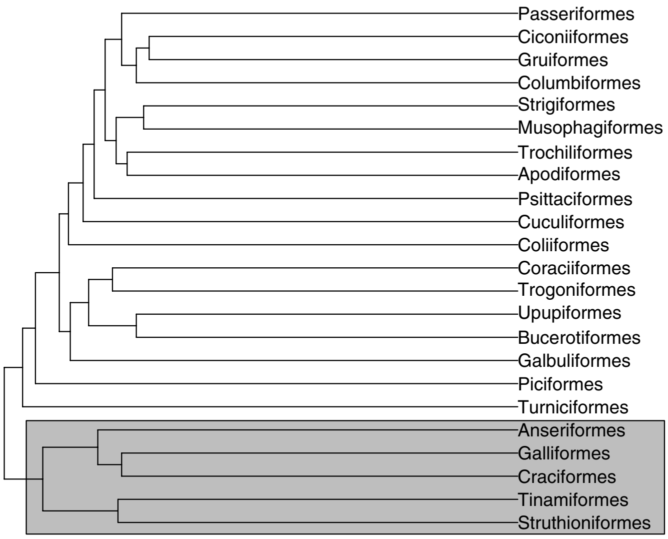


Fig. 4.15. A framed clade

```
fix(plot.phylo)
## add rect(1.2, 0.5, 36, 5.4, col = "lightgrey")
##   just after plot(0, ....)
## then save and close the editor
plot(bird.orders, font = 1, no.margin = TRUE)
```

Note that the modifications done by `fix` alter only the functions loaded in memory, not the ones on the disk. Thus the original functions are restored when R is closed.

4.2 Combining Plots

It may be enlightening to combine several plots in a single figure. This may be needed to indicate the distribution of some variables among recent species (represented by the tips of the tree). ape has no special function to combine trees with other plots: this must be done with standard R functions. ade4 has a few special functions to plot variables in the face of the tips of a tree. Let us first see what can be done with them.

If a variable must be plotted facing the tips of the tree, `symbols.phylog` or `dotchart.phylog` can be used. To illustrate them, first convert the class of our owl tree, and create a vector `x` with the mean body length (in cm) of these four species; the plots are then made (Fig. 4.16):

```
tg <- newick2phylog(write.tree(tree.owls))
```

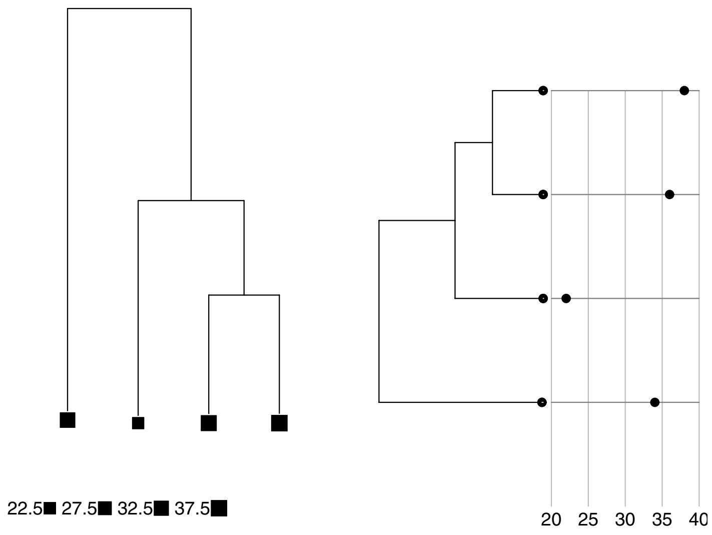


Fig. 4.16. The functions `symbols.phylog` (left) and `dotchart.phylog` (right)

```
x <- c(38, 36, 22, 34)
symbols.phylog(tg, squares = x)
dotchart.phylog(tg, x)
```

`table.phylog` is a multivariate version of `symbols.phylog`: the tree is plotted horizontally facing a matrix with symbols representing the variables arranged in columns. It is preferable that the variables are on the same scale.

To have a more flexible way of plotting variables, one can use `plot.phylo` and manually add further graphical elements. It is useful to know here that when plotting a phylogram or a cladogram, the tips have the coordinates 1, 2, and so on (whatever the direction). It is thus possible to add, for instance, horizontal bars after leaving extra space with `x.lim` (or `y.lim` if the tree is vertical). We could, for instance, plot the species richness of each avian order in the face of the corresponding phylogeny. We have the vector `Orders.dat` with names set as the orders:

```
> Orders.dat <- scan()
1: 10 47 69 214 161 17 355 51 56 10 39 152
13: 6 143 358 103 319 23 291 313 196 1027 5712
24:
Read 23 items
> names(Orders.dat) <- bird.orders$tip.label
> Orders.dat
Struthioniformes     Tinamiformes      Craciformes
              10               47               69
```

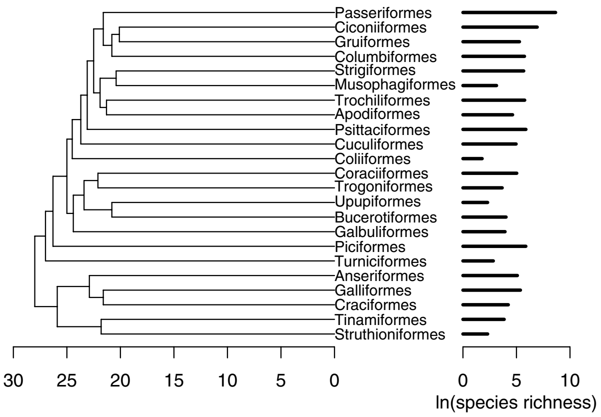


Fig. 4.17. Bars in the face of a tree plotted with `plot.phylo`

```
    Galliformes     Anseriformes    Turniciformes
            214              161               17
     Piciformes    Galbuliformes   Bucerotiformes
            355               51               56
    Upupiformes    Trogoniformes    Coraciiformes
             10               39              152
    Coliiformes     Cuculiformes   Psittaciformes
              6              143              358
    Apodiformes   Trochiliformes  Musophagiformes
            103              319               23
   Strigiformes    Columbiformes       Gruiformes
            291              313              196
  Ciconiiformes    Passeriformes
           1027             5712
```

Fortunately, the data are in the same order as in the tree.[3] We can thus proceed in a straightforward manner (Fig. 4.17):

```
plot(bird.orders, x.lim = 50, font = 1, cex = 0.8)
segments(rep(40, 23), 1:23, rep(40, 23) +
         log(Orders.dat), 1:23, lwd = 3)
axis(1, at = c(40, 45, 50), labels = c(0, 5, 10))
mtext("ln(species richness)", at = 45, side = 1, line = 2)
```

[3] If they were not in the correct order, the names would solve this easily with `Orders.dat[bird.orders$tip.label]`.

```
   axisPhylo()
```

Once we have determined that the bars will span between 40 and 50 on the horizontal scale (which could be done by examining the default `x.lim` of `plot.phylo`), it is easy to set the other values in the command. Note how we draw a 'custom' scale on the x-axis. We did not use `no.margin = TRUE` to leave some space for the scales under the plot.

In the examples we have seen above, the different graphics were plotted in the same plotting region. It is possible to plot different graphs on the same graphical device. This is usually done by splitting the graphical device (i.e., the window or the file) in several regions then calling successively different high-level plotting functions. The most useful approach is to use the function `layout`. The main argument of this function is a matrix with integer numbers indicating the numbers of the 'subwindows'. For instance, to divide the device into four equal parts:

```
> layout(matrix(1:4, 2, 2))
```

Printing the matrix makes clear how the device is divided:

```
> matrix(1:4, 2, 2)
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

The first graph will be plotted in the top-left quarter, the second in the bottom-left quarter, the third in the top-right quarter, and the fourth in the bottom-right quarter. Whereas with:

```
> matrix(c(1, 1, 2, 3), 2, 2)
     [,1] [,2]
[1,]    1    2
[2,]    1    3
```

the first graph will span the left half of the device, and the second and third ones will be in the top-right and bottom-right quarters, respectively. Quite a large number of graphs can be plotted on the same device, for instance 16 with:[4]

```
>  matrix(1:16, 4, 4)
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
```

[4] It may happen that R cannot plot the graphs if there is not enough space in the plotting region.

The `layout` function gives a lot of possibilities. To illustrate this, we consider plotting two trees of the same species but showing different information. Let us come back to the *Apodemus* data (Fig. 4.1). Michaux et al. [100] estimated divergence dates on their tree using a molecular clock. The tree on Fig. 4.1 could also be analyzed with the nonparametric rate smoothing method of Sanderson [135] using the calibration point of 12 Ma (million years ago) for the divergence *Mus*/*Rattus*. This is done with the function `chronogram` (Section 5.4). We can proceed very easily by reading the clock tree of Michaux et al., computing the chronogram, splitting the graphical device in two, and finally plotting both trees successively. The set of needed commands is straightforward:

```
trk <- read.tree("Apodemus_molclock.tre")
trc <- chronogram(tr, scale = 12)
layout(matrix(1:2, 1, 2))
plot(trk)
plot(trc, show.tip.label = FALSE, direction = "l")
```

The figure obtained this way will not display the information nicely because of the default margins which are too wide here. We need a little extra work to make the figure informative. We first change the tip labels of the first tree to replace the genus names with their initials. This could be done manually by editing `trk$tip.label` and replacing `"Apodemus_agrarius"` with `"A. agrarius"`, and so on. Fortunately, R has functions that manipulate regular expressions which considerably facilitates this kind of task. Here we use the function `gsub` (*global substitution*), for instance:

```
trk$tip.label <- gsub("Apodemus", "A.", trk$tip.label)
```

will replace every occurrence of `"Apodemus"` by `"A."`. We could do this for the five genera in the tree but this is still tedious, and there is a more general solution:

```
trk$tip.label <- gsub("[[:lower:]]{1,}_", "._", trk$tip.label)
```

The regular expression `"[[:lower:]]{1,}_"` means "one or more lowercase letter(s) followed by an underscore". We clearly take advantage of the fact that the genus and species names are separated by this last character.

We can now plot the trees but we need to care about the space around both. Let us first see the whole commands, then explain what has been done. The resulting plot is in Fig. 4.18.

```
layout(matrix(1:2, 1, 2), width = c(1.4, 1))
par(mar = c(4, 0, 0, 0))
plot(trk, adj = 0.5, cex = 0.8, x.lim = 16)
nodelabels(node = 12, "?", adj = 2, bg = "white")
axisPhylo()
```

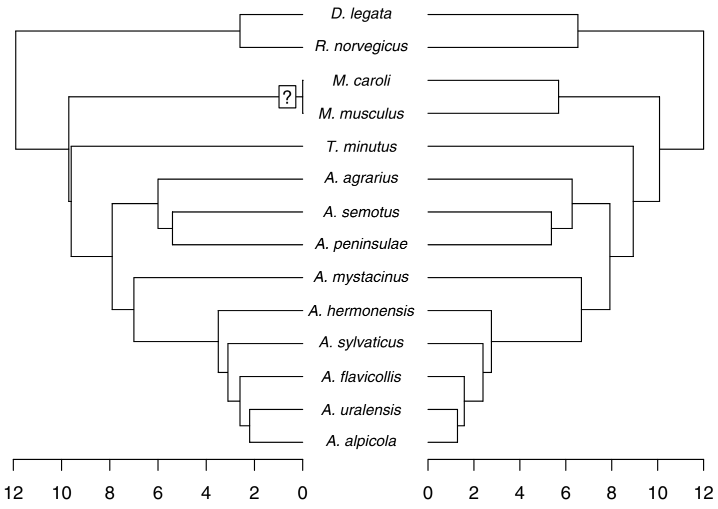


Fig. 4.18. Facing trees

```
plot(trc, show.tip.label = FALSE, direction = "l")
axisPhylo()
```

The critical options are `width` for `layout` and `x.lim` for `plot`: they allow us to have both trees of the same size on the figure. These commands will work for any other data providing these two options are set correctly. Note that we remove the space around the trees except that below, so we cannot use the option `no.margin` of `plot.phylo`: instead we use the `par` function. The call to `nodelabels` is to indicate that one node (the divergence between the two species of *Mus*) was not dated by Michaux et al. [100]. Finally, we draw the axis below each tree using `axisPhylo`.

Note the possibility with `layout` of inserting a graph within a larger one. In principle the different subwindows are completely independent, but if one of them is surrounded by another, then the graph in the first will overlap with the second. For instance, with the following matrix given as argument to `layout`:

```
matrix(c(2, 1, 1, 1), 2, 2)
     [,1] [,2]
[1,]    2    1
[2,]    1    1
```

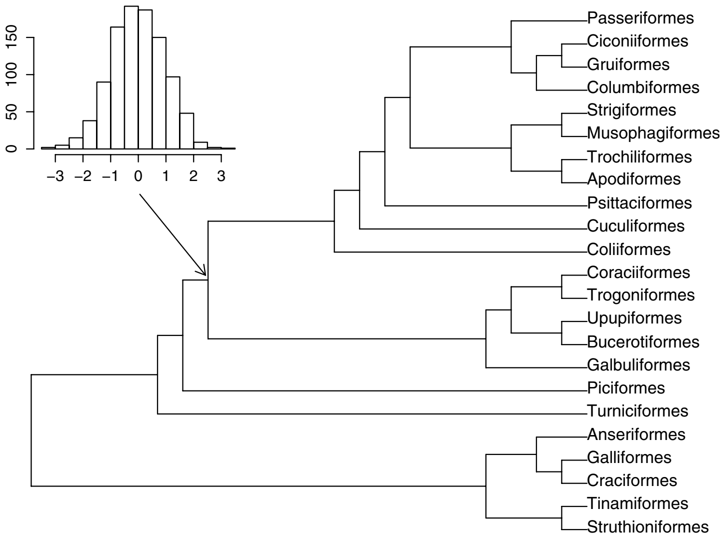


Fig. 4.19. Insert an histogram

the first graph will be plotted on the whole graphical device, and the second one will be on the top-left quarter, thus potentially partially overlapping the first one. To further reduce the size of the insert, one could do:[5]

```
layout(matrix(c(2, rep(1, 8)), 3, 3))
```

Here is an example of how this could be used (Fig. 4.19):

```
plot(bird.orders, "p", FALSE, font = 1,
     no.margin = TRUE)
arrows(4.3, 15.5, 6.9, 12, length = 0.1)
par(mar = c(2, 2, 0, 0))
hist(rnorm(1000), main = "")
```

4.3 Large Phylogenies

Large trees are a puzzle for phylogeneticists because trees are themselves ways to summarize the relationships among species and other taxonomic units, but when they reach a certain size, the information that was supposed to be summarized is likely to be no more visible. The recent literature has seen the definition of a terminology about "large trees", "very large trees", and even "huge trees" reaching tens of thousands of tips, but it is clear that even a

[5] `layout` has options `width` and `height` to modulate the sizes of the subwindows in a more flexible way than done here.

tree with a few hundred tips may hide the phylogenetic information that was originally sought.

Large trees have become an issue with the availability of larger and larger molecular databases such as GenBank, and the development of ambitious projects to assemble the tree of life. Large trees are also becoming present in fields such as genomics where a single experiment can result in thousands of observations.

The general strategy to visualize a large tree is to plot only a portion of the full phylogeny, while indicating its context, that is, how it relates to the rest of the tree.

We show that most of the necessary ingredients to visualize and explore large trees are present in various functions in `ape`. `plot.phylo` and `drop.tip` may be used in conjunction with R's functions `layout` and `X11` to give a powerful and flexible environment for the graphical exploration of phylogenies. One function in `ape`, `zoom`, integrates these ideas to give an automated way to explore large trees.

We have seen that `drop.tip` removes some terminal branches from a `"phylo"` object, and eventually trims the corresponding internal branches. It is thus possible to use this function to extract a subtree by passing all but the wanted tips as argument. If one has the numbers of the wanted tips, say in a vector `x`, this can be done with:

```
drop.tip(tr, tr$tip.label[-x])
```

Alternatively, if `x` is a vector with the labels of the tips to be kept, one could do:

```
drop.tip(tr, which(!tr$tip.label %in% x))
```

The expression `tr$tip.label %in% x` returns a logical value for each tip label: it is `TRUE` if the label is in `x`, `FALSE` otherwise. The operator `!` inverts these logical values, and the function `which` returns the indices of those that are `TRUE`.

Thus the action of `drop.tip` is quite straightforward, but it may be useful to show in some way the relationship of the returned subtree with the original tree. This can be done with the option `subtree` which takes a logical value. If it is `TRUE` (the default is `FALSE`), a branch is included in the returned tree that shows how many tips have been deleted in the operation; this is done for as many monophyletic groups as have been removed.

Let us see how this works with a supertree of the mammal order Chiroptera [76]. Our goal is to extract a subtree with the first 15 tips. The tree has 921 tips, thus the second argument to `drop.tip` could either be `16:921` or `chiroptera$tiplabel[-(1:15)]` with exactly the same result. We then plot the extracted tree (Fig. 4.20). The three commands are:

```
data(chiroptera)
tr <- drop.tip(chiroptera, 16:921, subtree = TRUE)
```

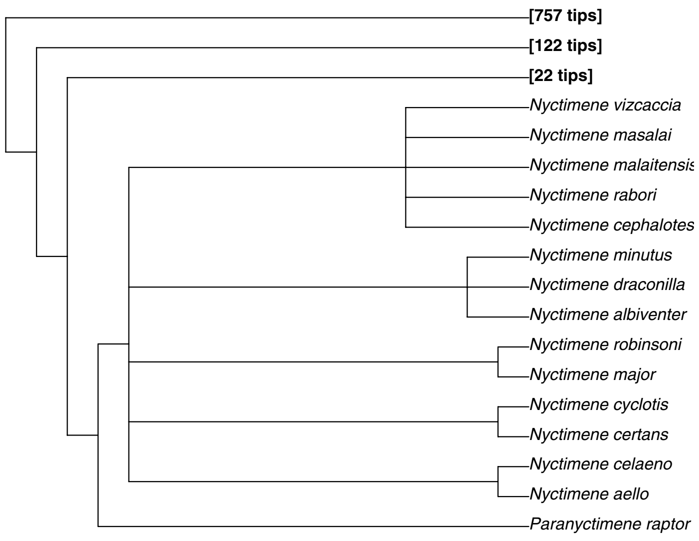


Fig. 4.20. Extracting a subtree

```
plot(tr, font = c(rep(3, 15), rep(2, 3)), cex = 0.8,
     no.margin = TRUE)
```

Note how we specified the `font` argument to have only the species names in italics.

`drop.tip` can thus be used to explore large trees. One can use `layout`, as we have seen above, to plot the whole tree and a subtree on the same device. Another possibility is to open another device and plot the whole tree and the subtrees on the different devices. For instance, to explore the bat supertree, the following commands can be used.

```
plot(chiroptera)
X11()
plot(tr)
```

This will open a second graphical window, and plot the extracted subtree. Because this second window is the active device, all subsequent graphics will be plotted in it.[6]

`zoom` is a function that allows exploration of large trees in a more user-friendly way. Its principle is to plot the whole tree in the left third of the device, and one or several subtrees in the remaining portion of the device. The locations of the subtrees are indicated with colors on the whole tree. The subtree(s) is (are) specified in the same way as in `drop.tip`. There are two options: `subtree` which has the same effect as in `drop.tip`, and `col`

[6] See `?dev.list` on how to set the priority of graphical devices.

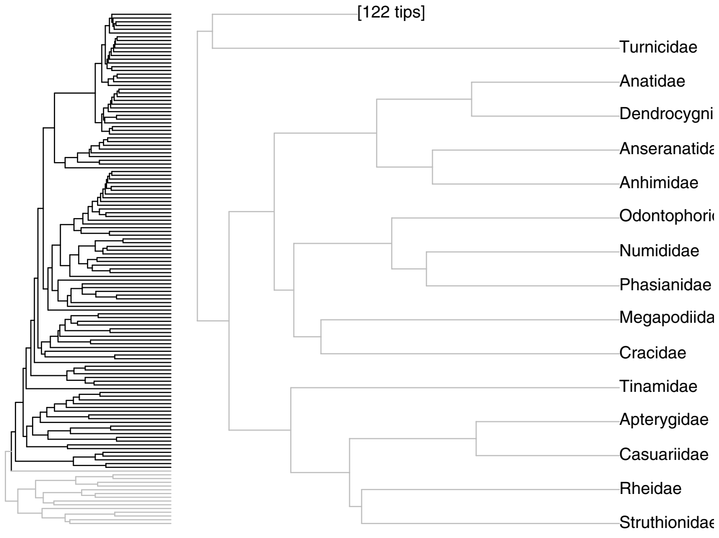


Fig. 4.21. Using `zoom`

which indicates the colors to be used. By default, a preset rainbow palette is used. Any further argument recognized by `plot.phylo` (see Table 4.1) may be passed thanks to the "dot-dot-dot" argument (see p. 71).

A simple example of the use of `zoom` could be (Fig. 4.21):

```
data(bird.families)
zoom(bird.families, 1:15, col = "grey", no.margin = TRUE,
     subtree = TRUE)
```

We have set `subtree = TRUE` (the default is `FALSE`) to show the context of the specified subtree, and `no.margin = TRUE` (which is passed to `plot.phylo` as part of the "dot-dot-dot" argument) to use as much space as available on the device.

If several subtrees need to be visualized on the same plot, they have to be specified as a list (because they could differ in size). For instance (Fig. 4.22),

```
zoom(bird.families, list(1:15, 38:48), col = rep("grey", 2),
     no.margin = TRUE, font = 1, subtree = TRUE)
```

Here we have used the same grey color for both subtrees, but by default red and cyan (green-blue) are used.

4.4 Perspectives

The graphical analysis and exploration of phylogenies are in their early days. There is undoubtedly much to expect from research in this area. With the now

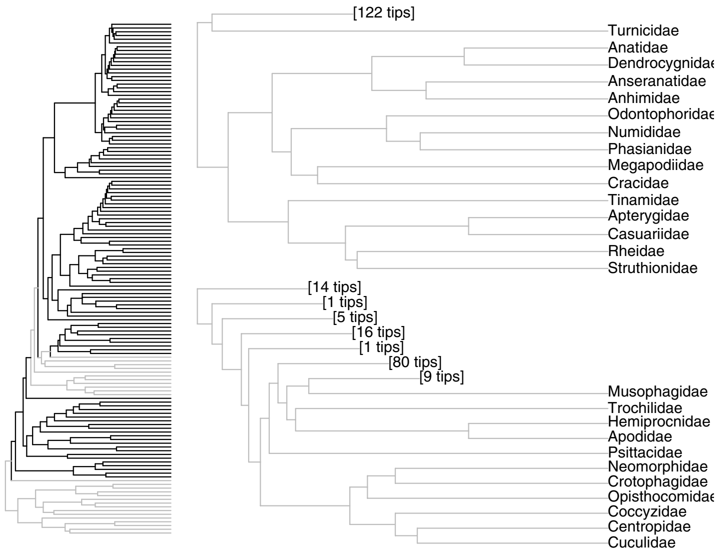


Fig. 4.22. Using `zoom` to show two groups

widespread availability of powerful computers, it will be possible to explore and analyze large phylogenies in a flexible way. Future developments will need to take care of integration with other tools, and operability for the interchange of information among different systems.

The examples presented in this chapter all use the graphics package of R which is the default graphical environment of R. Future developments may consider instead the grid package developed by Paul Murrell. This is a reimplementation of R's graphical environment with greater performance and flexibility. Among the improvements are:

- Graphical objects are editable and can be modified without redrawing the whole plot;
- Plots may be arranged in many ways (rotated, scaled, overlapping, etc.);
- The user can "navigate" among plots;
- Graphical objects may be shared among plots.

Using grid clearly needs further development but some of the existing codes in ape can be reused directly (such as the functions that compute the coordinates of the edges of the tree). Another example of a potentially useful development is the use of the 3-D graphical libraries OpenGL which is already interfaced with R via the package rgl. I have already conducted some experiments with both grid and rgl demonstrating the ease of such adaptations. The issue now is to determine which tools need to be developed on these environments.

4.5 Exercises

1. Draw Fig. 4.11 using a color scale in place of the grey one. The figure should include a legend.

2. Plot the phylogeny of avian orders, and color the Proaves in blue. Repeat this but only for the terminal branches of this clade.

3. Suppose you have a factor, say representing a character state, for each node and each tip of a tree. Find a way to associate a color with each branch depending on the state at both ends of the branch.

5 Phylogeny Estimation

Reconstructing the evolutionary relationships among living species is one of the oldest problems in biology. It has clearly enjoyed an increasing interest as witnessed by the reviews published in the last few years [4, 12, 68, 69, 155]. There have been some real advances during the past two decades, but several difficulties remain.

- The estimation of phylogenies is a computationally hard problem which is analytically intractable in the general case [19].
- Realistic models of character evolution involve many parameters, and it is likely that real processes are much more complex than the most complex models available in the literature.
- A common biological complication is that the species and the characters under study do not have the same history; this is particularly the case for genetic data [4].
- It is often necessary to estimate many parameters simultaneously but only some of them are of interest [68].
- There is some confusion in the use of some terminology related to estimation and statistics that is likely to reveal difficulties in communicating across different scientific fields [69].
- Some confusion arises because phylogeny estimation methods are also used for systematics (i.e., classification of species) rather than estimating evolutionary parameters.
- Many studies assessed the "performance" of phylogenetic methods using simulations but these considered only special cases, and the conclusions drawn from these simulations are of very limited value [69].
- The different methods, models, and algorithms for phylogeny estimation are available in distinct programs resulting in several practical difficulties.

The last point is of particular interest here. All these programs have their own features and requirements in terms of operating systems, user interfaces, data formats, or licenses. Many of them are not free. Comparing different methods is difficult because it is often hard to decide whether the observed differences in the results are due to different assumptions, algorithms, run-time environments, computer architectures, or other features that vary among programs. Even the analysis of a single data set is made difficult by the need to switch between different software and / or operating systems.

The development of phylogeny estimation in R is very new, and some progress has been made in distance-based and maximum likelihood methods. This is limited compared to the methods available in the literature (particularly with respect to the old, well-established parsimony methods, and the current success of Bayesian methods). There are good reasons to focus on distance and likelihood methods, because these methods have been shown to perform well in a number of situations (although we have to be cautious in generalizing these conclusions as mentioned above). There has been a long-lasting debate on the merits of parsimony, and although this method has been severely criticized [37], it can be viewed as a valid nonparametric method [69]. Bayesian methods enjoy a current success, but some critics pointed out the limitations of this approach [39, 148]. However, Bayesian phylogeny estimation may be implemented in a straightforward way because all the necessary ingredients exist in R or have been developed in various packages.

5.1 Distance Methods

Distance methods have a long history because in their simplest formulation they are generally tractable even with a large amount of data [152]. I concentrate on only two methods: UPGMA and neighbor-joining. The first section deals with how to compute distances in R.

5.1.1 Calculating Distances

There is a difference between the concepts of statistical and evolutionary distances. In statistics, a distance can be viewed as a “physical” or geometric distance between two observations, each variable being a dimension in a hyperspace. In evolutionary biology, a distance is an estimate of the divergence between two units (individuals, populations, or species). This is usually measured in quantity of evolutionary change (e.g., numbers of mutations).

R has various functions to compute distances available in different packages. Table 5.1 lists these functions, which are detailed in the following sections.

Classical Distances

R has a rich set of methods to compute classical distances. `dist` in package `stats` performs distance calculations taking a matrix as its main argument. Its main option is `method` which can take one of the six following

Table 5.1. Functions for computing distances in R

Package	Function	Data Types
stats	`dist`	Continous or binary
	`cophenetic`	Objects of class `"hclust"` or `"dendrogram"`
cluster	`daisy`	Continuous and / or discrete
ade4	`dist.binary`	Binary
	`dist.prop`	Relative frequencies
	`dist.genet`	An object of class `"genet"`
ape	`dist.gene`	Discrete
	`dist.dna`	Aligned DNA sequences
	`weight.taxo`	'Taxonomic' levels
	`cophenetic`	An object of class `"phylo"`

strings: `"euclidean"` (the default), `"maximum"`, `"manhattan"`, `"canberra"`, `"binary"`, or `"minkowski"`. As a simple example:

```
> X <- matrix(rep(c(0, 1, 5), 3), 3)
> rownames(X) <- LETTERS[1:3]
> X
  [,1] [,2] [,3]
A    0    0    0
B    1    1    1
C    5    5    5
> dist(X)
         A        B
B 1.732051
C 8.660254 6.928203
> dist(X, method = "maximum")
  A B
B 1
C 5 4
> dist(X, method = "manhattan")
   A  B
B  3
C 15 12
```

`dist` returns an object of class `"dist"` which is a vector storing only the lower triangle of the distance matrix (because it is symmetric and all its diagonal elements are equal to zero). These objects can be converted to matrices using the generic function `as.matrix`, and matrices can be converted with `as.dist`:

```
> d <- dist(X)
> class(d)
[1] "dist"
> as.matrix(d)
```

```
         A        B        C
A 0.000000 1.732051 8.660254
B 1.732051 0.000000 6.928203
C 8.660254 6.928203 0.000000
```

The function `daisy` in the package `cluster` also performs distance calculations but it implements some methods that can deal with mixed data types. Two metrics are available via the option `metric`: `"euclidean"` or `"manhattan"`. The data types are specified with the option `type`.

Evolutionary Distances

`ape` has two functions to calculate evolutionary distances: `dist.gene` and `dist.dna`. They handle allelic data and DNA sequences, respectively. Additionally `ade4` has the function `dist.genet` that computes distances between populations using allele frequency data

`dist.gene` provides a simple interface to compute the distance between two haplotypes using a simple binomial distribution of the pairwise differences. This allows us to compute easily the variance of the estimated distances with the expected variance of the binomial distribution. The input data are a matrix or a data frame where each row represents a haplotype, and each column a locus.

`dist.dna` provides a comprehensive function for the estimation of distances from aligned DNA sequences using substitution models (Table 5.2). If a correction for among-sites heterogeneity (usually based on a Γ distribution) is available, this may be taken into account. The variances of the distances can be computed as well.

`dist.genet` takes as input the allele frequencies from one or several loci, and computes the distances between populations. The data must be a list of class `"genet"`. Such a list may be obtained from a matrix with the function `char2genet` (see the help of this function for details). Five methods are available to compute these distances: standard (or Nei's), angular (or Edwards's), Reynolds's, Rogers's, and Provesti's. This is specified with the option `method` which takes an integer value between 1 and 5.

By contrast to `dist.gene`, `dist.dna` and `dist.genet` return an object of class `"dist"`.

Special Distances

The package `ade4` has two functions that compute distances with some special types of data: `dist.binary` and `dist.prop`, for binary data and proportions, respectively. The first one has the option `method` which takes an integer between 1 and 10; this includes the well-known Jaccard, and the Sokal and Sneath methods. The second function has a similar option taking an integer between 1 and 5; this includes Rogers's, Nei's, and Edwards's methods.

Table 5.2. Options of the function `dist.dna`

Options	Effect	Possible Values
`model`	Specifies the substitution model	`"raw"`, `"JC69"`, `"K80"` (d), `"K81"`, `"F81"`, `"F84"`, `"T92"`, `"TN93"`, `"GG95"`
`variance`	Whether to compute the variances	`FALSE` (d), `TRUE`
`gamma`	The value of α for the Γ correction	`NULL` (no correction) (d), a numeric giving the value of α
`pairwise.deletion`	Whether to delete the sites with missing data in a pairwise way	`FALSE` (d), `TRUE`
`base.freq`	The frequencies of the four bases	`NULL` (calculated from the data) (d), four numeric values
`as.matrix`	Whether to return the results as a matrix or as an object of class `"dist"`	`TRUE` (d), `FALSE`

`ape` has the function `weight.taxo` that computes a similarity matrix between observations characterized by categories that can be interpreted as a taxonomic level (i.e., a numeric code, a character string, or a factor). The value is 1 if both observations are identical, 0 otherwise.

Finally, `stats` has a generic function `cophenetic` that computes the distances among the tips of a hierarchical data structure: there are methods for objects of class `"hclust"`, `"dendrogram"`, and `"phylo"`.

5.1.2 Simple Clustering and UPGMA

There is a corpus of phylogeny estimation methods that are based on statistical clustering methods. They were popular in the past, but have recently declined since the rise of likelihood and Bayesian methods. These methods are limited, mostly because of their assumption of constant rates of evolution [106]. We do not consider them in detail, but using these methods is a nice illustration of how different functions from different packages in R can interact simply.

R has a reasonably large number of functions that perform clustering [154]. They mostly work on a distance (also called dissimilarity) matrix, but some of them work directly on the original data matrix (observations and variables). Remarkably, a tree estimated with the unweighted pair-group method using arithmetic average (UPGMA) is built in exactly the same way as a hierarchical clustering with the average method. Thus such a tree can be estimated in a straightforward way, for instance, from a set of DNA sequences named `X` with:

```
M <- dist.dna(X)
```

```
hc <- hclust(M, "average")
tr <- as.phylo(hc)
```

The substitution model can be changed with the appropriate option in dist.dna. Giving the graphical functions detailed in the previous chapter, it is easy to compare the trees estimated with different substitution models; for instance:

```
M1 <- dist.dna(X)
tr1 <- as.phylo(hclust(as.dist(M1), "average"))
M2 <- dist.dna(X, model = "F84")
tr2 <- as.phylo(hclust(as.dist(M2), "average"))
layout(matrix(1:2, 2, 1))
plot(tr1, main = "Kimura (80) distances")
plot(tr2, main = "Felsenstein (84) distances")
```

We show some practical examples in Section 5.5.

5.1.3 Neighbor-Joining

The neighbor-joining (NJ) method is a fast and straightforward method for estimating a phylogenetic tree from a distance matrix [134]. Its principle is to construct a tree by successive pairing of taxons (the neighbors): the pair that leads to the tree with the smallest total branch length is selected. The procedure is iterated until the tree is dichotomous.

ape has the function nj that performs the NJ algorithm. Its use is extremely simple: it takes a distance matrix as unique argument, and returns the estimated tree as an object of class "phylo". As for the UPGMA, it is easy to obtain NJ trees with different substitution models. It is also possible to call nj repeatedly for a series of models:

```
mod <- list("JC69", "K80", "F81", "F84")
lapply(mod, function(m) nj(dist.dna(X, model = m)))
```

In the above command, we insert the call to dist.dna with the call to nj in a function where the model is treated as a variable. lapply then dispatches the different models to this function, and returns the results as a list.

A strength of the NJ method is that it is fast [152], even with large sample sizes, both in terms of number of tips (which is dealt with by the NJ method) and in terms of number of sites (which is dealt with by the distance computation methods).

5.2 Maximum Likelihood Methods

Maximum likelihood is the cornerstone of modern statistics [27, 30]. The two critical ingredients in estimating a phylogeny by maximum likelihood are:

- A parametric model of evolution appropriate for the characters;
- An algorithm that will search through the trees in order to find the maximum likelihood one.

All the other ingredients (deriving the probability distribution of the data and the likelihood function, etc.) are somewhat straightforward. The model chosen depends essentially on the nature of the characters under study. Among the many possible models of character evolution, those commonly used fall into two categories: Markovian and Brownian. Markovian models are appropriate for modeling the evolution of discrete characters, whereas Brownian ones are more appropriate for continuous characters.

5.2.1 Substitution Models: A Primer

The vast majority of models of evolution for discrete characters are Markovian implying that:

- The number of character states is finite;
- The probabilities of transitions among these states are controlled by some parameters;
- The process is at equilibrium.

This can be applied to many kinds of data [110], but the recent rise of large-scale molecular databases has led to this approach being applied essentially to nucleotide (DNA) and protein sequences. An intermediate kind of data often considered for coding nucleotide sequences is based on codons.

A substitution model is a formulation of the instantaneous rates of change among the different states of the character. For instance, for a character with two states, A and B, where the rate of change (i.e., the probability of change from one state to another for a very short time) is symmetric and equal to 0.1, the *rate matrix*, usually denoted Q, is:

$$Q = \begin{bmatrix} -0.1 & 0.1 \\ 0.1 & -0.1 \end{bmatrix} . \tag{5.1}$$

The rows of Q correspond to the initial state, and its columns to the final one. The elements on the diagonal are set so that the sum of each row is zero. For an arbitrary time interval t, the *probability matrix* P is obtained by the matrix exponentiation of Q:

$$P = \mathrm{e}^{tQ} . \tag{5.2}$$

The element p_{ij} from the ith row and jth column of P is the probability of being in state j after time t giving that the initial state was i. The probabilities in P take into account possible multiple changes (e.g., a change from A to B may be the result of A $\rightarrow$ B, or A $\rightarrow$ B $\rightarrow$ A $\rightarrow$ B, ...). The matrix exponentiation is usually calculated with an infinite sum:

$$\mathrm{e}^{tQ} = I + tQ + \frac{(tQ)^2}{2!} + \frac{(tQ)^3}{3!} + \cdots \tag{5.3}$$

$$= I + \sum_{i=1}^{\infty} \frac{(tQ)^i}{i!} \ . \tag{5.4}$$

In practice, an approximation is done. Several functions in R perform matrix exponentiation. We use mexp in the package rmutil:

```
> library(rmutil)
> Q <- matrix(c(-0.1, 0.1, 0.1, -0.1), 2)
> Q
     [,1] [,2]
[1,] -0.1  0.1
[2,]  0.1 -0.1
> mexp(Q) # t = 1
           [,1]       [,2]
[1,] 0.90936538 0.09063462
[2,] 0.09063462 0.90936538
> mexp(10*Q) # t = 10
          [,1]      [,2]
[1,] 0.5676676 0.4323324
[2,] 0.4323324 0.5676676
```

We effectively have probabilities because the rows sum to one. Note that Q is independent of time whereas P is not. Both calculated matrices are symmetric; they would be asymmetric if Q were.

When fitting a substitution model to some data, its parameter(s) will usually be unknown. For the hypothetical two-states character we write:

$$Q = \begin{bmatrix} . & \alpha \\ \alpha & . \end{bmatrix} , \tag{5.5}$$

where α is the parameter and the dots on the diagonal indicate that these values are set so that the rows sum to zero.

This methodology is generalized to DNA sequences (by assuming that Q is 4×4), to protein sequences (20×20), and codons (64×64). The substitution models differ in the way the rate matrix Q is modeled. We consider here in detail the case of DNA sequences because substitution models for this kind of data are implemented in several functions in ape.

For the simplest models of DNA substitution, it is possible to derive the transition probabilities (i.e., the elements of P) without matrix exponentiation: this is nicely explained by Felsenstein [39, p.156]. In the following, each model is cited, the character code used in ape is given, and the model is briefly described.

Jukes and Cantor 1969 (`"JC69"`)

This is the simplest model of DNA substitution [77]. The probability of change from one nucleotide to any other is the same. It is assumed that all four bases have the same frequencies (0.25). The rate matrix Q is:

$$\begin{array}{c|cccc} & \mathrm{A} & \mathrm{G} & \mathrm{C} & \mathrm{T} \\ \mathrm{A} & . & \alpha & \alpha & \alpha \\ \mathrm{G} & \alpha & . & \alpha & \alpha \\ \mathrm{C} & \alpha & \alpha & . & \alpha \\ \mathrm{T} & \alpha & \alpha & \alpha & . \end{array} \; .$$

As with the general case above, the rows correspond to the original state of the nucleotide, and the columns to the final state (the row and column labels are omitted in the following models).

The overall rate of change in this model is thus 3α. The probability of change from one base to another during time t can easily be derived (see [39]):

$$p_{ab}(t) = (1 - \mathrm{e}^{-4\alpha t})/4 \qquad a \neq b\,, \tag{5.6}$$

where a and b are among A, G, C, and T.

The expected mean number of substitutions between two sequences is $3(1 - \mathrm{e}^{-4\alpha t})/4$ because there are three different types of change. From this, it is straightforward to derive an estimate of the distance.

This model is available in `dist.dna`, `mlphylo`, and `phymltest`.

Kimura 1980 (`"K80"`)

Because there are two kinds of bases with different chemical structures, purines (A and G) and pyrimidines (C and T), it is likely that the changes within and between these kinds are different. Kimura [81] developed a model whose rate matrix is:

$$\begin{bmatrix} . & \alpha & \beta & \beta \\ \alpha & . & \beta & \beta \\ \beta & \beta & . & \alpha \\ \beta & \beta & \alpha & . \end{bmatrix} \; .$$

A change within a type of base is called a *transition* and occurs at rate α; a change between types is called a *transversion* and occurs at rate β. The base frequencies are assumed to be equal.

This model is available in `dist.dna`, `mlphylo`, and `phymltest`.

Felsenstein 1981 (`"F81"`)

Felsenstein [34] extended the JC69 model by relaxing the assumption of equal frequencies. Thus the rate parameters are proportional to the latter:

$$\begin{bmatrix} . & \alpha\pi_G & \alpha\pi_C & \alpha\pi_T \\ \alpha\pi_A & . & \alpha\pi_C & \alpha\pi_T \\ \alpha\pi_A & \alpha\pi_G & . & \alpha\pi_T \\ \alpha\pi_A & \alpha\pi_G & \alpha\pi_C & . \end{bmatrix} .$$

There are three additional parameters (the base frequencies, π_A, π_G, π_C, and π_T, sum to one, thus only three of them must be estimated) but they are usually estimated from the pooled sample of sequences.

This model is available in `dist.dna`, `mlphylo`, and `phymltest`.

Kimura 1981 (`"K81"`)

Kimura [82] generalized his model K80 by assuming that two kinds of transversions have different rates: A $\leftrightarrow$ C and G $\leftrightarrow$ T on one side, and A $\leftrightarrow$ T and C $\leftrightarrow$ G on the other.

$$\begin{bmatrix} . & \alpha & \beta & \gamma \\ \alpha & . & \gamma & \beta \\ \beta & \gamma & . & \alpha \\ \gamma & \beta & \alpha & . \end{bmatrix} .$$

This model is available in `dist.dna`.

Felsenstein 1984 (`"F84"`)

This model can be viewed as a synthesis of K80 and F81: there are different rates for base transitions and transversions, and the base frequencies are not assumed to be equal. The rate matrix is:

$$\begin{bmatrix} . & \pi_G(\alpha/\pi_R + \beta) & \beta\pi_C & \beta\pi_T \\ \pi_A(\alpha/\pi_R + \beta) & . & \beta\pi_C & \beta\pi_T \\ \beta\pi_A & \beta\pi_G & . & \pi_T(\alpha/\pi_Y + \beta) \\ \beta\pi_A & \beta\pi_G & \pi_C(\alpha/\pi_Y + \beta) & . \end{bmatrix} ,$$

where $\pi_R = \pi_A + \pi_G$, and $\pi_Y = \pi_C + \pi_T$ (the proportions of purines and pyrimidines, respectively). Felsenstein and Churchill [40] gave formulae for the probability matrix and the distance. This model is available in `dist.dna`, `mlphylo`, and `phymltest`.

Hasegawa, Kishino, and Yano 1985 (`"HKY85"`)

This model is very close in essence to the previous one but its parameterization is different [66]:

$$\begin{bmatrix} . & \alpha\pi_G & \beta\pi_C & \beta\pi_T \\ \alpha\pi_A & . & \beta\pi_C & \beta\pi_T \\ \beta\pi_A & \beta\pi_G & . & \alpha\pi_T \\ \beta\pi_A & \beta\pi_G & \alpha\pi_C & . \end{bmatrix} .$$

Due to some mathematical properties of this rate matrix, it does not seem possible to derive analytical formulae of the transition probabilities, and so for the distance as well [156]. This model is available in `mlphylo` and `phymltest`.

Tamura 1992 (`"T92"`)

The model developed by Tamura [150] is a generalization of K80 that takes into account the content of G + C. The rate matrix is:

$$\begin{bmatrix} . & \alpha\theta & \beta\theta & \beta(1-\theta) \\ \alpha(1-\theta) & . & \beta\theta & \beta(1-\theta) \\ \beta(1-\theta) & \beta\theta & . & \alpha(1-\theta) \\ \beta(1-\theta) & \beta\theta & \alpha\theta & . \end{bmatrix} ,$$

where $\theta = \pi_G + \pi_C$. Tamura [150] gave formulae for the distance, and Galtier and Gouy [44] gave formulae for the transition probabilities. This model is available in `dist.dna` and `mlphylo`.

Tamura and Nei 1993 (`"TN93"`)

Tamura and Nei [151] developed a model where both kinds of base transitions, A $\leftrightarrow$ G and C $\leftrightarrow$ T, have different rates α_R and α_Y, respectively. The base frequencies may be unequal. All the above models can be seen as particular cases of the TN93 model. The rate matrix is:

$$\begin{bmatrix} . & \pi_G(\alpha_R/\pi_R+\beta) & \beta\pi_C & \beta\pi_T \\ \pi_A(\alpha_R/\pi_R+\beta) & . & \beta\pi_C & \beta\pi_T \\ \beta\pi_A & \beta\pi_G & . & \pi_T(\alpha_Y/\pi_Y+\beta) \\ \beta\pi_A & \beta\pi_G & \pi_C(\alpha_Y/\pi_Y+\beta) & . \end{bmatrix} .$$

Fixing $\alpha_R = \alpha_Y$ results in the F84 model, whereas fixing $\alpha_R/\alpha_Y = \pi_R/\pi_Y$ results in the HKY85 model [39]. This model is available in `dist.dna`, `mlphylo`, and `phymltest`.

The "General Time-Reversible" Model ("GTR")

This is the most general time-reversible model. All substitution rates are different, and the base frequencies may be unequal [87]. The rate matrix is:

$$\begin{bmatrix} . & \alpha\pi_G & \beta\pi_C & \gamma\pi_T \\ \alpha\pi_A & . & \delta\pi_C & \epsilon\pi_T \\ \beta\pi_A & \delta\pi_G & . & \zeta\pi_T \\ \gamma\pi_A & \epsilon\pi_G & \zeta\pi_C & . \end{bmatrix} .$$

There are no analytical formulae for the transition probabilities, nor for the distance [39]. This model is available in `mlphylo` and `phymltest`.

Galtier and Gouy 1995 ("GG95")

Galtier and Gouy [43] developed a nonequilibrium model where the G + C content is allowed to change through time. Sequences are assumed to evolve on each lineage depending on its G + C content. This is estimated from the G + C content of the recent species or populations. It is thus necessary to estimate ancestral G + C contents. The rate matrices for each lineage are similar to the one for the T92 model except that θ may vary.

This model is available in `dist.dna`.

5.2.2 Estimation with Molecular Sequences

If the probabilities of change along a tree are known (using one of the models described in the previous section), the likelihood of the tree can be computed. However, the states of the data on the nodes of the tree are unknown, and it is necessary to sum the probabilities for all possible states on the nodes which may involve a very large number of terms even for a moderate data set. Felsenstein [34] presented an algorithm that allows considerable time saving in this computation. The idea is to compute successively the likelihoods of each character state at each node by summing the probabilities giving the likelihoods of the descendants (hence the name "pruning algorithm").

Denote as M the number of states (e.g., $M = 4$ for DNA data), $p_{ab}(t)$ the probability of change from state a to state b during time t. Then the likelihood of state a at node z, given the likelihood of its descendants x and y (assuming a binary tree) and the branch lengths t_{xz} and t_{yz} is:

$$L_{az} = \left(\sum_{b=1}^{M} p_{ab}(t_{xz})L_{bx}\right)\left(\sum_{b=1}^{M} p_{ab}(t_{yz})L_{by}\right) . \tag{5.7}$$

If x is a tip, then $L_{bx} = 1$ if state b is observed, 0 otherwise.

Once this computation has been applied to all nodes of the tree, the likelihood of the character for the tree is obtained by:

$$L = \sum_{a=1}^{M} \pi_a L_{ar} \ , \tag{5.8}$$

where π_a is the frequency of the ath state, and r is the root of the tree. The root can actually be placed on any internal node of the tree because the latter is unrooted [34]. The likelihood of the full data set is:

$$L = \prod_{i=1}^{N} \sum_{a=1}^{M} \pi_a L_{air} \ , \tag{5.9}$$

where N is the number of characters. Taking the logarithm of this expression leads to:

$$\ln L = \sum_{i=1}^{N} \ln \left(\sum_{a=1}^{M} \pi_a L_{air} \right) \ . \tag{5.10}$$

With molecular sequences, a further layer of complexity is added by considering heterogeneity among characters (sites). Two types of heterogeneity are often considered: partitions and mixtures [123, 158]. With partitions, the different characters are assigned in different categories, whereas with mixtures we assume that there are different categories, but we do not know which sites belong to which categories. Denote as f_k the frequency of the kth category in the mixture (with $\sum_k f_k = 1$), then (5.7) would become:

$$L_{aiz} = \sum_k f_k \left(\sum_{b=1}^{M} p_{ab}^k(t_{xz}) L_{bix} \right) \left(\sum_{b=1}^{M} p_{ab}^k(t_{yz}) L_{biy} \right) \ . \tag{5.11}$$

The exponent k of p indicates that these probabilities depend on the categories of the mixture.

The presence of partitions is ignored in this formulation, but they can be taken into account easily because the log-likelihood is summed over all sites: the full log-likelihood would become a sum of individual log-likelihoods similar to (5.10) for each partition.

The partitions can have different models of evolution and different mixtures as well. On the other hand, the models of evolution and / or the mixtures can be constrained to be the same across partitions (possibly with different parameter values). One can also imagine nested partitions with different shared model components, for instance, four partitions each with different mixtures, and a model of substitutions common to two partitions.

This general framework is implemented in `ape`. This covers many models of molecular evolution currently used in phylogenetics. Among those not included in this framework are the nonequilibrium models where some parameters are assumed to change over time (typically the nucleotide frequencies). Not included as well are the models with a nonfinite number of states, such as the number of repeats in microsatellites, and the models of insertions–deletions (indels).

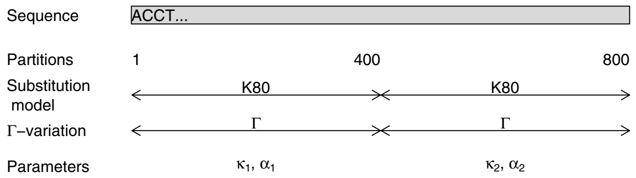
The user interface for defining a model of evolution is one of the functions `DNAmodel`, `AAmodel`, or `CODONmodel`, depending on the kind of data analyzed. These functions create an object whose class has the same name. Let us focus on DNA sequence data: the two other functions work sensibly in the same way.

`DNAmodel` has six arguments that define three aspects of a model of DNA evolution: the substitution model, the Γ-variation among sites, and the proportion of invariant sites. A partition can be defined for each of these aspects. The option `part.model` defines the partitions used for the substitution models: it needs a single vector of integers that specifies the partition each site belongs to; this vector is recycled if necessary. For instance, `part.model = c(1, 1, 2)` is used for a coding sequence in which the third codon position will be in a different partition. Another choice for two concatened sequences of, say 800 and 900 nucleotides, `part.model = c(rep(1, 800), rep(2, 900))` will specify a different partition for each sequence. If more than one partition is specified, it is possible to use different substitution models by giving a vector of models to `model`; for instance, `model = c("K80", "JC69")` means using Kimura's 1980 model for the first partition and Jukes–Cantor's one for the second (in other words, the transition / transversion ratio will be allowed to vary only in the first partition).

The intersite variation is specified in a way similar to the substitution models. Two arguments can be used: `part.gamma` which is used in the same way as `part.model`, and `ncat` which specifies the number of categories of the discretized Γ distribution [157] (1 by default meaning that there is no intersite variation).

The specification of invariant sites follows the same logic with two arguments: `part.invar` and `invar`. The latter is a logical vector giving whether there are invariant sites for each partition.

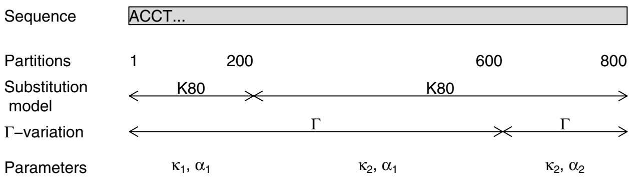
We have just seen that partitions are specified separately for the three components of the model. The partitions that are actually used (i.e., the sets of nucleotides with the same parameters of evolution) when fitting the model specified by `DNAmodel` result from crossing over all three components. This allows us to formulate a large number of models. To see how this works we consider a simple example with two partitions for the substitution model and two partitions for the intersite variation. If both partitions coincide the resulting model obviously has two partitions:



where κ_1 and κ_2 are the transition/transversion ratios, and α_1 and α_2 are the shape parameters for the Γ distribution of intersite variation. The code to specify this model is:

```
DNAmodel(part.model = c(rep(1, 400), rep(2, 400)),
         model = "K80",
         part.gamma = c(rep(1, 400), rep(2, 400)))
```

On the other hand, if they do not coincide the model has three partitions resulting from crossing over the two specified partitions. This allows us to specify parameters that are shared across several partitions.



The code is now:

```
DNAmodel(part.model = c(rep(1, 200), rep(2, 600)),
         model = "K80",
         part.gamma = c(rep(1, 600), rep(2, 200)))
```

Of course, the interest of `DNAmodel` is to let the user formulate some models that make sense biologically for the particular data at hand. A model of interest for a sequence could be:

```
DNAmodel(part.model = c(1, 1, 2), model = c("K80", "JC69"),
         part.gamma = c(1, 1, 2), ncat = c(4, 1))
```

This defines two partitions with respect to the codon positions: in the first one, Kimura's two-parameter model is assumed with an intersite variation following a Γ-distribution with four categories, and in the second one Jukes–Cantor's model is assumed with no intersite variation (because one category has been assumed for the second partition of `part.gamma`). This model seems biologically reasonable because mutations on the third codon position are likely to be less constrained than on the first and second ones, and thus transitions and transversions may occur at equal rates. Because mutations on the first and second codon positions have greater structural impact on the protein, it is likely that they vary along the sequence.

The above model assumes that the base frequencies are balanced: to relax this assumption, the model can be modified with:

```
DNAmodel(part.model = c(1, 1, 2), model = c("F84", "F81"),
         part.gamma = c(1, 1, 2), ncat = c(4, 1))
```

All the options of `DNAmodel` have default values which are:

```
DNAmodel(part.model = 1, model = "K80",
         part.gamma = 1, ncat = 1,
         part.invar = 1, invar = FALSE)
```

This implies that calling `DNAmodel()` generates a model with Kimura's two-parameter model for all sites, with no intersite variation, and no invariants.

When flexibility in model-building is possible, it is critical to assess the relevance of the models with empirical data [13, 14]. This is possible in the maximum likelihood framework, and this has been discussed repeatedly in the phylogenetic literature [71, 121]. This is dealt with in the next two sections.

5.2.3 Finding the Maximum Likelihood Tree

Once a model of sequence evolution has been chosen, its parameters must be estimated. In the maximum likelihood framework, this involves finding the values of the parameters that maximize (5.10) for a given data set. A difficulty comes from the fact that there are two kinds of parameters that need to be estimated: purely numeric parameters (branch lengths, substitution parameters, shape parameter of the Γ-distribution of intersite variation, etc.) and the topology of the tree. Maximum likelihood methods for tree estimation use numerical methods to estimate the first kind of parameter [40, 58, 159]. This is relatively straightforward because computer scientists have devoted a lot of effort to creating numerical methods that maximize complex functions with possibly many variables [e.g., 6, 139].

On the other hand, finding the topology that maximizes the likelihood is a much more difficult task. Several algorithms (sometimes called *heuristics*[1]) have been proposed for exploring the tree space.

`ape` has the function `mlphylo` that performs maximum likelihood estimation of phylogeny using molecular sequences. Its interface is:

```
mlphylo(model = DNAmodel(), x, phy, search.tree = FALSE)
```

where `x` is a DNA sequence data set, `phy` is a phylogenetic tree (as an object of class `"phylo"`), and `search.tree` specifies whether to search the tree space for the best topology (the default is only to estimate the branch lengths and other parameters). If the option `model` is omitted, Kimura's [81] model is used.

This function can be used to estimate the parameters of a relatively complex model of DNA evolution for a given phylogeny (leaving the default for `search.tree`).

If the tree space is searched (i.e., `search.tree = TRUE`), a method close to that of Guindon and Gascuel [58] is used. This involves starting from an initial tree (e.g., using `nj`), and then rearranging its topology with nearest-neighbor

[1] This redefinition is unfortunate because "heuristics" has a more useful meaning in epistemology.

interchanges (NNI). In Guindon and Gascuel's algorithm, NNIs are selectively done under some optimization criteria, leading to a very fast method of tree space search.

`mlphylo` returns an object of class `"phylo"` which is the estimated tree, with additional attributes. There are several method functions to extract this information: `logLik` returns the log-likelihood, `AIC` the Akaike information criterion, and `summary` prints details on the estimated tree and parameters (they are all generic).

5.2.4 DNA Mining with PHYML

The previous section explains how to define and fit a variety of molecular evolution models. How to select the appropriate model(s) for parameter estimation is an issue that has attracted a lot of attention and debate among statisticians [13, 15, 21, 99]. The importance of model selection in a likelihood framework has been made repeatedly in the phylogenetic literature [101, 122, 120]. Posada and Crandall [121] developed a computer program, to be used with the program PAUP*, that fits a series of DNA evolution models to a given data set. This program is supposed to help in selecting a substitution model for further analyses.[2]

In order to provide a similar functionality, but with a free phylogeny estimation program, `ape` has the function `phymltest` which, instead of PAUP*, uses PHYML developed by Guindon and Gascuel [58]. Another difference is that `phymltest` lets PHYML search for the best tree for all fitted models. All substitution models available in PHYML are used; these are: JC69, K80, F81, F84, HKY85, TN93, and GTR. Additionally, models with(out) invariant sites and / or intersite variation (with the usual Γ distribution) are used. This results in 28 fitted models. The interface is:

```
phymltest(seqfile, format = "interleaved", itree = NULL,
          exclude = NULL, execname, path2exec = NULL)
```

where `seqfile` is the name of the file with the sequences (given as a character). The other arguments have default values, except `execname`, the name of the PHYML executable, which must be specified as a character string. Under Windows, `execname` may be left missing if the PHYML executable file is named 'phyml_win32.exe' (its original name in PHYML's distribution).

Some care must be taken to set correctly the three diffferent paths involved here: the path to PHYML's executable, the path to the sequence file, and the path to R's working directory. Here are two possible uses under Linux and Windows, respectively:

```
phymltest("/home/paradis/data/seq.txt",
```

[2] MODELTEST has had remarkable success: the paper published in *Bioinformatics* was cited 3068 times (source: Web of Science, January 23, 2006).

```
           execname = "phyml_linux",
           path2exec = "/usr/local/bin")
phymltest("D:/data/seq.txt", path2exec = "D:/phyml")
```

If R returns an error message because of a problem in finding one of the files, it might be better to move all files in the same directory, say '/home/paradis/phyml' or 'D:/phyml', and set the latter as R's working directory:

```
# Linux:
setwd("/home/paradis/phyml")
phymltest("seq.txt", execname = "phyml_linux")
# Windows:
setwd("D:/phyml")
phymltest("seq.txt")
```

`phymltest` returns an object of class `"phymltest"` that has three methods: the `print` method prints a table of all fitted models with the number of free parameters, the values of the log-likelihood, and the Akaike Information Criterion (AIC); the `summary` method computes and prints all possible likelihood ratio tests (LRTs) between pairs of nested models; and the `plot` method plots, on a vertical axis, all AIC values with an indication of the corresponding model (see Section 5.5 for an example).

5.3 Bootstrap Methods and Distances Between Trees

The use of the bootstrap has enjoyed great success in phylogenetic analyses [35]. The idea of the bootstrap can be sketched as follows: suppose we are interested in quantifying the confidence level in a parameter estimate given some data, but we cannot apply the methods based on distributional theory of this parameter. Then we could resample the sample at hand many times, mimicking the process of sampling the real population several times. The variation in the estimated parameter from the "bootstrap" samples is a measure of the confidence level in this estimate [29].

The idea is simple, intuitive, and elegant, but, in some situations, requires intensive computations [32]. The application of the bootstrap in phylogeny estimation is almost as simple: estimate a tree with a given method, resample the original data (the matrix taxa $\times$ characters) a large number of times, and analyze these "bootstrap" samples with the same method, and calculate the number of times the clades observed in the estimated tree appear in the "bootstrap" ones.

The application of the bootstrap to assess confidence levels in phylogenetic estimation has been criticized, but Efron, Halloran, and Holmes [31] showed that this was due to confusion in the interpretation of the original bootstrap method by Felsenstein [35]. Efron et al. also proposed another way to compute the bootstrap values for hypothesis testing rather than assessing confidence levels [31].

In this section, we examine the different ways of resampling phylogenetic data, comparing (possibly a large number of) phylogenetic trees, and computing bootstrap values.

5.3.1 Resampling Phylogenetic Data

R has a powerful function, `sample`, that can be used to create a bootstrap sample from a data set: this function returns a sample, by default without replacement, of the vector given as argument. If the option `replace = TRUE` is used, then sampling is done with replacement which is clearly what is needed for a bootstrap sample. Below is a simple example with a vector `x` containing 10 values 1, 2, . . . , 10:

```
> x <- 1:10
> sample(x)
 [1]  9  8  6  1 10  7  5  4  3  2
> sample(x, replace = TRUE)
 [1]  7  5  2  4 10  6  2  1  2  2
```

Note that `sample(x)` returns a (random) permutation of the data. We can also give a single integer value to `sample`, say 10, which will then return a sample of integers from 1 to 10.

With phylogenetic data we are mostly interested in resampling the columns of the matrix taxa $\times$ characters (where taxa are the rows, and characters the columns). If this matrix is called `X`, then one can simply do:

```
X[, sample(ncol(X), replace = TRUE)]
```

Note the presence of the comma just after the left bracket which means that all rows of `X` will be selected (see p. 15). Here is an example of how this could be used:

```
> x <- scan(what = "")
1: a a c t t a a c t t c a c c t
16:
Read 15 items
> X <- matrix(x, 3, 5, byrow = TRUE)
> X
     [,1] [,2] [,3] [,4] [,5]
[1,] "a"  "a"  "c"  "t"  "t"
[2,] "a"  "a"  "c"  "t"  "t"
[3,] "c"  "a"  "c"  "c"  "t"
> X[, sample(ncol(X), replace = TRUE)]
     [,1] [,2] [,3] [,4] [,5]
[1,] "a"  "c"  "c"  "a"  "a"
```

```
[2,] "a"  "c"  "c"  "a"  "a"
[3,] "a"  "c"  "c"  "a"  "c"
```

It happens sometimes that the columns of a matrix are affected with weights, for instance, because the same values have been observed several times for all taxa [31, 84]. This may be a useful way to reduce the size of the data matrix, particularly if few sites are polymorphic. In these cases, resampling must take these weights into account. Suppose each column of `X` is associated with a weight stored in a vector `w` (`length(w)` is equal to `ncol(X)`), then a bootstrap sample is obtained using the option `prob` of `sample`:

```
X[, sample(ncol(X), replace = TRUE, prob = w)]
```

The values passed to `prob` need not sum to 1 because they are used as relative probability weights. If the values in `w` are integer weights, one may need to use the option `size` to produce a sample of the appropriate size:

```
X[, sample(ncol(X), replace = TRUE, prob = w, size = sum(w))]
```

An issue in resampling phylogenetic data is that the columns may not be independent, particularly in the case of molecular sequences. A solution is to sample the sites by groups (or blocks) rather than individually. There are several ways to do this in R. One is to build blocks of sites using the function `splitseq` in seqinr, sample among these blocks, and reconstitute the sequence:

```
> library(seqinr)
> x <- scan(what = "")
1: a a a c c c g g g t t t
13:
Read 12 items
> x
 [1] "a" "a" "a" "c" "c" "c" "g" "g" "g" "t" "t" "t"
> x.codon <- splitseq(x)
> x.codon
[1] "aaa" "ccc" "ggg" "ttt"
> x.boot <- sample(x.codon, replace = TRUE)
> x.boot
[1] "ccc" "aaa" "ttt" "ggg"
> s2c(c2s(x.boot))
 [1] "c" "c" "c" "a" "a" "a" "t" "t" "t" "g" "g" "g"
```

The length of the blocks sampled may be altered with the option `word` of `splitseq` (which is 3 by default):

```
> s2c(c2s(sample(splitseq(x, word = 2), replace = TRUE)))
 [1] "t" "t" "a" "a" "g" "t" "g" "g" "g" "g" "a" "c"
```

A more general solution to this problem is to sample the indices of the vector instead of the vector itself. Let us consider the same case of sampling blocks of three nucleotides in the vector x. First, build a vector with the indices 3, 6, . . . :

```
> block <- 3
> i <- seq(block, length(x), block)
> i
[1]  3  6  9 12
```

Then, sample this vector i as before:

```
> i.boot <- sample(i, replace = TRUE)
> i.boot
[1] 12  6 12  9
```

What we want in fact is a vector with the values 10, 11, 12, 4, 5, 6, 10, 11, 12, 7, 8, and 9. The pattern is clear: the 3rd, 6th, 9th, and 12th values are those in i.boot, the 2nd, 5th, 8th, and 11th ones can be obtained with i.boot - 1, and the 1st, 4th, 7th, and 10th ones can be obtained with i.boot - 2. We first create a vector of the appropriate length, and then feed in the values with a loop:

```
> boot.ind <- numeric(length(x))
> boot.ind[i] <- i.boot
> for (j in 1:(block - 1)) boot.ind[i - j] <- i.boot - j
> boot.ind
 [1] 10 11 12  4  5  6 10 11 12  7  8  9
```

The bootstrap sample is finally obtained with:

```
> x[boot.ind]
 [1] "t" "t" "t" "c" "c" "c" "t" "t" "t" "g" "g" "g"
```

Note that we did not use the value of block (3) or length(x) (12) in the above commands, so they can be used in different situations. They also can be used to resample blocks of columns of a data matrix: in this case it is necessary to replace length(x) by ncol(x), and the final command by x[, boot.ind].

Because in most cases, a large number of bootstrap samples will be needed, it is useful to include the appropriate sampling commands in a loop and / or a function. This is what is done by the function boot.phylo described below.

5.3.2 Bipartitions and Computing Bootstrap Values

Once bootstrap samples and trees have been obtained, it is necessary to summarize the information from them. ape provides several functions for this task depending on the approach taken.

A bipartition is made with two subsets of the tips of a tree as defined by an internal branch. `prop.part` takes as its argument a list of trees and returns an object of class `"prop.part"` which is a list of all observed bipartitions together with their frequencies. There are `print` and `summary` methods for this class; the latter prints only the frequencies. Here is the result with a four-taxa tree:

```
> tr <- read.tree(text = "((a,(b,c)),d);")
> prop.part(tr)
==> 1 time(s):[1] a b c d
==> 1 time(s):[1] a b c
==> 1 time(s):[1] b c
```

Instead of a list of bipartitions indexed to the internal branches, `prop.part` returns a list indexed to the numbers of the nodes, and gives the tips that are descendants of the corresponding node: thus the first vector in the list includes all tips because the first node is the root. It is then straightforward to get the bipartitions. The following code prints them for an object named `Y`:

```
for (i in 2:length(Y)) {
    cat("Internal branch", i - 1, "\n")
    print(Y[[i]], quote = FALSE)
    cat("vs.\n")
    print(Y[[1]][!(Y[[1]] %in% Y[[i]])], quote = FALSE)
    cat("\n")
}
```

`prop.clades` takes two arguments: a tree (as a `"phylo"` object), and either a list of trees, or a list of bipartitions as returned by `prop.part`. In the latter case, the list of bipartitions must be named explicitly (e.g., `prop.clades(tr, part = list.part)`). This function returns a numeric vector with, for each clade in the tree given as first argument, the number of times it was observed in the other trees or bipartitions. For instance, we have the obvious following result:

```
> prop.clades(tr, tr)
[1] 1 1 1
```

Like the previous functions, the results are indexed according to the node numbers.

Note that both `prop.part` and `prop.clades` do not require that all trees analyzed have the same tips (as identified by the labels). This may give undesirable results with `prop.part`, but this may be useful in some situations, particularly with `prop.clades`, because the “support” values may come from a sample of trees with a larger number of tips.

Using the two functions just described, bootstrap samples obtained as described in the previous section, and the appropriate function(s) for phylogeny estimation, one can perform the bootstrap for the estimated phylogeny in a straightforward way using basic programming techniques. However, to do such an analysis directly, the function `boot.phylo` can be used instead. Its interface is:

```
boot.phylo(phy, x, FUN, B = 100, block = 1)
```

with the following arguments:

`phy` an object of class `"phylo"` which is the estimated tree;
`x` the original data matrix (taxa as rows and characters as columns);
`FUN` the function used to estimate `phy` from `x`. Note that if the tree was estimated with a distance method, this must be specified as something such as:
```
FUN = function(xx) nj(dist.dna(xx))
```
or:
```
FUN = function(xx) nj(dist.dna(xx, "TN93"))
```
`B` the number of bootstrap replicates;
`block` the size of the "block" of columns, that is, the number of columns that are sampled together during the bootstrap sampling process (e.g., if `block = 2`, columns 1 and 2 are sampled together, the same for columns 3 and 4, 5 and 6, and so on; see above).

`boot.phylo` returns exactly the same vector as `prop.clades`. The bootstrap trees generated by this function are not saved, and so cannot be examined or further analyzed, for instance, to perform the two-level bootstrap procedure developed by Efron et al. [31]. This can be circumvented by doing the bootstrap samples beforehand. A typical program to get a list of bootstrap trees could be:

```
B <- 100
btr <- list()
length(btr) <- B
for (i in 1:B)
  btr[[i]] <- nj(dist.dna(x[, sample(ncol(x), replace = TRUE)]))
```

Then, if `tr.est` is the estimated tree, doing:

```
prop.clades(tr.est, brt)
```

or:

```
pp <- prop.part(brt)
prop.clades(tr.est, part = pp)
```

will give the same results as using `boot.phylo` (except for the differences due to random sampling!)

5.3.3 Distances Between Trees

The idea of distances between trees is somehow related to the bootstrap because this requires summarizing and quantifying the variation in topology from different trees. Several ways to compute these distances have been proposed in the literature [39, Chap. 30]. Two of them are available in the function `dist.topo`.

Penny and Hendy [118] proposed measuring the distance between two trees as twice the number of internal branches that differ in their bipartitions. Rzhestky and Nei [133] proposed a modification of this distance to take multichotomies into account: this is the default method in `dist.topo`. For a trivial example:

```
> tr <- read.tree(text = "((a,b),(c,d));")
> tb <- read.tree(text = "((a,d),(c,b));")
> dist.topo(tr, tb)
[1] 2
> dist.topo(tr, tr)
[1] 0
```

Billera, Holmes, and Vogtmann [9] developed a more elaborate distance based on the concept of tree space. This space is actually a cube complex because it is made up of cubes that share certain faces. Two trees with the same topology lie in the same cube of dimension $n - 2$ (n being the number of tips). If they do not have the same topology they will be in two distinct cubes. However, these cubes meet at the origin where the internal branches that are different between the trees are equal to zero. Thus it is possible to define a geometric distance for different topologies. This is computed with the option `method` of `dist.topo`:

```
> tr <- rtree(10)
> trb <- rtree(10)
> dist.topo(tr, trb)
[1] 12
> dist.topo(tr, trb, method = "BHV01")
[1] 3.455182
> dist.topo(tr, tr, method = "BHV01")
[1] 0
```

5.3.4 Consensus Trees

Consensus trees are an interesting way to summarize a set of trees: if they are dichotomous, the clades not observed in all (strict consensus) of the majority (majority-rule consensus) will be collapsed as multichotomies.

The function `consensus` returns the consensus from a list of trees given in the same way as for `prop.part` or `prop.clades`. There is one option, `p`, which specifies the threshold, as a real between 0.5 and 1, of the proportion of the bipartitions for their inclusion in the consensus tree. If `p = 1` (the default), then the strict consensus tree is returned, whereas `p = 0.5` returns the majority-rule consensus tree. This corresponds to the parameter l of the M_l consensus methods in [39].

5.4 Molecular Dating

Some parameters are confounded in phylogenetic models (branch lengths and substitution rates), therefore it is not possible to estimate branch lengths in units that are proportional to time. This must be done using additional assumptions on rate variations. Sanderson [135, 136] proposed two approaches for the estimation of dates using molecular phylogenies. These are implemented in `ape` with a set of functions that are explained below.

Nonparametric rate smoothing (NPRS) assumes that each branch of the tree has its own rate, but these rates change smoothly between connected branches [135]. Given a tree with estimated branch lengths in terms of number of substitutions, it is possible to estimate the dates of the nodes by minimizing the changes in rates from one branch to another. Practically this is done by minimizing the function:

$$\sum |\hat{r}_k - \hat{r}_j|^p \ , \tag{5.12}$$

where $\hat{r}$ is the estimated absolute rate, k and j are two nodes of the same branch, and p is an exponent (usually 2). The function `ratogram` computes the absolute rates for a tree with branch lengths using the NPRS method. By default the age of the root is one, but this can be changed with the option `scale`. The function `chronogram` computes the ages of the nodes with the same method. Its options are the same as for `ratogram`.

In order to make a trade-off between nonparametric and parametric methods, Sanderson [136] proposed to modify his method by using a semiparametric approach based on a penalized likelihood. The latter (denoted Ψ) is made of the likelihood of the "saturated" model (the one that assumes one rate for each branch of the tree) minus a roughness penalty (denoted Φ) which is similar to (5.12) multiplied by a smoothing parameter λ:

$$\Psi = \ln L - \lambda \Phi \ , \tag{5.13}$$

$$L = \prod r_k^x \frac{\exp(-r_k)}{x!} \ . \tag{5.14}$$

with x being the number of substitutions observed on a branch. The product of the likelihood function L is made over all branches of the tree. If $\lambda = 0$ then the above model is the (saturated) model with one distinct rate for each branch. If $\lambda = +\infty$, then the model converges to a clocklike model with the same rate for all branches. In order to choose an optimal value for λ, Sanderson [136] suggested a cross-validation technique where each terminal branch is removed from the data and then its length is predicted from the remaining data. A different criterion is used here:

$$D_i^2 = \sum_{j=1}^{n-2} \frac{(\hat{t}_j - \hat{t}_j^{-i})^2}{\hat{t}_j} , \tag{5.15}$$

where $\hat{t}_j$ is the estimated date for node j with the full data, and $\hat{t}_j^{-i}$ is the one estimated after removing tip i. This criterion is easier to calculate than Sanderson's [136].

The penalized likelihood method is implemented in the function `chronopl`; its interface is:

```
chronopl(phy, lambda, node.age = NULL, nodes = NULL,
         CV = FALSE)
```

where `phy` is an object of class `"phylo"` with branch lengths giving the number of substitutions (or its expectation), `lambda` is the smoothing parameter λ, `node.age` is a numeric vector giving the dates that are known, `nodes` is the number of the nodes dates of which are known, and `CV` is a logical specifying whether to do the cross-validation. This function returns a tree with branch lengths proportional to time (i.e., a chronogram) with attributes `rates` (the estimated absolute rates, $\hat{r}$), and `ploglik` (the penalized likelihood). If `CV = TRUE`, an additional attribute `D2` is returned with the value calculated with (5.15) for each tip.

The cross-validation may be done for different values of λ in a straightforward way, for instance, for $\lambda = 0.1, 1, 10, \ldots, 10^6$:

```
l <- 10^(-1:6)
cv <- numeric(length(l))
for (i in 1:length(l))
  cv[i] <- sum(attr(chronopl(phy, lambda = l[i]), "D2"))
plot(l, cv)
```

Sanderson suggested selecting the value of λ that minimizes the cross-validation criterion. If `CV = TRUE`, `chronopl` returns a value D_i^2 for each tip, so it is possible to examine which observations are particularly influential, for instance with:

```
chr <- chronopl(phy = phy.est, lambda = 1)
plot(attr(chr, "D2"), type = "l")
```

5.5 Case Studies

In this section, we come back to some of the data prepared in Chapter 3. We see how we can estimate phylogenies, eventually repeat some analyses done in the original publications, and possibly see how we could go further with R.

5.5.1 *Sylvia* Warblers

To continue with the *Sylvia* data, it may be necessary to reload the data prepared and saved previously:

```
load("sylvia.RData")
```

A distance matrix can be estimated from these aligned sequences using `dist.dna`; because 2 of the 25 sequences are substantially incomplete, we use the option `pairwise.deletion = TRUE`:

```
syl.K80 <- dist.dna(sylvia.seq.ali, pairwise.deletion = TRUE)
```

We recall that the default model for this function is Kimura's two-parameter one. We use the option `model` to try different models:

```
syl.F84 <- dist.dna(sylvia.seq.ali, model = "F84",
                    pairwise.deletion = TRUE)
syl.TN93 <- dist.dna(sylvia.seq.ali, model = "TN93",
                    pairwise.deletion = TRUE)
syl.GG95 <- dist.dna(sylvia.seq.ali, model = "GG95",
                    pairwise.deletion = TRUE)
```

A way to compare these distance matrices is simply to look at their correlations. We do this by binding all distances in a single matrix, and compute the correlations among its columns (the results are rounded to three digits):

```
> round(cor(cbind(syl.K80, syl.F84, syl.TN93, syl.GG95)), 3)
         syl.K80 syl.F84 syl.TN93 syl.GG95
syl.K80    1.000   0.908    1.000    0.927
syl.F84    0.908   1.000    0.911    0.686
syl.TN93   1.000   0.911    1.000    0.925
syl.GG95   0.927   0.686    0.925    1.000
```

This shows some substantial differences in the estimated distances. Note that a perfect correlation does not guarantee that the distances are the same: some graphical analyses are needed to check this. We do this to examine the saturation of substitutions in the sequences. We first compute the distances using the Jukes–Cantor model and the raw distance (i.e., proportion of different sites):

```
syl.JC69 <- dist.dna(sylvia.seq.ali, model = "JC69",
                     pairwise.deletion = TRUE)
syl.raw <- dist.dna(sylvia.seq.ali, model = "raw",
                    pairwise.deletion = TRUE)
```

We then plot these two distances in a simple plot expecting the raw distances to be smaller because they do not consider multiple substitutions on a single site; we also plot the Jukes–Cantor distance *versus* the Kimura one to show the potential influence of the transition/transversion ratio (Fig. 5.1):

```
layout(matrix(1:2, 1))
plot(syl.JC69, syl.raw)
abline(b = 1, a = 0) # draw x = y line
plot(syl.K80, syl.JC69)
abline(b = 1, a = 0)
```

These plots show, as expected, that the most divergent sequences are slightly saturated, whereas the transition/transversion ratio does not seem to affect the estimated distances greatly.

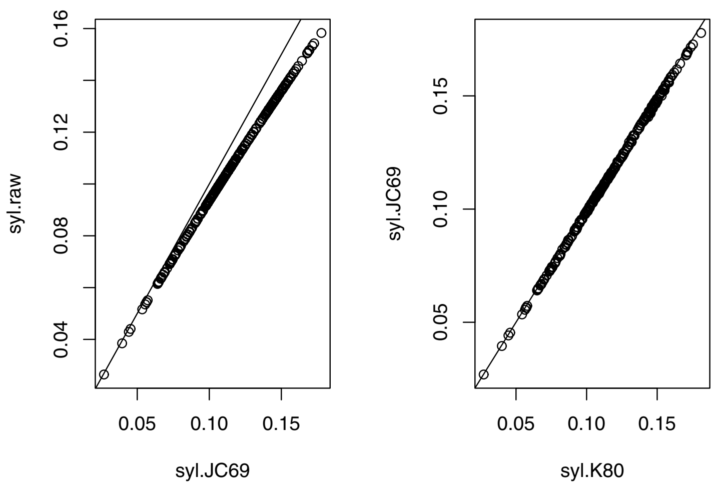


Fig. 5.1. Saturation plots for the cytochrome *b* sequences of 25 species of *Sylvia* showing the effects of multiple substitutions (left) and of the transition/transversion ratio (right)

A point we explore briefly is the impact of the choice of the substitution model on the phylogeny estimation with the NJ method. We estimate a tree with the function `nj` for each distance matrix:

```
nj.sylvia.K80 <- nj(syl.K80)
nj.sylvia.F84 <- nj(syl.F84)
nj.sylvia.TN93 <- nj(syl.TN93)
nj.sylvia.GG95 <- nj(syl.GG95)
```

To see if the estimated topology is the same, we compute the topological distance among them:

```
> dist.topo(nj.sylvia.K80, nj.sylvia.F84)
[1] 20
> dist.topo(nj.sylvia.K80, nj.sylvia.TN93)
[1] 0
> dist.topo(nj.sylvia.K80, nj.sylvia.GG95)
[1] 16
> dist.topo(nj.sylvia.F84, nj.sylvia.TN93)
[1] 20
> dist.topo(nj.sylvia.F84, nj.sylvia.GG95)
[1] 26
> dist.topo(nj.sylvia.TN93, nj.sylvia.GG95)
[1] 16
```

The same topologies were obtained with Kimura's and Tamura and Nei's models. We visualize the clades that are consistently observed with the different substitution models by computing the consensus tree with the function `consensus` and plot it after changing its tip labels with the species names in place of the GenBank numbers (Fig. 5.2):

```
sylvia.cons <- consensus(nj.sylvia.K80, nj.sylvia.F84,
                         nj.sylvia.GG95, nj.sylvia.TN93)
sylvia.cons$tip.label <- taxa.sylvia[sylvia.cons$tip.label]
plot(sylvia.cons, no.margin = TRUE)
```

We now do a bootstrap analysis like the one reported by Böhning-Gaese et al. [10] using `boot.phylo` directly:

```
> nj.boot.sylvia <- boot.phylo(phy = nj.sylvia.K80,
                         x = sylvia.seq.ali,
                         FUN = function(xx) nj(dist.dna(xx,
                                pairwise.deletion = TRUE)),
                         B = 200)
> nj.boot.sylvia
 [1] 200   9 181 192  96  40  74  78  75 194 196  91 193 185
[15]  99 147  74 169 194 136  85 199  76
```

Note how the `FUN` argument is used here: because we resample the original aligned sequences, the tree is estimated by first computing the distances, then performing the neighbor-joining. We use 200 bootstrap replicates as in [10].

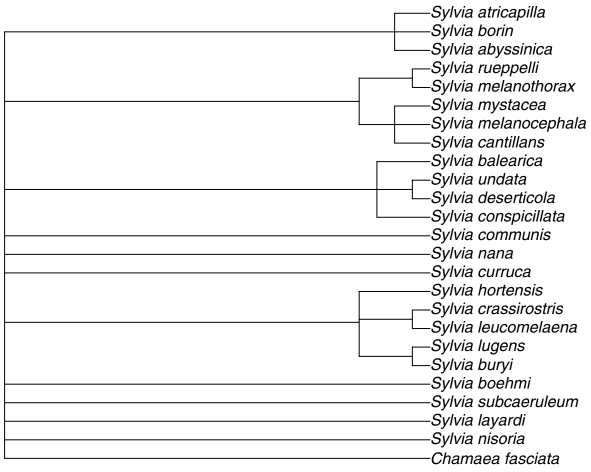


Fig. 5.2. Consensus tree for *Sylvia* based on four neighbor-joining trees estimated with different substitution models

How could these bootstrap values have been influenced by the fact that we deal with coding sequences? We can assess this by using the option `block` of `boot.phylo`; this will result in resampling at the codon level instead of at the site level:

```
> nj.boot.sylvia.codon <- boot.phylo(nj.sylvia.K80,
                                     sylvia.seq.ali,
                                     function(xx) nj(dist.dna(xx,
                                      pairwise.deletion = TRUE)),
                                     200, 3)
> nj.boot.sylvia.codon
 [1] 200  13 179 199  83  37  74  92  84 192 196  92 187 179
[15]  99 135  71 167 197 134  86 199  91
```

The results are very close to the site-level resampling analysis; we thus consider the latter in the following.

We now plot the estimated tree by NJ with the bootstrap values on the nodes. We first copy the estimated tree, substitute the accession numbers (which were used as tip labels) with the species names, add to this tree the bootstrap values (as percents), and finally root this unrooted tree using *Chamaea fasciata* as outgroup:

```
nj.est <- nj.sylvia.K80
nj.est$tip.label <- taxa.sylvia[nj.est$tip.label]
nj.est$node.label <- nj.boot.sylvia / 2
nj.est <- root(nj.est, "Chamaea_fasciata")
```

The tree is then plotted with `plot`, the bootstrap values are added with `nodelabels`, and we draw a scale bar (Fig. 5.3):

```
plot(nj.est, no.margin = TRUE)
nodelabels(nj.est$node.label, bg = "white")
add.scale.bar(y = 0.5, length = 0.01)
```

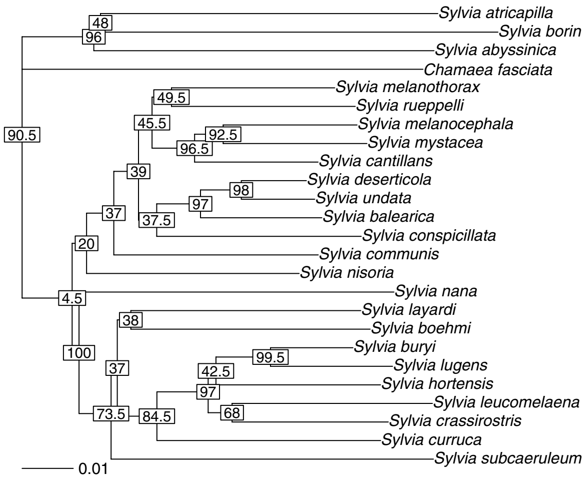


Fig. 5.3. Phylogenetic relationships among 25 species of the genus *Sylvia* based cytochrome *b* sequences analyzed with neighbor-joining and Kimura's two-parameter distance

The bootstrap values shown in Fig. 5.3 are very close to those obtained by Böhning-Gaese et al. [10]. It is interesting to note that the clades well supported by the bootstrap analysis were also those that were consistently found in the four trees estimated by NJ with the different substitution models.

We finish by saving the final tree in a file using the Newick format:

```
write.tree(nj.est, "sylvia_nj_k80.tre")
```

5.5.2 Phylogeny of the Felidae

To continue the analyses with the Felidae data [75], we first load the data previously prepared and saved:

```
load("felid.RData")
```

We focus here on an analysis with `phymltest`. PHYML has been installed (this is a single executable file) in the same directory where the sequence file has been saved (which is also set as R's working directory). The command for the present analysis is thus simply:

```
phymltest.felid <- phymltest("felidseq16S.phy",
                             execname = "phyml_linux")
```

This takes a few minutes to run on a PC with a processor at 3 GHz and 521 Mb of cache memory, and 2 Gb of RAM memory. Displaying the results shows the log-likelihood and AIC values for each model:

```
> phymltest.felid
          nb.free.para    loglik      AIC
JC69                 1 -2301.182 4604.364
JC69+I               2 -2162.121 4328.243
JC69+G               2 -2151.150 4306.300
JC69+I+G             3 -2144.011 4294.023
K80                  2 -2174.653 4353.306
K80+I                3 -2031.520 4069.040
K80+G                3 -2012.344 4030.688
K80+I+G              4 -2001.572 4011.144
F81                  4 -2301.058 4610.116
F81+I                5 -2161.592 4333.185
F81+G                5 -2143.511 4297.022
F81+I+G              6 -2139.011 4290.023
F84                  5 -2163.884 4337.768
F84+I                6 -2013.974 4039.949
F84+G                6 -1993.178 3998.356
F84+I+G              7 -1985.373 3984.745
HKY85                5 -2170.704 4351.408
KHY85+I              6 -2018.443 4048.886
HKY85+G              6 -1997.119 4006.238
HKY85+I+G            7 -1988.521 3991.041
TN93                 6 -2138.149 4288.298
TN93+I               7 -2002.168 4018.336
TN93+G               7 -1977.770 3969.539
TN93+I+G             8 -1972.596 3961.192
GTR                  9 -2132.276 4282.553
GTR+I               10 -1998.004 4016.009
GTR+G               10 -1973.184 3966.367
GTR+I+G             11 -1967.919 3957.838
```

The summary function computes all possible paired likelihood ratio tests (211 tests):

```
> summary(phymltest.felid)
       model1    model2       chi2 df  P.val
1        JC69    JC69+I 278.121860  1 0.0000
2        JC69    JC69+G 300.064594  1 0.0000
3        JC69  JC69+I+G 314.341858  2 0.0000
```

```
4        JC69       K80 253.058660  1 0.0000
5        JC69     K80+I 539.323882  2 0.0000
....
```

We can plot these results to have a more synthetic view (Fig. 5.4):

```
plot(phymltest.felid)
```

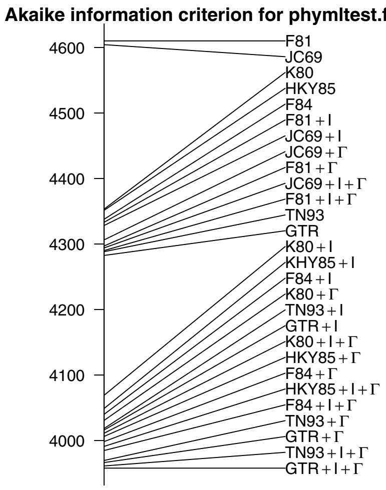


Fig. 5.4. Results of the analysis of 16S mitochondrial sequences from 35 species of Felidae and two other carnivores with `phymltest`

The most complex model GTR + I + Γ is the one that best explains the data in terms of AIC. An interesting pattern from Fig. 5.4 is that for a given substitution model, adding invariants (I) considerably improves the fit, whereas this improvement is even better by adding Γ, and again better with both; thus there is a hierarchy $X >>> X + I >> X + \Gamma > X + I + \Gamma$.

When comparing the substitution models, the key element seems to take the transition / transversion ratio into account. Once this has been included in the model (F80 being the simplest one), taking unequal base frequencies into account is also important although less than the previous parameter.

Once the analysis with `phymltest` has been done, it is possible to read the trees estimated by PHYML:

```
tr <- read.tree("felidseq16S.phy_phyml_tree.txt")
```

This file contains the 28 trees estimated by PHYML, the last one being the one estimated with the most complex model. We extract this tree, substitute its tip labels to get the species names in place of the accession numbers, root the tree with *Galidia elegans* as outgroup, remove the two non-felid species, and plot the final tree (Fig. 5.5):

```
mltree.felid <- tr[[28]]
mltree.felid$tip.label <- taxa.felid[mltree.felid$tip.label]
mltree.felid <- root(mltree.felid, "Galidia_elegans")
mltree.felid <- drop.tip(mltree.felid, c("Crocuta_crocuta",
                                          "Galidia_elegans"))
plot(mltree.felid)
add.scale.bar(length = 0.01)
```

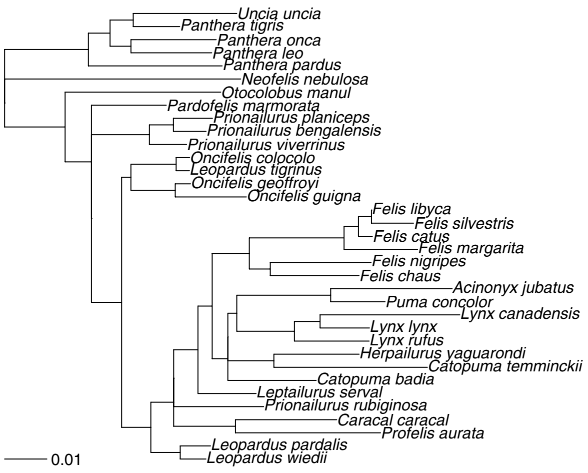


Fig. 5.5. Maximum likelihood estimate of the extant Felidae using 16S mitochondrial sequences with GTR + I + Γ

From this ML estimate of the phylogeny of the Felidae, we can now estimate a chronogram with the NPRS method [137]. Johnson and O'Brien [75] wrote that extant felids last shared a common ancestor 10–15 million years ago. We use the midpoint of this range as the age of the root. We plot the estimated chronogram and draw the time-axis with `axisPhylo` (Fig. 5.6):

```
felid.chrono <- chronogram(mltree.felid, scale = 12.5)
par(mar = c(2, 0, 0, 0))
plot(felid.chrono, cex = 0.8)
axisPhylo()
```

We save this chronogram for further analysis:

```
write.tree(felid.chrono, "felid.chrono.tre")
```

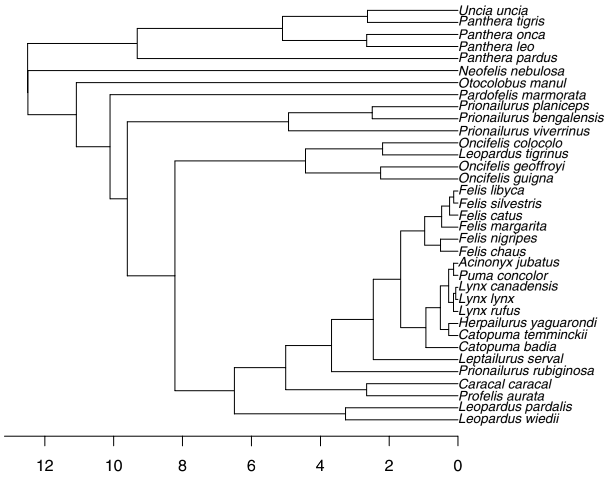


Fig. 5.6. Chronogram of the extant Felids estimated with the NPRS method

5.5.3 Butterfly DNA Barcodes

We have 466 aligned sequences of COI: we limit ourselves here to simple analyses. Hebert et al. [67] showed that there seem to be several (ten actually) species instead of one originally recognized. We compute the pairwise distances between all specimens with `dist.dna`. We take care to use the option `pairwise.deletion = TRUE` because many sequences do not have the same length:

```
M.astraptes.K80 <- dist.dna(astraptes.seq.ali,
                            pairwise.deletion = TRUE)
```

We look at the distribution of the distances using `summary`:

```
> summary(M.astraptes.K80)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.00000 0.01590 0.02107 0.02749 0.03887 0.08326
```

As a comparison, we can look at the summary of the distances without the option `pairwise.deletion = TRUE`:

```
> summary(dist.dna(astraptes.seq.ali))
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.00000 0.00000 0.00000 0.00122 0.00000 0.07155
```

This shows that most distances would be equal to zero because only a few sites remain after removing all those with at least one missing data (which is the default of `dist.dna`).

We may plot an histogram of the 108,345 distances (Fig. 5.7):

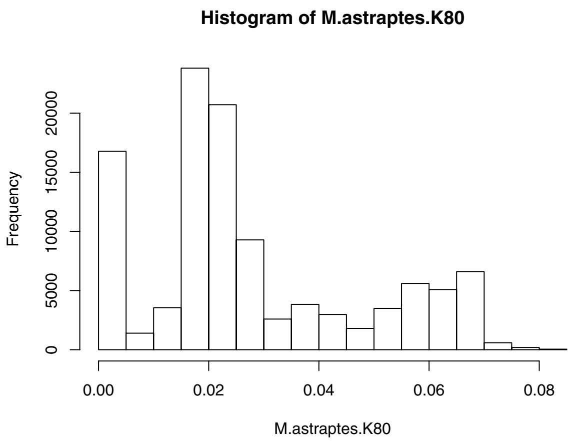


Fig. 5.7. Distribution of pairwise distances among 466 specimens *Astraptes fulgerator* based on cytochrome oxydase I sequences analyzed with Kimura's two-parameter distance

```
hist(M.astraptes.K80)
```

This clearly shows three peaks in the distribution: at 0, around 0.02, and around 0.07. This is in complete agreement with Hebert et al.'s results which showed that these peaks correspond to differentiation within populations, intraspecies, and interspecies, respectively.

It is possible to estimate an NJ tree with the distance matrix to assess how the different taxa are differentiated:

```
tr <- nj(M.astraptes.K80)
tr$tip.label <- taxa.astraptes[tr$tip.label]
```

The resulting tree is a bit too large to be displayed with `plot.phylo`, so we may use `zoom` instead. For this we have to find the indices of each taxon in the vector of tip labels. Here is a possible solution:

```
taxon <- unique(taxa.astraptes)
L <- list()
length(L) <- 10
for (i in 1:10)
  L[[i]] <- grep(taxon[i], tr$tip.label)
```

We can now use `L` as an argument to `zoom`. We may plot all the subtrees at once in a large PDF file with:

```
pdf("astraptes.pdf", width = 30, height = 30)
zoom(tr, L)
dev.off()
```

and then open it with an appropriate viewer. Each taxon can be visualized separately with, for instance, `zoom(tr, L[1])`.

5.6 Perspectives

The capabilities of R to estimate phylogenies are still limited compared to programs such as Phylip of PAUP*; however, there are good reasons to continue the current development of these methods.

- Some methods are easily implemented because the needed functions already exist in R. For instance:
  - The implementation of Bayesian methods should be eased by the functionalities already present in `ape` (computation of tree likelihood, generation of random trees) and other packages (random numbers, probability density functions);
  - The flexibility of R for reading and manipulating various kinds of data will ease the implementation of new methods of phylogeny estimation, such as those based on genomic rearrangements [88].
- R has many functionalities for efficient computation, particularly for large data sets, which are useful in the estimation of large phylogenies [58, 144, 152, 159].
- The integration of phylogeny estimation with other facets of phylogenetics, such as tree drawing (Chapter 4) or analysis of macroevolution (Chapter 6) is a very useful feature for users.
- The implementation of different methods in different programs makes their comparison difficult, because even the implementation of the same method in different programs could result in substantial differences among the results.

The last point has rarely been considered in the phylogenetic literature, although it has been demonstrated that even simple computational tasks (such as computing a sample variance) may give very different results depending on the statistical package [98, 97]. Kosiol and Goldman [85] showed that analyzing the same protein sequences with the same method but using different packages resulted in differences that would be considered statistically significant.

5.7 Exercises

1. Consider a DNA sequence that evolves according to the Jukes–Cantor (JC69) model.
   (a) Build the corresponding rate matrix using for the overall rate of change the value 3×10^{-4}.

   (b) Compute, using two different approaches, the probability matrix for $t = 1$, $t = 1000$, and $t = 1 \times 10^6$. What do you observe? Was that expected?
   (c) What could you conclude about phylogeny estimation from this exercise?

2. Consider a GTR model with the following parameters: $\alpha = 0.001$, $\beta = 5 \times 10^{-4}$, $\gamma = 2 \times 10^{-4}$, $\delta = 3 \times 10^{-4}$, $\epsilon = 1 \times 10^{-4}$, $\zeta = 5 \times 10^{-5}$, $\pi_A = 0.35$, $\pi_G = 0.17$, $\pi_C = 0.25$, and $\pi_T = 0.23$.
   (a) Build the corresponding rate matrix.
   (b) Compute the probability matrix for $t = 1$.
   (c) Find a method to simulate the evolution of a DNA sequence under this GTR model for an arbitrary t.
   (d) What are the expected base frequencies when t is very large?

3. Sketch a function doing Bayesian estimation of phylogeny. The code should include comments explaining the rationale of the choices.

4. Take the data prepared in Exercise 5 of Chapter 3.
   (a) Build saturation diagrams for the whole sequence, and for each codon position.
   (b) Examine graphically the effects of unequal transition and transversion rates and / or unequal base frequencies on the distance estimates for each data set (whole sequences and each codon position).

5. Analyze the data prepared in Exercise 6 of Chapter 3. Use the function `phymltest`, and compare the results with those from the Felidae analysis above.

6

Analysis of Macroevolution with Phylogenies

Reconstructing the history of species is a necessary step in understanding the mechanisms of biological evolution. Once a phylogeny has been estimated, a lot of questions on how species have evolved can be addressed. Why are some taxonomic groups more diverse than others? How have species traits evolved? Have some traits favored diversification? Are some traits linked through evolution?

By contrast to the field of molecular evolution which is recent in the history of sciences, these questions are old issues that were already lively debated in the nineteenth century. The remarkable development of phylogenetics during the past decades has renewed interest in these long-standing issues, and led to the development of new analytical methods to address them. This chapter presents these methods. Their common feature is that they take an estimated tree as raw data. The first section presents methods to analyze species data in a phylogenetic framework, the second one, methods that estimate ancestral characters, and the third one, methods to analyze diversification above the species level. All sections consider, in most cases, ultrametric trees with dated nodes as a key element of raw data.

6.1 Phylogenetic Comparative Methods

Comparing observations made on different species is an intuitive and appealing approach that certainly dates back to antiquity [64]. For instance, if some combinations of traits are consistently associated across several species, this could suggest that evolutionary forces, such as selection, shaped these associations. However, nonrandom associations of some traits among some species may be due to common heritage from their ancestor, and thus concomitant change through time cannot be inferred [131]. Conversely, if characters have evolved randomly without association, more closely related species are more likely to be similar than others, thus creating apparent relationships among characters [36].

It is consequently necessary to consider the phylogenetic relationships among species when analyzing their characters. Several attempts in this direction have been made early on by considering partial phylogenetic information such as taxonomic information (see [64] for a review). With the growing availability of complete phylogenies with estimated branch lengths, it is now possible to go further [36].

From an analytical perspective, two issues may be addressed when incorporating phylogeny into comparative data:

- Taking interspecies nonindependence into account when studying traits and their relationships, and
- Estimating the parameters of character evolution.

Both issues are tightly connected. It is indeed important to realize that the impact of phylogeny on trait distributions depends not only on phylogeny but also on the way these traits evolve.

Particular emphasis has been given to the first issue because traditional comparative methods (i.e., without phylogeny) have been widely used for decades [65]. The methods devised to "correct for phylogenetic dependence" usually assume a simple model of character evolution: Brownian motion for continuous characters, or parsimonious change for discrete ones. However, even if these models do not apply to a particular situation, phylogeny is still important in the distribution of species traits [62].

When estimating parameters of character evolution, a model must be formulated explicitly and fit to the data (the characters and the tree), usually by maximum likelihood. Several models can be fit to the same data set and compared with the usual statistical techniques (e.g., likelihood ratio tests, or information criteria).

Table 6.1 lists the methods currently available in `ape` and `ade4` together with their main features. Most of these methods do not specifically require an ultrametric tree: different sets of branch lengths may be used implying different assumptions on rates of evolution [48, 54]. The branch lengths may be modified, or even created if the tree has none, with `compute.brlen`.

Table 6.1. Comparative methods implemented in R and their main features

	PIC	Auto-regres.	Auto-correl.	Multiv. decomp.	GLS	GEE	Mixed	OU
Correct for phylo. dependence	✓	✓	✓	✓	✓	✓		
Estimate evol. parameters							✓	✓
Univariate	✓	✓	✓					✓
Relationships among variables	✓				✓	✓		
Continuous variables	✓	✓	✓	✓	✓	✓	✓	✓
Categorical variables						✓		
Allow multichotomies		✓	✓	✓	✓	✓	✓	✓

6.1.1 Phylogenetically Independent Contrasts

Felsenstein [36] was probably the first to propose a method that fully takes phylogeny into account in the analysis of comparative data. The idea behind the "contrasts"[1] method is that, if we assume that a continuous trait evolves randomly in any direction (i.e., the Brownian motion model), then the "contrast" between two species is expected to have a distribution centered on zero, and a variance proportional to the time since divergence. If the contrasts are scaled with the latter, then they have a variance equal to one.

A contrast is computed with [36]:

$$C_{ij} = \frac{x_i - x_j}{\sqrt{d_{ij}}} \,, \tag{6.1}$$

where x_i and x_j are the values of the trait observed on species i and j, and the distance between both species d_{ij} is measured on the tree. This is straightforward if x_i and x_j are observed on recent species, but this can be done also for internal nodes because under the assumptions of the Brownian model the ancestral state of the variable can be calculated; a rescaling of the internal branches eventually occurs [36].

In this formulation, the tree needs to be binary (fully dichotomous), and a contrast is computed for each node. Thus for n species, $n-1$ contrasts will be computed. The contrasts are independent with respect to the phylogeny (unlike the original values of x), and standard statistical methods for continuous variables can be used.

The method of phylogenetically independent contrasts (PICs), is implemented in the function `pic`. This function computes the PICs giving a tree and a vector of values. The result is a vector of numeric values with the computed PICs.

As a simple example we take a data set analyzed by Lynch [94] consisting of the log-transformed body mass and longevity of five species of primates.

```
> tree.primates <- read.tree("primfive.tre")
> body <- c(4.09434, 3.61092, 2.37024, 2.02815, -1.46968)
> longevity <- c(4.74493, 3.3322, 3.3673, 2.89037, 2.30259)
> names(body) <- names(longevity) <- c("Homo",
+               "Pongo", "Macaca", "Ateles", "Galago")
> pic.body <- pic(body, tree.primates)
> pic.longevity <- pic(longevity, tree.primates)
> pic.body
       -1        -2        -3        -4
3.3583189 1.1929263 1.5847416 0.7459333
```

[1] Phylogenetically independent contrasts, often called "contrasts" in the phylogenetic literature, are related to the *statistical contrasts* used in analysis of variance and other methods (see `?contrasts` in R) in the sense that they both consider contrasts in expected means.

```
 > pic.longevity
      -1        -2        -3        -4
0.8970604 0.8678969 0.7176125 2.1798897
```

We plot the tree and show the values of the PICs with `nodelabels` (Fig. 6.1):

```
plot(tree.primates)
nodelabels(round(pic.body, 3), adj = c(0, -0.5),
           frame = "n")
nodelabels(round(pic.longevity, 3), adj = c(0, 1),
           frame = "n")
```

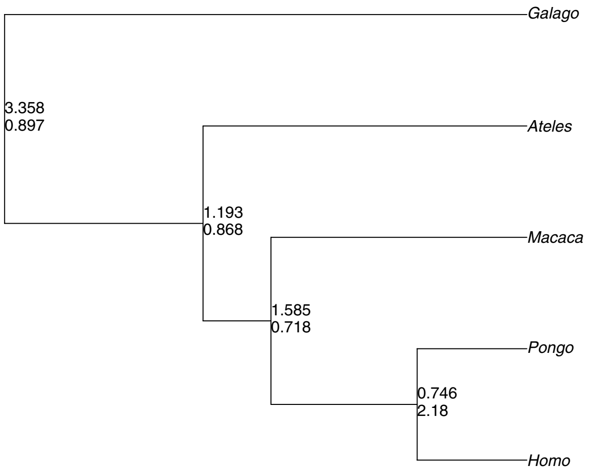


Fig. 6.1. A tree of five primate genera showing phylogenetically independent contrasts of ln(body mass) and ln(longevity), above and below, respectively

A plot of the two sets of PICs shows no clear relationship between them (Fig. 6.2):

```
plot(pic.body, pic.longevity)
abline(a = 0, b = 1, lty = 2) # x = y line
```

This is confirmed by a correlation and a simple regression:

```
> cor(pic.body, pic.longevity)
[1] -0.5179156
> lm(pic.longevity ~ pic.body)

Call:
lm(formula = pic.longevity ~ pic.body)
```

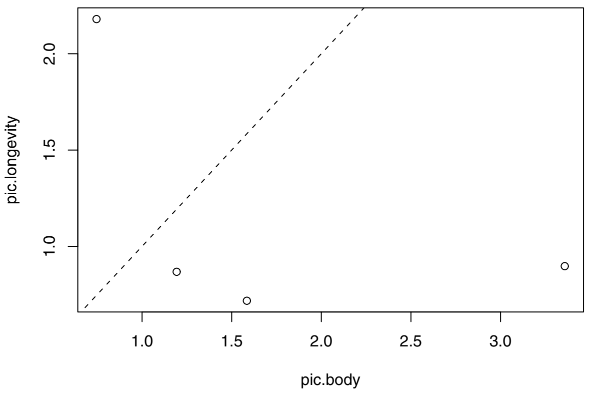


Fig. 6.2. Plot of the four pairs of contrasts from Fig. 6.1; the dashed line is $x = y$

```
Coefficients:
(Intercept)      pic.body
     1.6957       -0.3081
```

Garland et al. [48] recommended that linear regressions with PICs should be done through the origin (i.e. the intercept is set to zero). It is clear from Fig. 6.2 that the result will be different if their suggestion is followed:

```
> lm(pic.longevity ~ pic.body - 1)

Call:
lm(formula = pic.longevity ~ pic.body - 1)

Coefficients:
 pic.body
0.4319
```

None of the above coefficients is significantly different from zero which is hardly surprising considering the small sample size. Doing the regression among PICs through the origin is justified if the characters evolve under a Brownian motion model and there is a linear relation between them [48]. However, this is likely to ignore a possible nonlinear relationship [127]. In all cases, it seems wise to plot the PICs as done here.

Purvis and Garland [124] introduced a modification of Felsenstein's [36] method in order to take multichotomies into account. This is not implemented in the function `pic`, but this may done by combining this function with others such as `multi2di` (Section 3.4.3). There are alternative approaches, such as generalized least squares, to cope with multichotomies with continuous traits (Section 6.1.5).

6.1.2 Phylogenetic Autoregression

If it is postulated that species are not independent through their phylogenetic relationships, then the latter may be used to quantify the association between the variables observed on the species. This approach was used by Cheverud, Dow, and Leutenegger [18] and later refined by Rohlf [132]. This is based on the following model:

$$x = \rho W x + \epsilon \ , \tag{6.2}$$

where x is the studied variable, W is a connectivity matrix based on the phylogeny, ρ is a parameter, and ϵ is the variation not explained by the phylogeny. The rows of W sum to one and the values indicate the “distance” between the different species (the diagonal elements are thus equal to zero). The parameter ρ is estimated from the data: positive values indicate an influence of the phylogeny on x, whereas negative values indicate the opposite (distantly related species are more identical). If $\rho = 0$, then the phylogeny has no influence on x. The variation in x explained by the phylogeny can be calculated as [18]:

$$R^2 = 1 - \frac{Var(\epsilon)}{Var(x)} \ . \tag{6.3}$$

Cheverud et al.’s [18] method, including Rohlf’s [132] correction, is implemented in the function `compar.cheverud`. This function takes as arguments a numeric vector, and a matrix that is transformed to give the connectivity matrix. The matrix given to the function could be a correlation matrix (obtained with `vcv.phylo`), or a distance matrix (obtained with `cophenetic`): the results will be the same.

Let us consider again the small primate data set. The correlation matrix is obtained with the function `vcv.phylo`:

```
> W <- vcv.phylo(tree.primates, cor = TRUE)
> CM.prim <- compar.cheverud(body, W)
> CM.prim
$rhohat
[1] -2.623383

$Wnorm

               Homo      Pongo     Macaca    Ateles    Galago
  [1,] 0.00000000 0.09051724 0.2112069 0.2672414 0.4310345
  [2,] 0.09051724 0.00000000 0.2112069 0.2672414 0.4310345
  [3,] 0.18846154 0.18846154 0.0000000 0.2384615 0.3846154
  [4,] 0.21678322 0.21678322 0.2167832 0.0000000 0.3496503
  [5,] 0.25000000 0.25000000 0.2500000 0.2500000 0.0000000

$residuals
            [,1]
Homo   -1.681081
Pongo  -2.049707
Macaca -1.740552
Ateles -1.296137
Galago -1.237742
```

The result is a list with three elements: the estimated value of ρ (`rhohat`), the normalized matrix W (`Wnorm`), and the estimated residuals ϵ_i (`residuals`). The proportion of variation explained by the phylogeny is thus:

```
> 1 - var(CM.prim$residuals) / var(body)
          [,1]
[1,] 0.9763006
```

This analysis suggests a negative influence of phylogeny on the distribution of body mass in these primates. This is quite nonintuitive, but looking at the contrasts calculated in the previous section, we can see they are all positive. This suggests that there is a trend in the evolution of body mass, and thus the Brownian motion model does not apply.

6.1.3 Autocorrelative Models

Gittleman and Kot [53] introduced a method close to Cheverud et al.'s [18] but based on an autocorrelation approach. This uses Moran's autocorrelation index I [102]:

$$I = \frac{n}{S_0} \frac{\displaystyle\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}(x_i - \bar{x})(x_j - \bar{x})}{\displaystyle\sum_{i=1}^{n}(x_i - \bar{x})^2} , \tag{6.4}$$

$$S_0 = \sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij} , \tag{6.5}$$

where w_{ij} is the distance between species i and j, and $\bar{x}$ is the observed mean of x. This is somehow similar to the correlation between two variables, but instead looks at different values of the same variables (in the present context, made on different species), and where each pair is weighted with w. Because it is expected that more closely related species are more similar, the latter can be derived from the phylogeny. Gittleman and Kot [53] proposed that in the absence of an accurate phylogeny, the weights can be derived from the taxonomy.

In the absence of phylogenetic autocorrelation, the mean expected value of I and its variance are known [53]. It is thus possible to test the null hypothesis of the absence of dependence among observations.

Gittleman and Kot's [53] method is implemented in the function `Moran.I`. Considering the primate small data set, the distances between species can be computed with the function `cophenetic`:

```
> Moran.I(body, cophenetic(tree.primates))
$observed
-0.4250254

$expected
[1] -0.25

$sd
[1] 0.0743147

$p.value
0.01851316
```

The result is a list with four elements: the observed value of I (`observed`), its expected value under the null hypothesis of no correlation (`expected`), the standard-deviation of the observed I (`sd`), and the P-value of the null hypothesis (`p.value`).

In agreement with the autoregression analysis, a negative autocorrelation was found. Note that the expected value is negative (-0.25): this is not really intuitive, but in the absence of correlation among observations, the expected value of Moran's autocorrelation coefficient is negative (see [102]).

`ade4` has the function `gearymoran` that computes Moran's coefficient and tests its significance with a randomization procedure. The two main arguments of this function are a distance matrix and a data frame with one or several vectors. The option `nrepet` specifies the number of replications of the randomization test (999 by default). We leave this option as its default for the present analysis:

```
> gearymoran(cophenetic(tree.primates),
+            data.frame(body, longevity))
class: krandtest
test number:   2
permutation number:   999
  test      obs    P(X<=obs) P(X>=obs)
1 body      -0.423 0.014     1
2 longevity -0.339 0.166     0.849
```

The result for body mass is very close to the one with `Moran.I`. This latter function gives with longevity:

```
> Moran.I(longevity, cophenetic(tree.primates))
$observed
[1] -0.3182082

$expected
[1] -0.25

$sd
[1] 0.0734518

$p.value
[1] 0.3530901
```

For this variable, the computed coefficients are very close between both functions, but the P-values are somehow different although both not significant.

Gittleman and Kot [53] suggested the use of correlograms to visualize the results of phylogenetic autocorrelative analyses. The idea is to look at the correlation at different distance categories. This can be done even in the absence of a complete phylogeny using taxonomic levels. If a phylogeny is available, then at least two distance categories must be defined. Both methods (with taxonomic levels or with a phylogeny) are implemented in two functions: `correlogram.formula` and `correlogram.phylo`, respectively. The options in these two functions are slightly different.

As an example, we take the data compiled by Gittleman [52] on 112 species of carnivores. This includes various life-history variables as well as taxonomic levels (species, genus, family, super-family, and order). We consider (as in [53]) the correlation levels in mean body mass at the various taxonomic levels. The function `correlogram.formula` requires a formula where the levels are separated with slashes:[2]

```
> data(carnivora)
> correl.carn <- correlogram.formula(
+        log10(SW) ~ Order/SuperFamily/Family/Genus,
+        data = carnivora)
> correl.carn
$obs
[1]  0.614371364  0.404715752 -0.266621894 -0.001377008

$p.values
[1] 9.529087e-07 0.000000e+00 0.000000e+00 5.432094e-01
```

[2] This is the usual notation to specify nested effects in R's formulae.

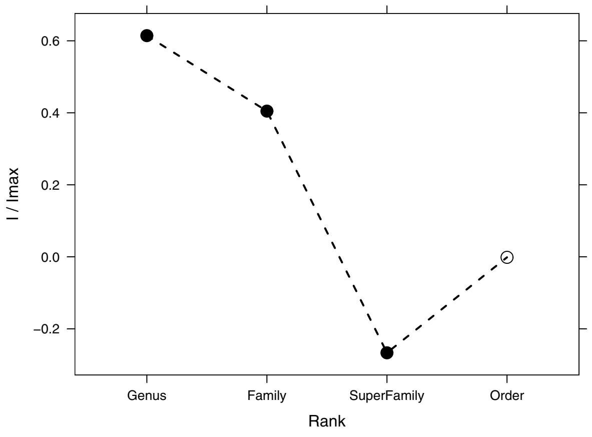


Fig. 6.3. Phylogenetic correlogram of ln(body mass) among 112 species of carnivores; the filled circles indicate the significant coefficients ($P < 0.05$)

```
$labels
[1] "Genus"       "Family"      "SuperFamily" "Order"

attr(,"class")
[1] "correlogram"
```

The returned object is of class `"correlogram"`; there is a `plot` method for this class (Fig. 6.3):

```
plot(correl.carn)
```

The correlation coefficient at the "Genus" level is computed among pairs of species belonging to the same genus, and the same for those at the "Family", "SuperFamily", and "Order" levels.

6.1.4 Multivariate Decomposition

Multivariate methods can be used to summarize the structure of phylogenetic trees leading to possible measures of phylogenetic dependence. Diniz-Filho, de Sant'Ana, and Bini [26] developed a method they called phylogenetic eigenvector regression (PVR). Its principle is to do an eigen decomposition of the doubly centered matrix of among-species distances. A regression of the studied variable is then made on the matrix of eigenvectors. Diniz-Filho et al. [26] recommended first running a phylogenetic autocorrelation analysis (Section 6.1.3) to test for the presence of significant phylogenetic dependence. If the test is significant, this dependence may be quantified with PVR: the number of eigenvectors used in the regression is selected according to the expectation under a broken-stick model.

Ollier, Couteron, and Chessel [108] proposed a related approach that differs substantially in the details. Instead of using a distance matrix, they use a matrix built from the topology of the tree. They then perform an orthonormal transform on this matrix leading to a matrix that is a linear combination of their original matrix. They finally perform an eigen decomposition of the last matrix, keeping only the eigenvectors with positive eigenvalues on which the studied variable is regressed.

The function `variance.phylog` in package `ade4` implements Ollier et al.'s [108] method: it takes as main arguments an object of class `"phylog"` and a numeric vector. To perform the analysis with the primates data we first need to transform the tree of class `"phylo"` into one of class `"phylog"` (Section 3.4.5):

```
> tpg <- newick2phylog(write.tree(tree.primates))
> variance.phylog(tpg, body)
$lm

Call:
lm(formula = fmla, data = df)

Coefficients:
(Intercept)           A1           A2
  2.139e-16   -8.685e-01   -1.371e-01



$anova
Analysis of Variance Table

Response: z
          Df Sum Sq Mean Sq F value  Pr(>F)
A1         1 3.7719  3.7719 56.2198 0.01733
A2         1 0.0940  0.0940  1.4005 0.35825
Residuals  2 0.1342  0.0671

$sumry
             Df Sum Sq  Mean Sq F value  Pr(>F)
Phylogenetic 2  3.86582 1.93291 28.81013 0.03355
Residuals    2  0.13418 0.06709
```

The test of the phylogenetic dependence (or inertia) corresponds to the test of the linear model with the selected eigenvectors as predictors. We thus conclude with a significant phylogenetic inertia for body mass. The same analysis with longevity gives:

```
> variance.phylog(tpg, longevity)
```

```
$lm

Call:
lm(formula = fmla, data = df)

Coefficients:
(Intercept)           A1           A2
 -2.958e-16   -7.305e-01    1.226e-01


$anova
Analysis of Variance Table

Response: z
          Df  Sum Sq Mean Sq F value Pr(>F)
A1         1 2.66807 2.66807  4.2460 0.1755
A2         1 0.07520 0.07520  0.1197 0.7624
Residuals  2 1.25673 0.62837

$sumry
             Df Sum Sq  Mean Sq F value Pr(>F)
Phylogenetic 2  2.74327 1.37163 2.18285 0.31418
Residuals    2  1.25673 0.62837
```

The test is in agreement with the results from the autocorrelation analysis.

Desdevises et al. [24] proposed a method close to Diniz-Flihol et al.'s [26]: instead of selecting the eigenvectors according to a broken-stick model, they suggested selecting all statistically significant eigenvectors in the regression.

Giannini [50] proposed a method with a matrix coding the tree structure similar to the one used by Ollier et al. [108]: he then performed a linear regression of the studied variable on this matrix. The best subset of the "tree" matrix was selected using Monte Carlo permutations.

6.1.5 Generalized Least Squares

The method of generalized least squares (GLS) can be seen as an extension of the method of ordinary least squares. With the latter, observations are assumed to have the same variance, and covariances equal to zero. These assumptions are relaxed with GLS.

The use of GLS in comparative methods came as a way to generalize the contrasts approach. Grafen [54] first proposed this approach as a way to deal with multichotomies in trees and also as a way to integrate more complex models of multi-character evolution. He suggested a model where each node is given a height equal to the number of tips minus one; these heights are then scaled so that the root has height one and the other heights are raised to power

ρ (with $\rho > 0$). Grafen's model is actually similar to a Brownian motion model with modified branch lengths. Under a Brownian motion model of character evolution, the covariance between species i and j, denoted v_{ij}, is given by:

$$v_{ij} = \sigma^2 T_a \ , \tag{6.6}$$

where T_a is the distance between the root and the most common recent ancestor of species i and j, and σ^2 is the variance of the Brownian process.

Martins and Hansen [96] suggested the Ornstein–Uhlenbeck model where the covariance between two species is given by:

$$v_{ij} = \sigma^2 \exp(-\alpha d_{ij}) \ , \tag{6.7}$$

where σ^2 is similar to the variance of the Brownian process, α specifies how "fast" the species character diverge after speciation, and d_{ij} is the distance between both species. We show the Ornstein–Uhlenbeck model again in Section 6.1.8.

The function `gls` in package `nlme` is used to fit models with GLS. This is a very general function that can accomodate correlation among observations and heterogeneous variance functions. The former is specified with an object of class `"corStruct"` (*correlation structure*): the variance–covariance matrix is then generated during the analysis through several functions called internally by `gls`.

Julien Dutheil introduced the idea of using the correlation structures used in the package `nlme` to code phylogenetic correlation structures. The three models sketched above are specified with the functions `corGrafen`, `corBrownian`, and `corMartins`, respectively:

```
corGrafen(value, phy, fixed = FALSE)
corBrownian(value = 1, phy)
corMartins(value, phy, fixed = FALSE)
```

where `value` is the parameter of the model, `phy` is an object of class `"phylo"`, and `fixed` a logical indicating whether to estimate the parameters from the data (the default). These functions return an object of class with three elements:

1. The name of the called function (i.e., `"corGrafen"`, `"corMartins"`, or `"corBrownian"`);
2. `"corPhyl"`;
3. `"corStruct"`.

The last one is important because it allows us to fit these models with `gls`.[3] An evolutionary model is then fit as is any linear model with GLS. For instance, coming back to the primate data, we first create a correlation structure that follows a Brownian motion model:

[3] The package `nlme` is loaded when `ape` is started.

```
bm.prim <- corBrownian(phy = tree.primates)
```

We then fit the linear model where longevity is a function of body mass. A small data manipulation is required by creating a data frame that includes the studied variables to ease the way they are passed to `gls`:[4]

```
DF.prim <- data.frame(body, longevity)
```

We can now fit the model:

```
m1 <- gls(longevity ~ body, correlation = bm.prim,
          data = DF.prim)
```

We extract the details of the model fit with `summary`:

```
> summary(m1)
Generalized least squares fit by REML
  Model: longevity ~ body
  Data: DF.prim
       AIC      BIC   logLik
  17.48072 14.77656 -5.74036

Correlation Structure: corBrownian
 Formula: ~1
 Parameter estimate(s):
numeric(0)

Coefficients:
                Value Std.Error  t-value p-value
(Intercept) 2.5000672 0.7754516 3.224014  0.0484
body        0.4319328 0.2864904 1.507669  0.2288

....
```

Note that no parameter is estimated in the present correlation structure, hence the output `numeric(0)`. In contrast to what was suggested by the plot of PIC values (Section 6.1.1), the relationship between variables now appears positive although not statistically significant. This underlines the difference between both methods: GLS focuses on the relationship between variables, whereas the PIC method focuses on the relationship between contrasts (i.e., between changes in the variables through the phylogeny). The two present variables are indeed strongly positively correlated:

```
> cor(body, longevity)
[1] 0.8296107
```

[4] When this data frame is created, the names of the vectors are used as rownames (see p. 16); the latter are then matched with the tip labels of the tree, even if they are not in the same order.

We now fit the Ornstein–Uhlenbeck model based on Martins and Hansen's correlation structure to the same data:

```
> ou.prim <- corMartins(1, tree.primates)
> m2 <- gls(longevity ~ body, correlation = ou.prim,
+           data = DF.prim)
> summary(m2)
Generalized least squares fit by REML
  Model: longevity ~ body
  Data: DF.prim
       AIC      BIC    logLik
  17.81707 14.21152 -4.908536

Correlation Structure: corMartins
 Formula: ~1
 Parameter estimate(s):
   alpha
51.55332

Coefficients:
                Value Std.Error  t-value p-value
(Intercept) 2.5989768 0.3843447 6.762099  0.0066
body        0.3425349 0.1330977 2.573561  0.0822

....
```

The AIC value does not indicate an improvement compared to the Brownian model. Not surprisingly, the parameter estimates are very close in these two models.

6.1.6 Generalized Estimating Equations

The use of generalized estimating equations (GEEs) for the analysis of comparative data had two motivations: to deal easily with multichotomies, and to analyze categorical variables in a natural way [116].

GEEs were introduced by Liang and Zeger [92] as an extension of generalized linear models (GLMs) for correlated data. The correlation structure is specified through a correlation matrix. Similarly to GLMs, the model is specified with a link function g:

$$g(E[y_i]) = x_i^T \beta \, , \tag{6.8}$$

However, the distinction comes from the way the variance–covariance matrix is given:

$$V = \phi A^{1/2} R A^{1/2} \, , \tag{6.9}$$

where A is an $n \times n$ diagonal matrix defined by diag$\{\mathcal{V}(E[y_i])\}$: that is, a matrix with all its elements zero except the diagonal which contains the variances of the n observations expected under the (marginal) GLM, R is the correlation matrix of the elements of y, ϕ is the scale (or dispersion) parameter, and $\mathcal{V}(E[y_i])$ is the variance function. These two components, ϕ and $\mathcal{V}(E[y_i])$, are defined with respect to the distribution assumed for y in the same way as in a standard GLM. If the observations are independent, then R is an $n \times n$ identity matrix.

Beyond the technicalities of the GEE approach lies the possibility of analyzing different kinds of variables thanks to the GLM framework. The analysis is done with the function `compar.gee`. This uses the same interface as `glm`: the model is given as a formula, and the distribution of the response is specified with the option `family`. By default this option is `"normal"`, thus we do not need to use it for the small primate data:

```
> compar.gee(longevity ~ body, phy = tree.primates)
[1] "Beginning Cgee S-function, @(#) geeformula.q 4.13 98/01/27"
[1] "running glm to get initial regression estimate"
[1] 2.5989768 0.3425349

Call:
  formula: longevity ~ body

Number of observations:  5

Model:
 Link:                      identity
 Variance to Mean Relation: gaussian

Summary of Residuals:
       Min         1Q     Median         3Q        Max
-0.7275418 -0.4857216 -0.1565515  0.4373258  0.4763833


Coefficients:
             Estimate      S.E.        t Pr(T > |t|)
(Intercept) 2.5000672 0.4325167 5.780279  0.06773259
body        0.4319328 0.1597932 2.703074  0.17406821

Estimated Scale Parameter:  0.4026486
"Phylogenetic" df (dfP):  3.32
```

The output from `compar.gee` is very close to the one from `gee`; the former additionally prints the phylogenetic number of degrees of freedom (df_P). Some simulations showed that if the statistical tests on the regression parameters are done with a t-test with the usual residual number of degrees of freedom, then type I error rates are inflated [116]. A solution to this problem is to correct the number of degrees of freedom with:

$$df_P = \frac{\sum_{\text{tree}} \text{branch length}}{\sum_{i=1}^{n} \text{distance from root to tip}_i} \times n \ , \tag{6.10}$$

where n is the number of species in the tree. This correction was found empirically, and works in practice, but it still needs to be confirmed theoretically, and possibly refined.

6.1.7 Mixed Models and Variance Partitioning

In the literature on comparative methods, some emphasis is put on relationships among variables: many comparative analyses are motivated by establishing relationships among ecological or physiological variables [45, 46]. Lynch [94] pointed out that these approaches do not consider all the available information on the evolutionary process. He suggested rather to shift the attention on (co)variation of the traits by using an approach close to one used in quantitative genetics to assess the different components of genetic variation. He proposed the following model:

$$x_i = \mu + a_i + e_i \ , \tag{6.11}$$

where μ is the grand mean of the trait, a_i comes from a normal distribution with a variance–covariance matrix $\sigma_a^2 G$ where G is a correlation matrix derived from the phylogeny (we can write this as $a \sim \mathcal{N}(0, \sigma_a^2 G)$), and the e_is are independent normal variables so that $e \sim \mathcal{N}(0, \sigma_e^2)$. This univariate model can be extended to several variables in which case there are additional parameters, Cov_a and Cov_e, namely the covariance explained by the phylogeny and the residual covariance, respectively [94].

Lynch [94] proposed an expectation–maximization (EM [23]) algorithm to fit model (6.11) by maximum likelihood but this is very slow and becomes intractable with large sample sizes. Housworth et al. [70] proposed a reparameterization of (6.11) and a new algorithm to remedy this problem, but this applied only to uni- and bivariate cases.

Fitting model (6.11) is actually a difficult task. A possible explanation may be because both components of variance are confounded, and cannot be estimated separately. In mixed-effects models, variance components are usually estimated with different groups that are statistically independent, but observations within groups can be correlated [119]. With phylogenetic data, there is only one group, and thus σ_a^2 and σ_e^2 are confounded.

The function `compar.lynch` uses the EM algorithm proposed by Lynch [94] to fit model (6.11). We illustrate its use with the small primate data set. We first build a correlation matrix in the way seen previously:

```
> G <- vcv.phylo(tree.primates, cor = TRUE)
> compar.lynch(cbind(body, longevity), G = G)
$vare
           [,1]      [,2]
[1,] 0.04908818 0.1053366
[2,] 0.10533661 0.2674316

$vara
                body longevity
body       3.0018670 0.9582542
longevity 0.9582542 0.3068966

$A
           [,1]       [,2]
[1,]  2.5056671  0.8006949
[2,]  2.5705959  0.8201169
[3,]  1.1485439  0.3663313
[4,]  0.9654236  0.3065841
[5,] -2.7534270 -0.8779460

$E
            [,1]        [,2]
[1,]  0.34915743  0.89988706
[2,] -0.19919129 -0.53226494
[3,] -0.01781930 -0.04337929
[4,] -0.17678902 -0.46056213
[5,]  0.04423158  0.13618796

$u
     body longevity
 1.239433  3.044322

$lik
          [,1]
[1,] -12.21719
```

The results are returned as a list with five elements:

`vare`: the estimated residual variance–covariance matrix;
`vara`: the estimated additive effect variance–covariance matrix;
`u`: the estimates of the phylogeny wide means;
`A`: the additive value estimates;
`E`: the residual value estimates;
`lik`: the log-likelihood.

6.1.8 The Ornstein–Uhlenbeck Model

The Brownian motion model assumes that continuous characters could diverge indefinitely after divergence from the same values. A more realistic model would be one where characters are constrained to evolve around a given value. A candidate model is the Ornstein–Uhlenbeck (OU) model. The quantity of character change along a short time interval dt according to a general OU model is [7, 86]:

$$\mathrm{d}x_t = -\alpha(x_t - \theta)\mathrm{d}t + \mathrm{d}\epsilon_t \ , \tag{6.12}$$

where α controls the strength of character evolution towards the "optimum" value θ, and $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$. If $\alpha = 0$, the OU model reduces to a Brownian motion model. A discrete-time version of (6.12) is:

$$x_{t+1} = -\alpha(x_t - \theta) + \epsilon_t \ . \tag{6.13}$$

It is straightforward to simulate an OU model in R using (6.13). If we set $\alpha = 0$ and $\theta = 0$, then we simulate a Brownian motion model with zero as initial value and $\sigma^2 = 1$, on 99 time-steps with:

```
x <- cumsum(c(0, rnorm(99)))
```

The OU equivalent with $\alpha = 0.2$ and $\theta = 0$ would be:

```
x <- numeric(100)
for (i in 2:100)
  x[i] <- -0.2 * x[i - 1] + rnorm(1)
```

To replicate the Brownian motion simulation, say five times, we can use the following code:

```
X <- replicate(5, cumsum(c(0, rnorm(99))))
```

For the OU version of this code, we first create a function that includes the commands above:

```
sim.ou <- function() {
  x <- numeric(100)
  for (i in 2:100)
    x[i] <- -0.2 * x[i - 1] + rnorm(1)
  x # returns the value of x
}
```

The function can then be used in the same way as above:

```
X2 <- replicate(5, sim.ou())
```

It is interesting to look at the variance of the five replicates of each model:

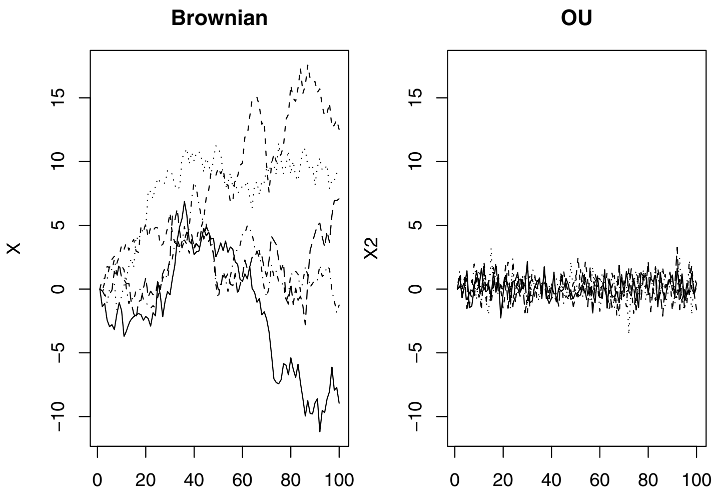


Fig. 6.4. Simulations with five replicates of the Brownian motion (left) and Ornstein–Uhlenbeck models (right)

```
> var(X[100 ,])
[1] 75.83865
> var(X2[100 ,])
[1] 0.8434638
```

A plot of the simulated values shows even more clearly the contrast between both models (Fig. 6.4):

```
layout(matrix(1:2, 1, 2))
yl <- range(X)
matplot(X, ylim = yl, type = "l", col = 1, main = "Brownian")
matplot(X2, ylim = yl, type = "l", col = 1, main = "OU")
```

The function `compar.ou` fits a general OU model where θ may vary through the phylogeny [61]. The interface is:

```
compar.ou(x, phy, node = NULL, alpha = NULL)
```

where `x` is a numeric variable, `phy` is a tree (as an object of class `"phylo"`), `node` specifies the nodes where θ changes, and `alpha` is the value of α. The latter parameter is assumed to be constant throughout the phylogeny; only the optimum θ can change. When a node number is given in `node`, then it is assumed that the optimum changes at this point for all branches from this node. By default (i.e., if `node = NULL`), it is assumed that θ is the same for all branches.

By default, α is estimated from the data but this is not usually a good idea as the estimation is unstable. It is preferable to give a fixed value when fitting the model. Hansen [61] made similar observations on the instability of the estimates of α.

As a simple example with the primate data, we fit an OU model to the longevity data using $\alpha = \{0.2, 2\}$:

```
> compar.ou(longevity, tree.primates, alpha = .2)
$deviance
[1] 17.87657

$para
       estimate    stderr
sigma2 8.218722 3.6762567
theta1 2.448405 0.4280387

$call
compar.ou(x = longevity, phy = tree.primates, alpha = 0.2)

> compar.ou(longevity, tree.primates, alpha = 2)
$deviance
[1] 12.42138

$para
        estimate    stderr
sigma2 0.7484398 0.3348018
theta1 3.0805691 0.3127302

$call
compar.ou(x = longevity, phy = tree.primates, alpha = 2)
```

The function returns the deviance ($-2 \times$ log-likelihood) of the model, the parameter estimates with their standard errors, and the function call recalling the fitted model. This example shows that the model with $\alpha = 2$ fits better because its deviance is smaller, indicating that there is substantial constraint in the evolution of longevity. The estimated optimum, with its 95% confidence interval, is $\hat{\theta} = 3.08 \pm 0.62$, and the estimated variance of the OU process is $\hat{\sigma}^2 = 0.74 \pm 0.67$. The estimates of α and σ^2 are highly correlated which could be the result of the small sample size.

6.1.9 Perspectives

Comparative methods have enjoyed great success during the past 20 years, both in terms of methodological and conceptual developments, and in terms of empirical applications. Much emphasis has been put on correcting for phylogenetic dependence in order to use standard statistical methods. There is surely some gain in shifting attention to estimating evolutionary parameters inasmuch as the essence of comparative data is the evolutionary processes that generated them. In this respect, the Ornstein–Uhlenbeck model is likely to be an interesting alternative to the commonly used Brownian motion model [16].

R offers a wide range of phylogenetic comparative methods. Some methods not discussed here are:

- Garland et al. [47] developed a method based on simulations.
- Read and Nee [130] developed a method for the analysis of binary traits (e.g., presence or absence).
- Grafen and Ridley [55, 56, 57] developed similar methods for discrete characters.
- Huelsenbeck et al. [72] developed a Bayesian method to take phylogeny uncertainty into account.

These methods can be easily programmed in R.

6.2 Estimating Ancestral Characters

For some time, the estimation of ancestral characters was considered as a component of phylogeny estimation with parsimony methods where deriving ancestral and derived characters is an essential step [39]. With the development of alternative methods where ancestral character values are not necessary (distance methods) or their probabilistic distribution is taken into account (likelihood methods), the estimation of ancestral values has become less critical in phylogeny estimation.

The use of phylogenies to test evolutionary hypotheses has created new interest in estimating ancestral character values. Many issues depend on how characters evolved from an ancestral value [42]. Some researchers have focused their attention on statistical methods of ancestral character estimation where uncertainty in the estimates is taken into account [107]. Ancestral character values are not observed, and thus it is more rational to consider them as parameters in a model where the character values of recent species are the observed variables. Consequently, the word "estimation" is preferable to "reconstruction". In the same way, it is better to write "character values" rather than "character states" inasmuch as we consider both continuous and discrete characters ("state" implicitly refers to discrete characters).

`ape` has a single function to perform ancestral character estimation: `ace`. By default, `ace` performs estimation for continuous characters assuming a Brownian motion model fit by maximum likelihood. The options of `ace` have different effects depending on the types of character under study. In all cases a fully resolved phylogeny is required.

6.2.1 Continuous Characters

Two methods can be used for continuous characters: least squares (`method = "pic"`), and maximum likelihood (`method = "ML"`, the default). The model of evolution is specified with the option `model`.

The least squares estimator follows from the phylogenetically independent contrasts method [36] (Section 6.1.1). This assumes a Brownian motion model of evolution: this allows us to compute the variance of each ancestral character estimate. A confidence interval can be computed with the usual formula $\hat{x}_a \pm 1.96\sqrt{V(\hat{x}_a)}$, with $\hat{x}_a$ being the estimated ancestral value.

The maximum likelihood estimator under a Brownian motion model developed by Schluter et al. [138] uses a likelihood function where the ancestral values are parameters:

$$L(\sigma^2, x_a|\mathcal{T}, x) = \frac{1}{\sigma^n} \exp\left(\frac{1}{2\sigma}\sum \frac{(x_i - x_j)^2}{t_{ij}}\right) , \tag{6.14}$$

where σ^2 is the variance of the Brownian motion process, x_a are the ancestral values, $\mathcal{T}$ is the phylogeny, and x are the observed values of the character at the tips of $\mathcal{T}$. Once (6.14) has been maximized, the standard errors of σ^2 and $\hat{x}_a$ are obtained with the second partial derivatives, and confidence intervals are computed as above. Note that σ^2 is also estimated.

Let us try these two methods on the body mass of the primate data set. We first fit a Brownian motion model with the default maximum likelihood method:

```
> ace(body, tree.primates)
$loglik
[1] -6.714469

$ace
[1] 1.183725 2.192018 2.571320 3.503182

$sigma2
[1] 1.9711502 0.6970463

$CI95
           [,1]     [,2]
[1,] -0.5058590 2.873308
[2,]  0.9868737 3.397163
[3,]  1.4844055 3.658235
[4,]  2.6858445 4.320519

$call
ace(x = body, phy = tree.primates)
```

The results are returned as a list with the ancestral estimates (`ace`) and their 95% confidence intervals in a matrix (`CI`); these values are indexed with the numbers of the node (see Section 3.1.1). With the default method, the function returns additionally the log-likelihood (`loglik`) and the estimated variance of the Brownian motion model with its standard error in a vector of length two (`sigma2`).

The option `CI`, whose default is `TRUE`, allows us to compute the 95% confidence intervals of the ancestral estimates. We now use the least squares method to fit the same model:

```
> ace(body, tree.primates, method = "pic")
$ace
      -1       -2       -3       -4
1.183725 2.780824 3.200378 3.852630

$CI95
          [,1]     [,2]
[1,] -1.296931 3.664381
[2,]  0.854866 4.706781
[3,]  1.367000 5.033757
[4,]  2.582428 5.122832

$call
ace(x = body, phy = tree.primates, method = "pic")
```

The least squares estimates are slightly larger than the maximum likelihood ones, particularly for the oldest nodes. Furthermore, the confidence intervals computed by maximum likelihood are usually narrower than those by least squares.

6.2.2 Discrete Characters

Markovian models provide a useful and practical tool for modeling the evolution of discrete characters [109]. We already have seen this framework with the substitution models of DNA sequences (Section 5.2.1). Because Markovian models have a probabilistic formulation, they can be fit by maximum likelihood and compared, for a given data set, with standard statistical methods. `ace` allows the user to set a variety of models in a flexible way.

Discrete characters are given as vectors or factors, and specify the option `type = "discrete"`. The option `model` is used to parameterize the transition rates among the states. The number of states is taken from the data (this can be seen with `unique(x)`).

The model is specified with a matrix of integers representing the indices of the parameters: 1 represents the first parameter, 2 the second one, and so on. The same number may appear several times in the matrix, meaning that the rates have the same values. For instance, with a two-state character, `model = matrix(c(0, 1, 1, 0), nrow = 2)` specifies that the transitions among both states occur at equal rates, and so there is only one parameter to be estimated from the data. This is best visualized by printing the matrix (the diagonal is always ignored here):

```
> matrix(c(0, 1, 1, 0), nrow = 2)
     [,1] [,2]
[1,]    0    1
[2,]    1    0
```

If instead we use the following matrix,

```
> matrix(c(0, 1, 2, 0), nrow = 2)
     [,1] [,2]
[1,]    0    2
[2,]    1    0
```

then different rates are assumed for both changes, and there are two parameters. We may recall that in the rate matrix, the rows represent the initial states and the columns the final states.

If there are three states, some possible models could have the following rate matrices.

```
> matrix(c(0, 1, 1, 1, 0, 1, 1, 1, 0), nrow = 3)
     [,1] [,2] [,3]
[1,]    0    1    1
[2,]    1    0    1
[3,]    1    1    0
> matrix(c(0, 1, 2, 1, 0, 3, 2, 3, 0), nrow = 3)
     [,1] [,2] [,3]
[1,]    0    1    2
[2,]    1    0    3
[3,]    2    3    0
> matrix(c(0, 1:3, 0, 4:6, 0), nrow = 3)
     [,1] [,2] [,3]
[1,]    0    3    5
[2,]    1    0    6
[3,]    2    4    0
```

To indicate that a transition is impossible, a zero must be given in the appropriate cell of the matrix. For instance, a “cyclical” change model could be specified by:

```
> matrix(c(0, 0, 3, 1, 0, 0, 0, 2, 0), nrow = 3)
     [,1] [,2] [,3]
[1,]    0    1    0
```

```
[2,]    0    0    2
[3,]    3    0    0
```

where, if the three states are denoted A, B, C, the permitted changes are the following: $A \rightarrow B \rightarrow C \rightarrow A$.

The number of possible models is very large, even with three states. The interest is to let the user define the models that may be sensible for a particular study and test whether they are appropriate.

There are short-cuts with character strings that can be used instead of a numeric matrix. The possible short-cuts are:

- `model = "ER"` for the equal-rates model,
- `model = "SYM"` for the symmetrical model,
- `model = "ARD"` for the all-rates-different model.

For a three-state character, these short-cuts result in exactly the same rate matrices shown above, respectively. By default, if the user sets `type = "discrete"`, then the default model is `"ER"`.

If the option `CI = TRUE` is used, then the likelihood of each ancestral state is returned for each node in a matrix called `lik.anc`. They are computed with a formula similar to (5.7), and scaled so that they sum to one for each node.

With the primate data, consider a character that sets *Galago* apart from the other genera (say "big eyes"). We first fit the default model (equal rates):

```
> x <- c(2, 2, 2, 2, 1)
> ace(x, tree.primates, type = "discrete")
$loglik
[1] -1.768921

$rates
[1] 0.3775508

$se
[1] 0.3058119

$lik.anc
             1         2
-1 0.304788504 0.6952115
-2 0.015697605 0.9843024
-3 0.019989199 0.9800108
-4 0.006221023 0.9937790

$call
ace(x = x, phy = tree.primates, type = "discrete")
```

The likelihood of the states "big eyes" and "small eyes" at the root are 0.3 and 0.7, respectively. Under this model, it is highly likely that the three other nodes of the tree were "small eyes".

We now fit the all-rates-different model:

```
> ace(x, tree.primates, type = "discrete", model = "ARD")
$loglik
[1] -1.602901

$rates
[1] 0.3059753 1.0892927

$se
[1] 0.3864119 1.2243605

$lik.anc
            1         2
-1 0.52689038 0.4731096
-2 0.11586096 0.8841390
-3 0.11724120 0.8827588
-4 0.04222992 0.9577701

$call
ace(x = x, phy = tree.primates, type = "discrete",
    model = "ARD")
```

Interestingly, the likelihoods on the root are quite affected by the model: the state of the root is now much less certain. For the other nodes, the likely state is still “small-eyes”. The increase in likelihood with the additional parameter is not significant:

```
> 1 - pchisq(2*(1.768921 - 1.602901), 1)
[1] 0.5644603
```

The genus *Homo* is sufficiently different from the other primate genera that it is not hard to find a discrete character that separates them. So we consider a character taking the value 1 in *Homo*,[5] and 2 in the four other genera. We fit the above two models and examine how their assumptions affect the likelihood of ancestral estimates.

```
> y <- c(1, 2, 2, 2, 2)
> ace(y, tree.primates, type = "discrete")
$loglik
[1] -2.772593

$rates
```

[5] This could be standing and moving upright, speaking complex languages, complex social structures, cooking food, writing poems, using computers to analyze phylogenies, and so on.

```
[1] 14.59506

$se
[1] 386.9471

$lik.anc
           1         2
-1 0.5000000 0.5000000
-2 0.5000000 0.5000000
-3 0.4999997 0.5000003
-4 0.5000000 0.5000000

$call
ace(x = y, phy = tree.primates, type = "discrete")

> ace(y, tree.primates, type = "discrete", model = "ARD")
$loglik
[1] -1.808865

$rates
[1]  8.030159 32.120695

$se
[1] NaN NaN

$lik.anc
           1         2
-1 0.5000000 0.5000000
-2 0.5000000 0.5000000
-3 0.5000000 0.5000000
-4 0.5002041 0.4997959

$call
ace(x = y, phy = tree.primates, type = "discrete",
    model = "ARD")
```

The distribution of `y` leads to much uncertainty in the ancestral likelihoods, a fact well-known to the users of the parsimony-based methods.

A more concrete application of `ace` with discrete characters is presented below with the *Sylvia* data.

6.3 Analysis of Diversification

The increasing availability of estimated phylogenies has led to a renewed interest in the study of macroevolution processes. For a long time, this issue was in the territory of paleontology. The fact that complete phylogenies become more numerous for more and more taxonomic groups has brought the biologists into the party.

The analysis of diversification is based on ultrametric trees with dated nodes. Most methods are based on a probabilistic model of speciation and extinction called the "birth–death" model [79]. This model assumes that there is an instantaneous speciation probability (denoted λ) and an instantaneous extinction probability (μ).

There are variations and extensions to this basic model. The most well known is when $\mu = 0$ (i.e., no extinction), which is called the Yule model. Nee et al. [104] suggested a generalization of the birth–death model where λ and μ vary through time. I suggested a model, called the Yule model with covariates, where λ varies with respect to species traits [115]. Because the birth–death model and its variants are probabilistic models, they can be fit to data by maximum likelihood. These models can be used both for parameter estimation and hypothesis testing. From a biological point of view, the main interest is the possibility of testing a variety of biological hypotheses depending on the fit models.

Other approaches consider a graphical or statistical analysis of the distribution of the branching times without assuming an explicit model. These methods focus on hypothesis testing.

6.3.1 Graphical Methods

Phylogenetic trees can be used to depict changes in the number of species through time. This idea has been explored by Nee et al. [105] and Harvey et al. [63]. The *lineages-through-time* plot is very simple in its principle: it plots the number of lineages observed on a tree with respect to time. With a phylogeny estimated from recent species, this number is obviously increasing because no extinction can be observed. If diversification has been constant through time, and the numbers of lineages are plotted on a logarithmic scale, then a straight line is expected. If diversification rates decreased through time, then the observed plot is expected to lay above the straight line, whereas the opposite result is expected if diversification rates increased through time.

The interpretation of lineages-through-time plots is actually not straightforward because in applications with real data the shape of the observed curve rarely conforms to one of the three scenarios sketched above [33]. This graphical method is of limited value to test hypotheses; particularly, its behavior is not known in the presence of heterogeneity in diversification parameters. However, it is an interesting exploratory tool given its very low computational cost.

There are three functions in `ape` for performing lineages-through-time plots: `ltt.plot`, `ltt.lines`, and `mltt.plot`. The first one does a simple plot taking a phylogeny as argument. By default, the x- and y-axes are labeled "Time" and "N", but this can be changed with the options `xlab` and `ylab`,

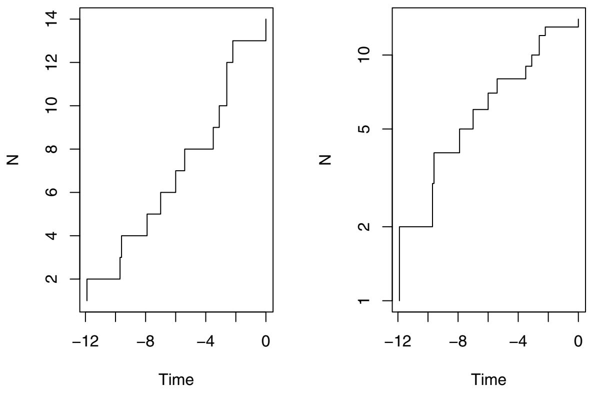


Fig. 6.5. Lineages-through-time plot of the clock tree of Michaux et al. [100] (Fig. 4.18) with a logarithmic scale on the right-hand side

respectively. This function has also a "dot-dot-dot" (...) argument (see p. 71 for an explanation of this argument) that can be used to format the plot (e.g., to alter the appearance of the line). As an illustration, let us come back to the rodent tree displayed in Fig. 4.18. It is ultrametric and so can be analyzed with the present method. We simply display the plot twice, with the default options, and set the y-axis on a logarithmic scale (Fig. 6.5):

```
layout(matrix(1:2, 1, 2))
ltt.plot(trk)
ltt.plot(trk, log = "y")
```

`ltt.lines` can be used to add a lineages-through-time plot on an existing graph (it is a low-level plotting command). It has only two arguments: an object of class `"phylo"` and the "dot-dot-dot" argument to specify the formatting of the new line (because by default, it is likely to look like the line already plotted). For instance, if we want to draw the lineages-through-time plots of both trees on Fig. 4.18, we could do:

```
ltt.plot(trk)
ltt.lines(trc, lty = 2)
```

`mltt.plot` is more sophisticated for plotting several lineages-through-time plots on the same graph. Its interface is:

```
mltt.plot(phy, ..., dcol = TRUE, dlty = FALSE,
          legend = TRUE, xlab = "Time", ylab = "N")
```

Note that the "dot-dot-dot" argument is not the last one; thus it does not have the same meaning as in the first two functions. Here, '`...`' means "a series of objects of class `"phylo"`". The options `dcol` and `dlty` specify whether the lines should be distinguished by their colors and / or their types (solid, dashed, dotted, etc.). To produce a graph without colors, one will need to invert the default values of these two options. The option `legend` indicates whether to draw a legend (the default). To compare the lineages-through-time plots of our two trees, we could do (Fig. 6.6):

```
mltt.plot(trk, trc, dcol = FALSE, dlty = TRUE)
```

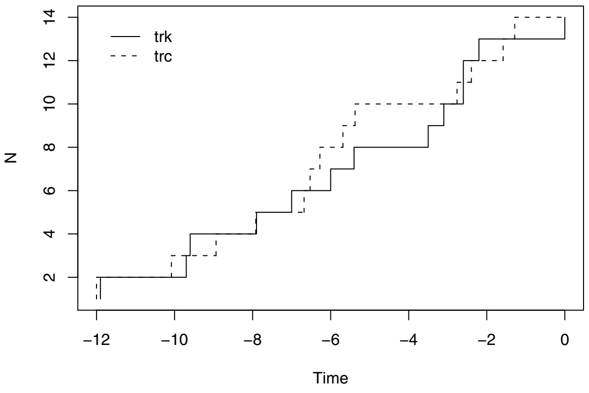


Fig. 6.6. Multiple lineages-through-time plot of the clock tree of Michaux et al. [100] and the tree estimated from the nonparametric rate smoothing method (Fig. 4.18)

Note that the axes are set to represent both lines correctly, which may not be the case when using `ltt.lines` (although the axes may be set with `xlim` and `ylim` passed to `ltt.plot` with the "dot-dot-dot"). The advantage of the latter is that the lines may be customized at will, whereas this is done automatically by `mltt.plot`.

6.3.2 Birth–Death Models

Birth–death processes provide a simple way to model diversification. There are reasons to believe that these models do not correctly depict macroevolutionary processes [93], but they are useful to use for data analysis because there has been considerable work to derive probability functions related to these processes making likelihood-based inference possible [79, 80].

The Simple Birth–Death Model

The estimation of speciation and extinction probabilities when all speciation and extinction events are observed through time is not problematic [78]. Some difficulties arise when only the recent species are observed. Nee et al. [104] derived maximum likelihood estimates of these parameters in this case. They used the following reparameterization: $r = \lambda - \mu$, $a = \mu/\lambda$. The estimates $\hat{\lambda}$ and $\hat{\mu}$ are then obtained by back-transformation. The function `birthdeath` implements this method: it takes as single argument an object of class `"phylo"`. Note that this tree must be dichotomous. If this is not the case, it could be transformed with `multi2di` (Section 3.4.3): this assumes that a series of speciation events occurred very rapidly. The results are returned as an object of class `"birthdeath"`. As an example, we come back to the 14-species rodent tree examined above with lineages-through-time plots:

```
> bd.trk <- birthdeath(trk)
> bd.trk

Estimation of Speciation and Extinction Rates
            With Birth-Death Models

     Phylogenetic tree: trk
        Number of tips: 14
              Deviance: 25.42547
        Log-likelihood: -12.71274
   Parameter estimates:
      d / b = 0   StdErr = 0
      b - d = 0.1438844   StdErr = 0.02939069
   (b: speciation rate, d: extinction rate)
   Profile likelihood 95% confidence intervals:
      d / b: [0, 0.5178809]
      b - d: [0.07706837, 0.2412832]
```

The standard errors of the parameter estimates are computed using the usual method based on the second derivatives of the likelihood function at its maximum. In addition, 95% confidence intervals of both parameters are computed using profile likelihood: they are particularly useful if the estimate of a is at the boundary of the parameter space (i.e., 0, which is often the case [117]).

`birthdeath` returns a list that allows us to extract the results if necessary. As an illustration of this, let us examine the question of how sensitive the above result could be to removing one species from the tree. The idea is simple: we drop one tip from the tree successively, and look at the estimated parameters (returned in the element `para` of the list). Instead of displaying the results directly we store them in a matrix called `res`. Each row of this matrix receives the result of one analysis:

```
> res <- matrix(NA, 14, 2)
> for (i in 1:14)
+   res[i, ] <- birthdeath(drop.tip(trk, i))$para
> res
      [,1]      [,2]
 [1,]    0 0.1354675
 [2,]    0 0.1354675
 [3,]    0 0.1361381
 [4,]    0 0.1369858
 [5,]    0 0.1376716
 [6,]    0 0.1439786
 [7,]    0 0.1410251
 [8,]    0 0.1410251
 [9,]    0 0.1421184
[10,]    0 0.1490515
[11,]    0 0.1318945
[12,]    0 0.1318945
[13,]    0 0.1361381
[14,]    0 0.1361381
```

This shows that the analysis is only slightly affected by the deletion of one species from the tree. With a larger tree, one could examine these results graphically, for instance, with a histogram (i.e., `hist(res[, 2])`).

Combining Phylogenetic and Taxonomic Data

It often occurs that a phylogeny is not complete in the sense that not all living species are included. This leads to some difficulties in the analysis of diversification because there are some obvious missing data. Pybus et al. [126] have approached this problem using simulations and randomization procedures. A more formal and general approach has been developed independently by Bokma [11] and myself [113]. The idea is to combine the information from phylogenetic data (branching times) and taxonomic data (species diversity). Formulae can be derived to calculate the probabilities of both kinds of observations, and because they depend on the same parameters (λ and μ) they can be combined into a single likelihood function.

The approach developed in [113] is implemented in the function `bd.ext`. Let us consider the phylogeny of bird orders (Fig. 4.13). The number of species in each order can be found in Sibley and Monroe [141]. These are entered by hand:

```
> data(bird.orders)
> S <- c(10, 47, 69, 214, 161, 17, 355, 51, 56, 10, 39, 152,
+        6, 143, 358, 103, 319, 23, 291, 313, 196, 1027, 5712)
> bd.ext(bird.orders, S)
```

```
Extended Version of the Birth-Death Models to
     Estimate Speciation and Extinction Rates

     Data: phylogenetic: bird.orders
                taxonomic: S
          Number of tips: 23
                Deviance: 289.1639
          Log-likelihood: -144.5820
    Parameter estimates:
       d / b = 0   StdErr = 0
       b - d = 0.2866789   StdErr = 0.007215592
    (b: speciation rate, d: extinction rate)
```

The output is fairly similar to the one from `birthdeath`. Note that it is possible to plot the log-likelihood function with respect to different values of a and r in order to derive profile likelihood confidence intervals of the parameter estimates (see [113] for examples).

The Yule Model with Covariates

The two applications of birth–death models above assume that speciation and extinction rates were constant through time. It is obvious that, biologically, this assumption must be relaxed because diversification has clearly fluctuated over time [142]. Nee et al. [104] suggested extending the simple birth–death model to include time-varying speciation and extinction rates, but this does not seem to have been implemented or further developed.

Another approach to this problem is to assume that these rates vary with respect to one or several species traits. This is appealing biologically because a major issue in biology is to identify the biological traits that lead to higher speciation and / or extinction rates [42, 73].

I proposed [115] to model speciation rates using a linear model written as

$$\ln \frac{\lambda_i}{1-\lambda_i} = \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \alpha \ , \tag{6.15}$$

where λ_i is the speciation rate for species i, x_{i1}, x_{i2}, ..., x_{ip} are variables measured on species i, and β_1, β_2, ..., β_p, α are the parameters of the model. The function $\ln(x/(1-x))$ is called the logit function (it is used in logistic regression and GLMs): it allows the term on the left-hand side to vary between $-\infty$ and $+\infty$. The terms on the right-hand side must be interpreted in the same way as a usual linear regression model. Let us rewrite (6.15) in matrix form as $\text{logit}(\lambda_i) = x_i^T\beta$. Giving some values of the vector β and of the traits x_i it is possible to predict the value of the speciation rate with the inverse logit function:

$$\lambda_i = \frac{1}{1 + e^{-x_i^T \beta}} \,. \tag{6.16}$$

If we make the assumption that there is no extinction ($\mu = 0$), then it is possible to derive a likelihood function to estimate the parameters of (6.15) giving an observed phylogeny and values of x [115]. Because this uses a regression approach, different models can be compared with likelihood ratio tests in the usual way.

A critical assumption of this model is that the extinction rate is equal to zero. This is clearly unrealistic but it appeared that including extinction rates in the model made it too complex to permit parameter estimation [115]. Some simulations showed that the test of the hypothesis $\beta = 0$ is affected by the presence of extinctions but it keeps some statistical power (it can detect an effect when it is present; see [115] for details).

The function `yule.cov` fits the Yule model with covariates. It takes as arguments a phylogenetic tree, and a one-sided formula giving the predictors of the linear model (e.g., `~ a + b`). The variables in the latter can be located in a data frame, in which case the option `data` must be used. They can be numeric vectors and / or factors: they are treated as continuous and discrete variables, respectively. The predictors must be provided for the tips and the nodes of the tree; for the latter they can be estimated with `ace` (Section 6.2). The results are simply displayed on the console. To fit the null model (i.e., with constant speciation rate), one can use the function `yule` which fits the simple Yule model. It returns an object of class `"yule"`. An application of these functions with the Felidae data is detailed below (Section 6.5.2).

6.3.3 Survival Models

The problem of missing species in phylogenies motivated some initial works on how to deal with this problem in the analysis of diversification. I suggested the use of continuous-time survival models for this purpose because they can handle missing data in the form of *censored* data [111].

Typical survival data are times to failure of individuals or objects [20]. It often occurs that some individuals are known to have been living until a certain time, but their exact failure times are unknown for various reasons (e.g., they left the study area, or the study ended before they failed or died). This is called *censorship*. The idea is to use this concept for missing species in phylogenies inasmuch as it is often possible to establish a minimum time of occurence for them [111].

Using survival models to analyze diversification implies that speciation and extinction rates cannot be estimated separately. The estimated survival (or hazard) rate must be interpreted as a diversification rate [111]. It is denoted δ ($= \lambda - \mu$). In theory a variety of models could be used, but only three are implemented in `ape` (see [111] for details):

- Model A assumes a constant diversification rate through time;

- Model B assumes that diversification changed through time according to a Weibull distribution with a parameter denoted β. If $\beta > 1$, then the diversification rate decreased through time; if $\beta < 1$, then the rate increased through time. If $\beta = 1$, then Model B reduces to Model A;
- Model C assumes that diversification changed with a breakpoint at time T_c.

These three models can be fit with the function `diversi.time`. This function takes as main arguments the values of the branching times (which can be computed beforehand, for instance, with `branching.times`). As a simple example, we take the data on *Ramphocelus* analyzed in [111]. This genus of passerine birds includes eight species: six of them were studied by Hackett [59] who resolved their phylogenetic relationships. For the two remaining species, some approximate dates of branching could be inferred from data reported in [59]. We enter the data by hand in R:

```
> x <- c(0.8, 1, 1.15, 1.55, 2.3, 0.8, 0.8)
> indicator <- c(rep(1, 5), rep(0, 2))
> diversi.time(x, indicator)

Analysis of Diversification with Survival Models

Data: x
Number of branching times: 7
         accurately known: 5
                 censored: 2

Model A: constant diversification
    log-likelihood = -7.594    AIC = 17.188
    delta = 0.595238    StdErr = 0.266199

Model B: diversification follows a Weibull law
    log-likelihood = -4.048    AIC = 12.096
    alpha = 0.631836    StdErr = 0.095854
    beta = 2.947881    StdErr = 0.927013

Model C: diversification changes with a breakpoint at time = 1
    log-likelihood = -7.321    AIC = 18.643
    delta1 = 0.15625    StdErr = 0.15625
    delta2 = 0.4    StdErr = 0.2

Likelihood ratio tests:
    Model A vs. Model B: chi^2 = 7.092    df = 1,    P = 0.0077
    Model A vs. Model C: chi^2 = 0.545    df = 1,    P = 0.4604
```

The results are simply printed on the screen. Note that here the branching times are scaled in million years ago (Ma), and thus the estimated parameters $\hat{\delta}$ (`delta`), $\hat{\alpha}$ (`alpha`), $\hat{\delta}_1$ (`delta1`, value of δ after T_c in model C), and $\hat{\delta}_2$ (`delta2`, value of δ before T_c in model C) must be interpreted with respect to this time scale. However, the estimate of $\hat{\beta}$ (`beta`) and the values of the LRTs are scale independent.

6.3.4 Goodness-of-Fit Tests

As pointed out earlier in this chapter, the estimation of extinction rates is difficult with phylogenies of recent species because extinctions are not observed [114]. However, it is clear that extinctions affect the distribution of branching times of a given tree [63, 103]. An alternative approach to parametric models is to focus on this distribution and compare it to a theoretical one with statistical goodness-of-fit tests based on the empirical cumulative distribution function (ECDF) [146, 149]. These tests compare the ECDF of branching times to the distribution predicted under a given model. The null hypothesis is that the observed distribution comes from this theoretical one. A difficulty of these tests is that their distribution depends on the null hypothesis, and thus the critical values must be determined on a case-by-case basis.

The function `diversi.gof` implements the goodness-of-fit tests as applied to testing a model of diversification [112]. It takes as main argument a vector of branching times in the same way as `diversi.time`. The second argument (`null`) specifies the distribution under the null hypothesis: by default `null = "exponential"` meaning that it tests whether the branching times follow an exponential distribution. The other possible choice is `null = "user"` in which case the user must supply a theoretical distribution for the branching times in a third argument (`z`).

As an application we consider the same data on *Ramphocelus* as in the previous section:

```
> diversi.gof(x)

Tests of Constant Diversification Rates

Data: x
Number of branching times: 7
Null model: exponential

Cramer-von Mises test: W2 = 0.841   P < 0.01
Anderson-Darling test: A2 = 4.81   P < 0.01
```

Two tests are computed: the Cramér–von Mises test which considers all data points equally, and the Anderson–Darling test which gives more emphasis in the tails of the distribution [147]. The critical values of both tests have been determined by Stephens [146]. If we want to consider only the five accurately known data points, the results are not changed:

```
> diversi.gof(x[indicator == 1])

Tests of Constant Diversification Rates

Data: x[indicator == 1]
Number of branching times: 5
Null model: exponential

Cramer-von Mises test: W2 = 0.578   P < 0.01
Anderson-Darling test: A2 = 3.433   P < 0.01
```

The results of these tests are scale independent.

Another goodness-of-fit test is the γ-statistic [125]. It is based on the internode intervals of a phylogeny: under the assumption that the clade diversified at constant rates, it follows a normal distribution with mean zero and standard deviation one. The γ-statistic can be calculated with the function `gammaStat` which takes as unique argument an object of class `"phylo"`. The null hypothesis can be tested with:

```
1 - 2*pnorm(abs(gammaStat(tr)))
```

6.3.5 Tree Shape and Indices of Diversification

The methods for analyzing diversification we have seen until now require knowledge of the branch lengths of the tree. Some researchers have investigated whether it is possible to get some information on diversification using only the topology of a phylogenetic tree (see [1, 3, 83] for reviews). Intuitively, we may expect unbalanced phylogenetic trees to result from differential diversification rates. On the other hand, different models of speciation predict different distributions of tree shapes.

apTreeshape implements statistical tests for two indices of tree shape: Sackin's and Colless's. Their formulae are:

$$I_S = \sum_{i=1}^{n} d_i \ , \tag{6.17}$$

$$I_C = \sum_{j=1}^{n-1} |L_j - R_j| \ , \tag{6.18}$$

where d_i is the number of branches between tip i and the root, and L_j and R_j are the number of tips descendant of the two subclades originating from node j. These indices have large values for unbalanced trees, and small values for fully balanced trees. They can be calculated for a given tree with the functions `sackin` and `colless`. Two other functions, `sackin.test` and `colless.test`, compute the indices and test, using a Monte Carlo method, the hypothesis that the tree was generated under a specified model. Both functions have the same options:

```
colless.test(tree, model = "yule", alternative = "less",
             n.mc = 500)
sackin.test(tree, model = "yule", alternative = "less",
            n.mc = 500)
```

where `tree` is an object of class `"treeshape"`, `model` gives the null model, `alternative` specifies whether to reject the null hypothesis for small (default) or large values (`alternative = "greater"`) of the index, and `n.mc` gives the number of simulated trees to generate the null distribution. The two possible null models are the Yule model (the default), and the PDA (`model = "pda"`). These models are described on p. 45.

A more powerful test of the above indices is the shape statistic which is the likelihood ratio under both Yule and PDA models. This statistic has distinct distributions under both models, so it is possible to define a most powerful test (i.e., one with optimal probabilities of rejecting either hypothesis when it is false). This is implemented in the function `likelihood.test`. We generate a random tree with the Yule model, and then try the function:

```
> trs <- rtreeshape(1, model = "yule")
> likelihood.test(trs)
Test of the Yule hypothesis:
statistic = -1.207237
p.value = 0.2273407
alternative hypothesis: the tree does not fit the Yule model

Note: the p.value was computed according to
      a normal approximation
> likelihood.test(trs, model = "pda")
Test of the PDA hypothesis:
statistic = -3.280261
p.value = 0.001037112
alternative hypothesis: the tree does not fit the PDA model

Note: the p.value was computed according to
      a normal approximation
```

Aldous [2, 3] introduced a graphical method where, for each node, the number of descendants of both subclades from this node are plotted one *versus* the other with the largest one on the x-axis. The expected distribution of these points is different under the Yule and PDA models. The function `aldous.test`

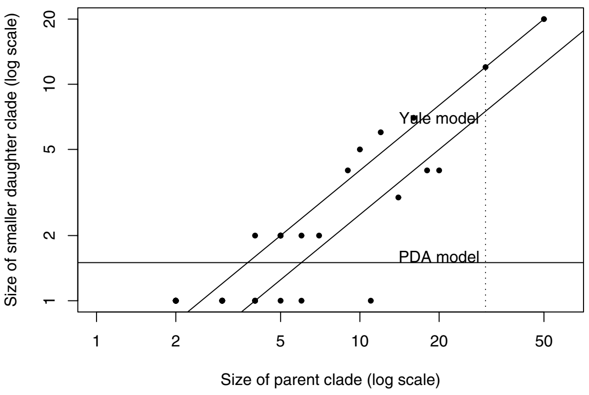


Fig. 6.7. Plot of the number of descendants of both subclades for each node of a tree with 50 tips simulated under a Yule model. The labeled lines are the expected distribution under these models, and the leftmost line is a quantile regression on the points

makes this graphical analysis. Together with the points, the expected lines are drawn under these two models. The option `xmin = 20` controls the scale of the x-axis: by default, the smallest clades are not represented which may be suitable for large trees. If we do the Aldous test with the small tree simulated above (Fig. 6.7):

```
aldous.test(trs, xmin = 1)
```

The expected lines are labeled with the null models just above them. The most leftward line (by default in red) is a quantile regression on the points.

6.4 Perspectives

There is certainly much to expect from the study of evolutionary processes using phylogenies of recent species. Phylogenetic data are accumulating at a rapid pace, and we can hope that more focus on macroevolutionary issues will lead to insights into the mechanisms of biological evolution. The methods already implemented in R cover a wide range of issues. It is likely that developments will continue in the same direction to offer biologists a complete environment for data analysis. Future developments could also include methods not yet available in R such as biogeographical models [90].

6.5 Case Studies

6.5.1 *Sylvia* Warblers

We begin by reading in the *Sylvia* data if necessary. We first drop the outgroup species (*Chamaea fasciata*) for which we have no ecological data:

```
load("sylvia.RData")
tr <- read.tree("sylvia_nj_k80.tre")
tr <- drop.tip(tr, "Chamaea_fasciata")
```

We also sort the data frame of ecological data so that its rows are in the same order as the tip labels of the tree:[6]

```
DF <- sylvia.eco[tr$tip.label, ]
```

We focus on an analysis of the geographical range by trying to reconstruct the evolution of this character. Migratory behavior is tightly linked with geographical range:

```
> table(DF$geo.range, DF$mig.behav)

           long resid short
  temp        0     4     0
  temptrop    9     0     4
  trop        0     7     0
```

We can assume in a first step that evolutionary changes among the three states occur at the same rate. We fit a model with `ace` using the option `type = "discrete"`—which may be abbreviated with `"d"`—and the default model (equal rates):

```
> syl.er <- ace(DF$geo.range, tr, type = "d")
> syl.er
$loglik
[1] -25.26805

$rates
[1] 136.5994

$se
[1] NaN
....
```

[6] Most functions in `ape` and `ade4` do not need this because the tip labels and the rownames are matched, but because here there are extra species in `sylvia.eco`, we do both operations at once.

The fact that no standard error has been computed for the rate parameter indicates that the likelihood surface of this model is flat, and the latter poorly fits the data. We fit the symmetrical model where transition rates differ from one state to another but transitions between two given states have equal rates in both directions. We use the short-cut `model = "SYM"`:

```
> syl.sym <- ace(DF$geo.range, tr, type = "d", model = "SYM")
$loglik
[1] -21.71442

$rates
[1]  28.20588 -18.23412  97.49406

$se
[1] 21.39655 22.74213 98.51708
....
```

This model clearly fits better: this is not surprising because we added two parameters. We can compute the likelihood ratio test comparing the two models to test whether the increase in fit is significant:[7]

```
> 1 - pchisq(2*(syl.sym$loglik - syl.er$loglik), 2)
[1] 0.0286206
```

This is significant, but we may want to try a more parsimonious "custom" model where only the transitions `temp` $\leftrightarrow$ `temptrop` $\leftrightarrow$ `trop` are permitted. We define a symmetric matrix `mod` that is used as a model in `ace`:

```
> mod <- matrix(0, 3, 3)
> mod[2, 1] <- mod[1, 2] <- 1
> mod[2, 3] <- mod[3, 2] <- 2
> mod
     [,1] [,2] [,3]
[1,]    0    1    0
[2,]    1    0    2
[3,]    0    2    0
```

The rate matrix `mod` has two parameters: the first one for the transitions `temp` $\leftrightarrow$ `temptrop`, and the second one for the transitions `temptrop` $\leftrightarrow$ `trop`.

```
> syl.mod <- ace(DF$geo.range, tr, type = "d", model = mod)
> syl.mod
$loglik
```

[7] A likelihood ratio test is computed as twice the difference in log-likelihoods, and follows a χ^2 distribution with the number of degrees of freedom given by the difference in number of parameters. The function `pchisq` gives the cumulative density function of the χ^2 distribution (i.e., $Pr(x \leq \chi^2)$).

```
[1] -24.29444

$rates
[1] 32.79765 98.11600

$se
[1] NaN NaN
....
```

This model does not fit better, so we stick to the symmetrical model.

How do we interpret the rates estimated by `ace`? We use the methodology described for substitution models to calculate a probability matrix from the rate matrix (Section 5.2.1). We first build the latter with the estimated rates that are arranged columnwise in the matrix:

```
> Q <- matrix(0, 3, 3)
> Q[1, 2] <- Q[2, 1] <- syl.sym$rates[1]
> Q[1, 3] <- Q[3, 1] <- syl.sym$rates[2]
> Q[2, 3] <- Q[3, 2] <- syl.sym$rates[3]
> Q
          [,1]     [,2]      [,3]
[1,]   0.00000 28.20588 -18.23411
[2,]  28.20588  0.00000  97.49406
[3,] -18.23411 97.49406   0.00000
```

We set the diagonal of the matrix so that the rows sum to zero (the command below will work if this diagonal is initially filled with zeros):

```
> diag(Q) <- -rowSums(Q)
> Q
           [,1]       [,2]      [,3]
[1,]  -9.971765   28.20588 -18.23411
[2,]  28.205880 -125.69994  97.49406
[3,] -18.234115   97.49406 -79.25994
```

The rate matrix is now ready and we can compute the probabilities for a given time. The latter must be relevant with respect to the estimated parameters (i.e., on the same scale as the original branch lengths); here we take $t = 0.05$:

```
> library(rmutil)
> P <- mexp(0.05 * Q)
> rownames(P) <- c("temp", "temptrop", "trop")
> colnames(P) <- c("temp", "temptrop", "trop")
> round(P, 3)
          temp temptrop  trop
temp     0.792    0.187 0.022
temptrop 0.187    0.380 0.433
trop     0.022    0.433 0.545
```

These probabilities suggest that temperate-tropical is the most “unstable” state, and that most transitions occur between this state and the tropical one. Temperate species seem to evolve only from temperate-tropical ones.

We now plot the likelihoods of the ancestral characters on the tree together with the values observed for the species. We first create a vector of mode character to store the colors used for the symbols on the tips: black for temperate, white for tropical, and grey for temperate-tropical.

```
co <- rep("grey", 24)
co[DF$geo.range == "temp"] <- "black"
co[DF$geo.range == "trop"] <- "white"
```

We plot the tree as a cladogram to better display the information; the option `label.offset` is used to leave some space for the symbols. The latter are drawn with `tiplabels`: the symbols are colored with the vector `co` prepared above, and `adj = 1` avoids the symbols overlapping with the tips of the tree. Finally, the likelihoods of the ancestral characters are added with `nodelabels` using the option `thermo` (Fig. 6.8):

```
plot(tr, "c", FALSE, no.margin = TRUE, label.offset = 1)
tiplabels(pch = 22, bg = co, cex = 2, adj = 1)
nodelabels(thermo = syl.sym$lik.anc,
           bg = c("black", "grey", "white"), cex = 0.8)
```

From this analysis we can infer that the ancestor of the genus *Sylvia* was, probably, a tropical bird. Because all tropical *Sylvia* are also resident, this genus probably evolved from a tropical resident species.

6.5.2 Phylogeny of the Felidae

We continue the analysis of the Felidae phylogeny by first reading back the tree in R:

```
tr <- read.tree("felid.chrono.tre")
```

We are interested here in the diversification parameters of this group. We first estimate the global speciation rate of this phylogeny by fitting a Yule model:

```
> yule(tr)
$lambda
[1] 0.2318725

$se
[1] 0.04036382

$loglik
[1] 7.349097
```

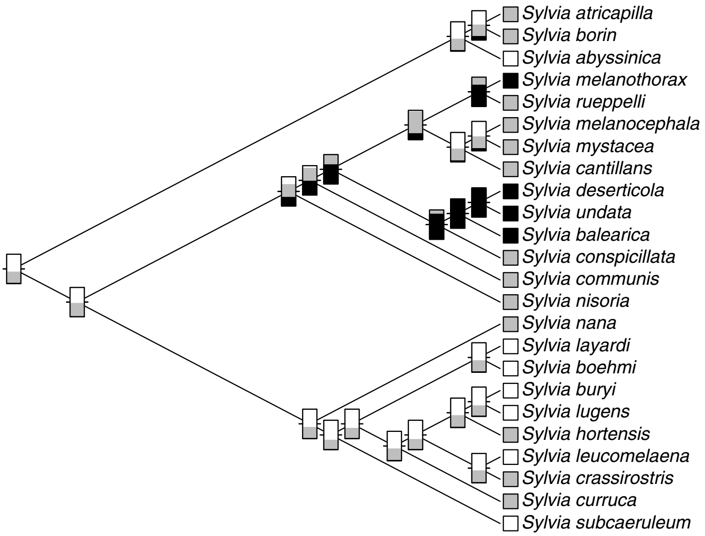


Fig. 6.8. Ancestral estimates of geographical range for 24 species of *Sylvia*. The thermometers on the nodes show the relative likelihoods of the three states: temperate (black), temperate-tropical (grey), tropical (white). The state of the recent species are shown on the tips of the tree

```
attr(,"class")
[1] "yule"
```

The estimated speciation probability is quite high ($\hat{\lambda} = 0.23 \pm 0.08$). We now try to fit the simple birth–death model:

```
> birthdeath(tr)

Estimation of Speciation and Extinction Rates
            with Birth-Death Models

     Phylogenetic tree: tr
        Number of tips: 35
              Deviance: -16.81068
        Log-likelihood: 8.405339
   Parameter estimates:
      d / b = 0.6280969   StdErr = 0.2103922
      b - d = 0.1340512   StdErr = 0.05509144
   (b: speciation rate, d: extinction rate)
   Profile likelihood 95% confidence intervals:
      d / b: [0.3227608, 0.7970532]
      b - d: [0.084925, 0.2029478]
```

This is an interesting result because in most applications of the birth–death model without fossils the estimated extinction probability is usually zero, even when there are speciations [114]. The estimated parameters are $\hat{a} = 0.63$ and $\hat{r} = 0.13$. By back-substitution using $\lambda = r/(1 - a)$ and $\mu = \lambda a$, we obtain $\hat{\lambda} = 0.36$ and $\hat{\mu} = 0.23$. We can compare the Yule model with the birth–death model with a likelihood ratio test because the latter has one additional parameter (μ):

```
> 1 - pchisq(2*(8.405339 - 7.349097), 1)
[1] 0.146102
```

This is not significant at the 0.05 level leading us to accept the null hypothesis that $\mu = 0$, but we need to be very cautious about this result because the estimation of extinction rates is particularly difficult with phylogenies of recent species [114].

We now explore the possible impact of body mass on speciation rate. We first load the previously saved workspace with the data on body mass:

```
load("felid.RData")
```

We check that each species in our tree has data on body mass:

```
> IN <- tr$tip.label %in% names(felid.body.mass)
> tr$tip.label[!IN]
[1] "Prionailurus_rubiginosa" "Felis_catus"
[3] "Felis_libyca"
```

This is not the case as three species appear to have no data on body mass. An examination of the latter data shows that a mismatch is due to a different termination of the species name of the rusty-spotted cat:

```
> names(felid.body.mass)[36]
[1] "Prionailurus_rubiginosus"
```

Thus we simply change the name of this species, and give a body mass of 3500 g to both species of cats:

```
names(felid.body.mass)[36] <- "Prionailurus_rubiginosa"
x <- rep(3500, 2)
names(x) <-  c("Felis_catus", "Felis_libyca")
felid.body.mass <- c(felid.body.mass, x)
```

As a final check before proceeding, we verify that all species in the tree have a body mass in our data:

```
> all(tr$tip.label %in% names(felid.body.mass))
[1] TRUE
```

We can now assess the effect of body mass on speciation rate of Felidae. We must first estimate the ancestral values of this variable using `ace`. The function `yule.cov` is sensitive to the distribution of the predictors: if they are too skewed the fitting procedure is likely to fail [115]. Consequently, we log-transform body mass and center the variable:

```
> range(felid.body.mass)
[1]   1300 433200
> X <- scale(log(felid.body.mass[tr$tip.label]), scale=FALSE)
> range(X)
[1] -1.930336  2.898372
```

The option `scale = FALSE` prevents data scaling (only centering is done). We also have sorted the data in the same order as in the tree (which is required by `yule.cov`). We then estimate the ancestral body mass with `ace` using the default maximum likelihood method:

```
X.node <- ace(X, tr)$ace
```

These values must be sorted according the node numbers of the tree, which is done by `ace`. We can now feed the data to `yule.cov`:

```
> yule.cov(tr, ~ c(X, X.node))

---- Yule Model with Covariates ----

    Phylogenetic tree: tr
       Number of tips: 35
      Number of nodes: 34
             Deviance: -15.25978
       Log-likelihood: 7.629888

  Parameter estimates:
               Estimate    StdErr
(Intercept)  -1.1870642 0.1612194
c(X, X.node) -0.1615685 0.1539165
```

The increase in log-likelihood is very small compared to the Yule model so it is not necessary to compute the P-value. In spite of this, we find a slight negative effect of body mass on speciation rate meaning that the smaller species tend to speciate more rapidly. An easier way to interpret this result is to use the inverse logit-transformation (6.16), and plot the calculated values of λ with respect to the predictor. In the present case, the predictor varies between -1.93 and 2.90, so we create a sequence between -2 and 3 (with a reasonable increment to smooth the plot) to cover the observed variation:

```
> x <- seq(-2, 3, 0.05)
```

We compute the corresponding predicted value of λ:

```
lambda <- 1 / (1 + exp(-(-0.1615685 * x + -1.1870642)))
```

We could simply make the plot with `plot(x, lambda)`, but we can make it more informative by transforming the scale of the x-axis so that it is similar to the scale of the original body mass data: this implies adding the mean of the log-transformed body mass (the inverse of centering), and then taking the exponential (the inverse of the logarithmic transformation). We do the plot with `type = "l"` to draw a curve, and we use `rug` to plot on the x-axis the observed values of body mass (Fig. 6.9):

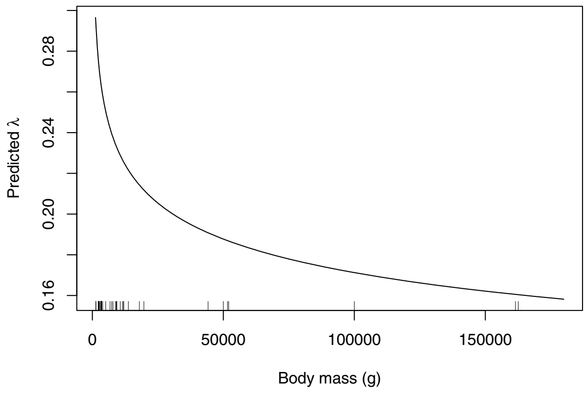


Fig. 6.9. Predicted variation in speciation rate (λ) with respect to body mass for the Felidae

```
ox <- exp(x + mean(log(felid.body.mass[tr$tip.label])))
plot(ox, lambda, type = "l", xlab = "Body mass (g)",
     ylab = expression("Predicted "*lambda))
rug(felid.body.mass[tr$tip.label])
```

The function `expression` allows us to write special characters on a plot. We should keep in mind that the depicted relationship is not statistically significant (see [115] for an example of significant effects with primates).

6.6 Exercises

1. Simulate for 99 time-steps two independent Brownian motion models with the same initial values. These variables should be taken as two species

that have diverged after $t = 1$, and they should be stored in a two-column matrix.

(a) Simulate the divergence of each species in two daughter-species at $t = 100$ under the same model for 100 time-steps: the results should be stored in a four-column matrix. Plot the whole evolution for the 200 time-steps on a single graph.
(b) Repeat (a) but using an Ornstein–Uhlenbeck model with $\alpha = 0.2$, $\theta_1 = -1$ for the first pair of species, and $\theta_2 = 1$ for the second one.
(c) Repeat (b) with $\theta_1 = -20$ and $\theta_2 = 20$. Compare the results.

2. Calculate the expected values of the Brownian motion and the Ornstein–Uhlenbeck models after 100 time-steps. Compare with the observed values from the simulations above.

3. Implement Desdevises et al.'s [24] method in R (see p. 144).

4. Consider the phylogeny estimated for the Felidae (Section 5.5.2). Compute the phylogenetically independent contrasts for body mass using the following branch lengths:
   - The maximum likelihood estimates from PHYML (Fig. 5.5);
   - From the chronogram estimated by NPRS (Fig. 5.6);
   - Setting the node heights so that they are equal to the number of descendants (see `compute.brlen`);
   - All equal to one.

   Compare the results and comment on the assumptions underlying the use of each set of branch lengths.

5. Consider the neighbor-joining tree estimated for the genus *Sylvia* and the associated bootstrap values.
   (a) Compute the phylogenetically independent contrasts for the continuous variable (migratory distance, `mig.dist`) in the ecological data set.
   (b) We want to give more importance in the analysis to the contrasts associated with the nodes that are well supported by the bootstrap analysis. Propose a solution.
   (c) Compare the two sets of contrasts.

6. Analyze the diversification pattern from the phylogeny estimated in Exercise 5 of Chapter 5.

7

Developing and Implementing Phylogenetic Methods in R

We have seen several times in this book that it is not necessary to know R in depth to use it for data analysis, even to tackle complex analyses. On the other hand, we need to know more of the language and R's features to develop and implement methods with it.

The materials in this chapter are not a formal introduction to R, but highlight some useful points in the present context. The primary references are the manuals distributed with R (located in the directory R_HOME/doc/manual/) and available on CRAN.[1] This chapter essentially uses materials from *Writing R Extensions* [129] and the *R Language Definition* [128].

7.1 Features of R

R is a language that is qualified as a dialect of S, a language for statistics [8]. The syntax of both languages is essentially identical, but their implementations differ. This implies that programs written in S will not necessarily run under R, but compatibility is very large. For a brief comparison of R and S, one can see the R-FAQ available both on CRAN,[2] and distributed with R (R_HOME/FAQ).

R is an interpreted language: all commands are read by a parser, then interpreted, and, if syntactically correct, executed. There are different ways to enter commands in R: they can be typed directly at R's prompt (in a console or a terminal), or read from a file with the function `source`.

7.1.1 Object-Orientation

R is an object-oriented language. Object-orientation is often seen as a complex mechanism in computer programming (e.g., C++ is often cited as being more

[1] http://cran.r-project.org/manuals.html.

[2] http://cran.r-project.org/faqs.html.

complex than C). In R, however, this feature is not as complex as in Java or in C++, and considerably simplifies things.

We have seen the use of generic functions several times in the previous chapters. Let us now see some details. A generic function is named after its main use: `print`, `summary`, `plot`, and so on. All these functions have similar content, for instance:[3]

```
> print
function (x, ...)
UseMethod("print")
<environment: namespace:base>
```

Consider an object `x` of class `"cls"`, then `print(x)` is equivalent to `print.cls(x)`. The function `print.cls` (as well as any function `print.*`) is called a *method*. If the method of a particular class does not exist, then the generic uses the default method (for instance, if `print.cls` does not exist, `print(x)` uses `print.default(x)`).

A nice example of the use of generics/methods is when plotting an object. Suppose `x` is a numeric vector (say, 1, 2, 3, ...), then the command `plot(x)` will do a simple plot of the values of `x`. But if `x` is a phylogenetic tree (e.g., an object of class `"phylo"`), we do not want this! Because the function `plot.phylo` is defined in the package `ape`, `plot(x)` will correctly plot the tree (Chapter 4).

A method is written in exactly the same way as another function: only its name must follow the rule *`generic.class`* where *`generic`* is the name of the generic, and *`class`* is the name of the class. A method must have, at least, all the arguments of the generic, with the same names and in the same order. If the generic function has a "dot-dot-dot" argument (which is often the case), this is almost always the last one. For instance, consider the function `all.equal` that compares two objects taking some approximations into account. The generic is:

```
> all.equal
function (target, current, ...)
UseMethod("all.equal")
<environment: namespace:base>
```

The method that does this comparison for two objects of class `"phylo"` is, of course, called `all.equal.phylo`, and its first few lines are:

```
> all.equal.phylo
function (target, current, ...)
{
```

[3] It may be useful to recall that typing the name of an object results in printing its content; thus typing the name of a function, without the parentheses, prints its content.

```
### commands to compare two objects of class "phylo"
...
```

A method is used practically as its generic is, but it is possible to force the use of a particular method. For instance, because an object of class "phylo" is a list, it is possible to compare two of these objects with `all.equal.list(tr1, tr2)` (which is done internally by `all.equal.phylo`).

7.1.2 Variable Definition and Scope

In R, it is not necessary to declare the variables and objects used within a function (in contrast to languages such as C or FORTRAN). For instance, an expression like `x <- 1` creates the vector `x` and sets its attributes accordingly; if `x` already exists then it is erased beforehand. On the other hand, for an expression like `y <- x`, `x` must already exist.

When writing a computer program (whatever the language), it is often necessary to decide whether a variable is local (used only within a function) or global (can be used by several functions in the program). In R, because the declaration of variables is implicit, a rule is needed. This rule is called *lexical scoping*. To understand this mechanism, let us consider the very simple function:

```
> foo <- function() print(x)
> x <- 1
> foo()
[1] 1
```

Because no variable named `x` has been created within `foo`, R will seek in the *enclosing* environment if there is an object called `x`, and will print its value (otherwise, a message error is displayed, and the execution is stopped).

If an object `x` is created within our function, the value of `x` in the global environment is not changed.

```
> x <- 1
> foo2 <- function() {
+    x <- 2
+    print(x)
+ }
> foo2()
[1] 2
> print(x)
[1] 1
```

Now `print(x)` uses the object `x` that is defined within its environment, that is, the environment of `foo2`.

The word *enclosing* above is important. In our two example functions, there are two environments: the global one and the one of the function `foo`

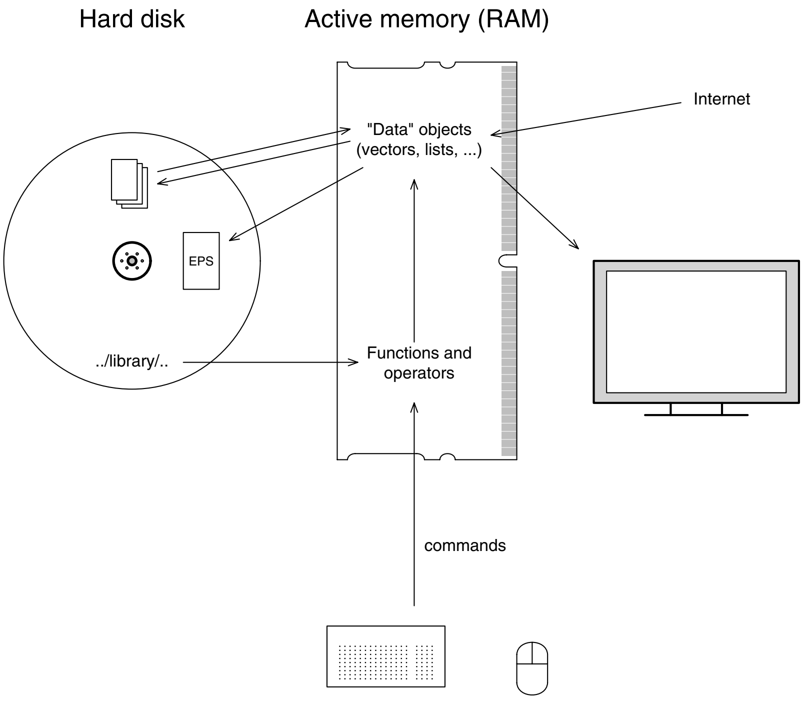


Fig. 7.1. A schematic view of how R works

or `foo2`. If there are three or more nested environments, the search for the objects is made progressively from a given environment to the enclosing one, and so on, up to the global one.

7.1.3 How R Works

All the actions of R are done on objects stored in the active memory of the computer: no temporary files are used (Fig. 7.1). Files on the disk are read and written for input and output of data and results (graphics, etc.) The user executes the functions via some commands. The results are displayed directly on the screen, stored in an object, or written on the disk (particularly for graphics). Because the results are themselves objects, they can be considered as data and analyzed as such. Data files can be read on the local disk or on a remote server through the Internet.

The functions available to the user are stored in a directory called R_HOME/library (R_HOME is the directory where R is installed). This directory contains *packages* of functions, which are themselves structured in directories. The package named base is in a way the core of R and contains the basic functions of the language for reading, manipulating, and writing data.

7.2 Writing Functions in R

Writing functions can be somehow extrapolated from what has been said in the previous sections. Quite logically, a function is defined with the function function which takes as arguments the variable(s) that will be used locally within the function when it is called. R functions are objects, and the result of the function function can be assigned in the same way as other objects (the examples below are purely didactical):

```
> f <- function(x) print(mode(x))
> f
function(x) print(mode(x))
> f(1)
[1] "numeric"
> f(TRUE)
[1] "logical"
> f("a")
[1] "character"
```

In this example, the object x is local to the variable and if an object called x exists in the workspace, it will not be used:

```
> x <- FALSE
> print(mode(x))
[1] "logical"
> f(x = 1)
[1] "numeric"
```

Note that we used the tagged argument in the last call to emphasize this point.

Default arguments (often called options) are set by preassigning them in the function definition:

```
> fb <- function(x, prefix = "Mode:")
+   print(paste(prefix, mode(x)))
> fb(1)
[1] "Mode: numeric"
> fb(1, "")
[1] " numeric"
> fb(1, "The mode is")
[1] "The mode is numeric"
```

Quite often, default arguments are logicals to control what is computed by the function. For instance, if we want a function that calculates the mean of a sample with the possibility of removing all negative values, we can control this with a logical argument whose default value will be `FALSE`:

```
> foo <- function(x, rm.negative = FALSE)
+   if (rm.negative) print(mean(x[x >= 0]))
+   else print(mean(x))
> y <- rnorm(100)
> foo(y)
[1] 0.04609175
> foo(y, TRUE)
[1] 0.751289
```

To be executed, a function must be loaded in memory, and this can be done in several ways. The commands of a function can be typed directly on the keyboard, as with any other command, or copied and pasted from an editor. If the function has been written in a text file, it can be loaded with `source` like another program; a single file can contain several functions. Similarly, functions can be saved in an '.RData' file, as with any R objects, and loaded in memory with `load`. Finally, it is possible to create a package: this is discussed in Section 7.4.

To load some functions, packages, or data in memory when R is started, the best option is to configure the file '.Rprofile'. This file, if it exists, is read by R at start-up: it must be located in the HOME directory of the user. This file is user dependent, so that if a computer is shared by several users, they may have different '`.Rprofile`' files. The path to the HOME directory can be printed in R with the command:

```
> Sys.getenv("HOME")
         HOME
"/home/paradis"
```

This directory should not be confused with the R_HOME directory which is the place where R is installed, and is unique to a computer. Here is an example on a Linux system:

```
> Sys.getenv("R_HOME")
    R_HOME
"/usr/lib/R"
```

The contents of '`.Rprofile`' are normal R commands, and comments can be included as well. This is normally the place where you will customize R by modifying the options. The list and meanings of these options is explained in `?options`. Here is an example:

```
options(width = 60) # narrower output on the screen
options(editor = "emacs") # the default on Linux is vi...
options(show.signif.stars = FALSE) # avoid the Milky Way
library(ade4)
library(ape)
library(seqinr)
load("/home/paradis/data/always_load_this.RData")
source("/home/paradis/data/always_source_this.R")
```

7.3 Interfacing R with Other Languages

Phylogenetic methods are often computationally intensive, and thus phylogenetic programs are mostly written in low-level languages (mainly C or C++). These programs need to be compiled (in contrast to programs in interpreted languages such as R) to be used. However, and this is completely transparent to the user, R uses compiled programs too: most computational tasks in R are made by compiled C or FORTRAN programs.

R has several mechanisms to interface compiled programs with its interpreter (the CLI we have seen through this book). At least three benefits can be found in using these interfaces when implementing a phylogenetic method in R.

- The performance of an R program can be greatly improved when the computationally demanding part is done with compiled codes (see an example below);
- The R application programmer interface (API) can be used making available many C functions useful in computational statistics (mathematical, matrix calculus, probability distribution, optimization functions, and so on);
- Existing programs in C or C++ can be ported to R.

The cost is that one has to learn these interfaces, but this is relatively easy, and outlined in this section.

7.3.1 Simple Interfaces

The R function `.C` gives the way to call a C function from R using a simple interface that matches the arguments in C. The latter must be pointers. An example could be:

```
void fcn(int * arg1, double * arg2, char ** arg3)
{
...
}
```

The code in this function can be any C code, and can call other functions. fcn can be called from R with:

```
.C("fcn", as.integer(i), as.double(x), as.character(b),
   PACKAGE = "pkg")
```

It is necessary that the data types to be checked before passing the variables to the C code: this explains the distinction between integers and doubles here. R does not distinguish these two data types, so there is a single numeric mode (Section 2.2.1). On the other hand, C has different data types for integers and reals, hence the conversion when passing data from R to C. "pkg" is the name of the R package where fcn can be found.

To be able to use fcn from R, this C function must be compiled and loaded into R. The compilation is done so as to produce a library file ('*.dll' under Windows, or '*.so' for the other operating systems). The library is loaded with the function library.dynam. Usually, it is easier to build a small package where the needed codes are included (Section 7.4).

In practice, .C is not called directly by the user but it is included in an R function, for example,

```
fcn <- function(i, x)
{
.C("fcn", as.integer(i), as.double(x), as.character(b),
   PACKAGE = "pkg")
}
```

so that the user does not see whether the function calls a compiled code:

```
fcn(i, x)
```

Programs written in C++ are called in a way similar to C from R, but in the C++ code a wrapper must be written:

```
// X_main.cc:
#include ...
extern "C" {
  void X_main () {
    ...
  }
} // extern "C"
```

Such a program must be compiled with a C++ compiler.

7.3.2 Complex Interfaces

We have seen that with .C, only simple data types can be passed to the C code. This may be problematic if one wants to manipulate R objects that have a complex structure, such as lists, and for which the number of elements is not known a priori. In this situation, the function `.Call` can be used. Its use, from the R side, is simpler than `.C`:

```
.Call("fcn", a, b)
```

There is no data type checking here: this is done in the C program. The structure of the latter is more complex, and makes use of the data type SEXP (*S expression*):

```
SEXP fcn(SEXP a, SEXP b)
{
...
}
```

All the details on how to handle SEXP data in C are explained in [129].

There is an even more complex mechanism with the function `.External` which can be used with an a priori unknown number of arguments. It is used in a similar way in R:

```
.External("fcn", a, b)
```

But in C there is only one argument:

```
SEXP fcn(SEXP args)
{
...
}
```

The elements passed with `args` may be extracted sequentially with special functions:

```
...
first = CADR(args);
second = CADDR(args);
third = CADDDR(args);
fourth = CAD4R(args);
...
```

The sources of `ape` and `ade4` provide some examples of the use of `.C` and `.Call` with phylogenetic data, and those of `seqinr` of the use of `.Call` with sequence data.

7.4 Writing R Packages

All the details of writing an R package are explained in a clear way in [129]. We show here only how we can make a minimal package that could be used to port some C codes to R.

A nice way to write an R package is to compile and install R and C codes so that it can be tested. If this is sucessful and the developer wants to publish the package, then the next stage is to write the documentation.

7.4.1 A Minimalist Package

A package may contain only R codes which is straightforward to make and install. We consider cases where some codes need to be compiled. Suppose we have written the R and C functions, and they are collected in files called accordingly ('*.R' and '*.c'). Then we need to create two other files: 'DESCRIPTION' and 'zzz.R'. The files must be arranged in the following directories.

/pkg/DESCRIPTION
/pkg/R/*.R
/pkg/src/*.C

The file 'DESCRIPTION' contains some general information on the package. It must contain at least the following fields.

```
Package: pkg
Version: 0.1
Date: 2005-12-25
Title: PKG
Author: John Marillion <john@marillion.net>
Maintainer: John Marillion <john@marillion.net>
Description: This is a minimalist install for pkg.
License: GPL version 2 or newer
```

This file must eventually be more detailed if there are dependencies with other packages or libraries. The file 'zzz.R' is necessary if there are compiled codes. Its content is:

```
.First.lib <- function(lib, pkg) {
    library.dynam("pkg", pkg, lib)
}
```

where `"pkg"` should be replaced by the quoted name of the package, but `pkg` should be left unchanged; for instance, for `ape` this is `library.dynam("ape", pkg, lib)`. The function `.First.lib` is executed when the package is loaded with `library(pkg)`.

Once the files and directories have been prepared, `pkg` can be installed with the command (from a shell):

R CMD INSTALL pkg

The package may then be used in R.

7.4.2 The Documentation System

Every function written in R when distributed in a package must be documented. This is not necessary for the installation.

There is a single documentation format called `Rd` that is processed during the installation to create help pages in simple text (read with `?`), HTML, and PDF.

Once the help pages have been prepared and put in a directory /pkg/man, it is possible to check the package with:

R CMD check pkg

7.5 Performance Issues and Strategies

From all we have seen in this book, it appears that we often have a choice among several possibilities for the same task. This is common in computer programming where different algorithms can be used to do the same operation. Here, we also have a choice among different computer languages that can be interfaced among each other.

Roughly, there are three strategies when implementing a method in R: use only R codes, interface C and / or C++ codes with R using the simple interface function `.C`, and doing the same but with the complex interface functions `.Call` and / or `.External`. These three strategies are detailed in Table 7.1 with their gains and costs.

Although more costs are listed for the "R + C" strategies, this actually reveals a contrast simplicity *versus* performance. Interfacing C programs with R will almost always result in a significant increase in performance at the cost of more complex programming.

To give an idea of the gain in performance that could result from transferring a computation done in R to C, we can consider a concrete example from `ape`. When plotting a tree, the function `plot.phylo` computes the coordinates of the nodes and tips in the graph, and then draws the appropriate lines. Originally, all computations were done only in R code. One of these functions returned the distance from the root to each node and tip using edge lengths:

```
node.depth.edgelength <- function(x, el)
### Input: the matrix ‘edge’ of an object of class
### "phylo", and the corresponding vector ‘edge.length’.
{
    tmp <- as.numeric(x)
```

Table 7.1. Comparative gains and costs of different strategies when implementing a computational method in R

	Gains	Costs
Pure R	Easily programmed. Programs can be tested directly. Programs can be shared directly among operating systems. Performance can be very good. Bugs are easily fixed.	Performance can be poor if vectorization cannot be achieved.
`.C`	C and C++ programs can be ported to R. C functions already programmed in R can be used. Performance is generally greatly improved.	Programs need to be compiled to be tested. Compilation is system dependent. Bugs are more difficult to find than in R. Only simple R data types (vectors) can be passed to C.
`.Call`	Same as `.C`. Complex R objects (e.g., lists) can be passed to C.	Same than `.C` but the last point. Need to learn the R macros to manipulate R objects in C.
`.External`	Same as `.Call`. The number of objects passed to C may vary.	Same as `.Call`.

```
    nb.tip <- max(tmp)
    nb.node <- -min(tmp)
    xx <- as.numeric(rep(NA, nb.tip + nb.node))
    names(xx) <- as.character(c(-(1:nb.node), 1:nb.tip))
    xx["-1"] <- 0
    for (i in 2:length(xx)) {
        nod <- names(xx[i])
        ind <- which(x[, 2] == nod)
        base <- x[ind, 1]
        xx[i] <- xx[base] + el[ind]
    }
    xx
}
```

From version 1.4 of ape, this function has been replaced by a small C program called from R:

```
void node_depth_edgelength(int *ntip, int *nnode, int *edge1,
       int *edge2, int *nms, double *edge_length, double *xx)
{
  int i, j, k;
```

```
    for (i = 1; i < *ntip + *nnode; i++) {
      j = 0;
      while (edge2[j] != nms[i]) j++;
      if (edge1[j] < 0) k = -edge1[j] - 1;
      else k = nnode + edge1[j] - 1;
      xx[i] = xx[k] + edge_length[j];
    }
  }
```

which is called from R with:

```
.C("node_depth_edgelength", as.integer(nb.tip),
   as.integer(nb.node), as.integer(x$edge[, 1]),
   as.integer(x$edge[, 2]), as.integer(nms),
   as.double(x$edge.length),
   as.double(numeric(nb.tip + nb.node)),
   DUP = FALSE, PACKAGE = "ape")[[7]]
```

Although the C program is slightly shorter than its R version, the way arguments are passed is more complex and needs more caution. It is possible to compare the performance of both approaches (Table 7.2).

Table 7.2. Comparative speed (in seconds) of two programs performing the same task on phylogenetic trees with n tips (times measured with the function `system.time`)

n	Pure R	R + C
100	0.04	< 0.01
1000	2.19	< 0.01
2000	6.62	0.01
5000	38.63	0.04
10,000	185.13	0.15

Two comments arise from this comparison. First, a program written in pure R can be very fast with small data sets: 0.04 s is actually negligible. In practice, a tree with more than 500 tips is not readable when plotted directly on the screen. The second comment is that with large data sets the gain in speed is critical, and this should be considered when developing computationally intensive methods.

A critical issue in R programming is *vectorization*. This means that repeated calls to compiled codes by the interpreter are avoided. For instance, when generating random variables, the number of independent replicates, say 100, is passed as argument, thus the compiled code is called only once which is more efficient than calling it 100 times. To fix ideas, we can use a trivial example consisting of the sum of many numbers. Say we generate 1,000,000 normal random variables with mean zero and variance unity, and we want to compute their sum. Ignoring the (vectorized) function `sum`, a possible solution could be:

```
x <- rnorm(1e6)
s <- 0
for (i in 1:1e6) s <- s + x[i]
```

The time needed to perform the `for` loop that does the summation is 2.5 s. Of course, a beginner with R quickly learns that there is the function `sum` and will never do the above: `sum(x)` actually takes 0.01 s.

The use of vectorization may be less obvious. Consider we want to sum only the negative values of `x`; the most intuitive approach may be to use an `if` statement such as:

```
s <- 0
for (i in 1:1e6) if (x[i] < 0) s <- s + x[i]
```

This takes 3.5 s to be completed. A vectorized version is possible with logical indexing:

```
sum(x[x < 0])
```

The computation time is now 0.12 s. To do the same task with a dedicated compiled C code, we can write the following function,

```
#include <R.h>

void sum_neg(double *x, int *n, double *sum)
{
  int i;

  *sum = 0;
  for (i = 0; i < *n; i++) {
    if (x[i] < 0) *sum += x[i];
  }
}
```

and call it (after compilation) from R with the function:

```
sumneg <- function(x)
{
    sumneg <- 0
    ans <- .C("sum_neg", as.double(x), as.integer(length(x)),
              as.double(sumneg), package = "apex")
    ans[[3]]
}
```

The time needed to complete `sumneg(x)` is 0.09 s. The gain will obviously be even smaller with a smaller data set. This shows clearly that writing compiled code may not always be advantageous with R.

The crucial point, in terms of performance, is thus whether vectorization can be achieved in an R program. We have seen above an example where a C code was used to manipulate objects of class `"phylo"`. This is a case where vectorization cannot be done easily because we need to manipulate the elements in a complex way so that we need repeated loops and `if` statements.

However, vectorization can be achieved in some cases with objects of class `"phylo"`. The functions `birthdeath`, `yule`, or `yule.cov` provide some examples. For instance, the speciation rate estimator under the Yule model is $\hat{\lambda} = B_T/X_T$ where B_T is the number of observed branching events during time T, and X_T is the sum of all branch lengths during the same time [78]. This estimator can be computed for a tree, say `tr`, relatively easily:

```
-min(as.numeric(tr$edge)) / sum(tr$edge.length)
```

This considers that the nodes are numbered with negative numbers, thus the smallest one is the number of nodes. The branch lengths are stored in a single numeric vector, thus the second term is easily computed.

A strategy often used by R developers is to first develop the program in pure R. When it is stable and some “computational bottlenecks” have been eventually identified, some tasks can be transferred to C programs. A mixed strategy is to keep the most complex data manipulation (e.g., involving lists, names, etc.) in R, and using compiled codes to do computations on vectors: this is the strategy used in `plot.phylo`.

References

[1] Agapow P.-M. & Purvis A. 2002. Power of eight tree shape statistics to detect nonrandom diversification: A comparison by simulation of two models of cladogenesis. *Systematic Biology* **51**: 866–872.

[2] Aldous D. 1996. Probability distributions on cladograms. In: *Random Discrete Structures*, Aldous D. & Pemantle R., editors, pages 1–18. IMA,.

[3] Aldous D. J. 2001. Stochastic models and descriptive statistics for phylogenetic trees, from Yule to today. *Statistical Science* **16**: 23–34.

[4] Baldauf S. L. 2003. Phylogeny for the faint of heart: A tutorial. *Trends in Genetics* **19**: 345–351.

[5] Baldauf S. L., Bhattacharya D., Cockrill J., Hugenholtz P., Pawlowski J. & Simpson A. G. B. 2004. The tree of life: An overview. In: *Assembling the tree of life*, Cracraft J. & Donoghue M. J., editors, pages 43–75. Oxford University Press, Oxford.

[6] Barhen J., Protopopescu V. & Reister D. 1997. TRUST: A deterministic algorithm for global optimization. *Science* **276**: 1094–1097.

[7] Barndorff-Nielsen O. E. & Shephard N. 2001. Non-Gaussian Ornstein–Uhlenbeck-based models and some of their uses in financial economics (with discussion). *Journal of the Royal Statistical Society. Series B. Methodological* **63**: 167–241.

[8] Becker R. A., Chambers J. M. & Wilks A. R. 1988. *The New S Language.* Chapman & Hall, London.

[9] Billera L. J., Holmes S. P. & Vogtmann K. 2001. Geometry of the space of phylogenetic trees. *Advances in Applied Mathematics* **27**: 733–767.

[10] Böhning-Gaese K., Schuda M. D. & Helbig A. J. 2003. Weak phylogenetic effects on ecological niches of *Sylvia* warblers. *Journal of Evolutionary Biology* **16**: 956–965.

[11] Bokma F. 2003. Testing for equal rates of cladogenesis in diverse taxa. *Ecology* **57**: 2469–2474.

[12] Brocchieri L. 2001. Phylogenetic inferences from molecular sequences: Review and critique. *Theoretical Population Biology* **59**: 27–40.

[13] Buckland S. T., Burnham K. P. & Augustin N. H. 1997. Model selection: An integral part of inference. *Biometrics* **53**: 603–618.

[14] Burnham K. P. & Anderson D. R. 2002. *Model Selection and Multimodel Inference. A Practical Information-Theoretic Approach (Second Edition).* Springer, New York.

[15] Burnham K. P. & White G. C. 2002. Evaluation of some random effects methodology applicable to bird ringing data. *Journal of Applied Statistics* **29**: 245–264.

[16] Butler M. A. & King A. A. 2004. Phylogenetic comparative analysis: A modeling approach for adaptive evolution. *American Naturalist* **164**: 683–695.

[17] Chenna R., Sugawara H., Koike T., Lopez R., Gibson T. J., Higgins D. G. & Thompson J. D. 2003. Multiple sequence alignment with the Clustal series of programs. *Nucleic Acids Research* **31**: 3497–3500.

[18] Cheverud J. M., Dow M. M. & Leutenegger W. 1985. The quantitative assessment of phylogenetic constraints in comparative analyses: Sexual dimorphism in body weight among primates. *Evolution* **39**: 1335–1351.

[19] Chor B. & Tuller T. 2005. Maximum likelihood of evolutionary trees: Hardness and approximation. *Bioinformatics* **21**: i97–i106.

[20] Cox D. R. & Oakes D. 1984. *Analysis of Survival Data.* Monographs on statistics and applied probability. Chapman and Hall, London.

[21] Crosbie S. F. & Manly B. F. J. 1985. Parsimonious modelling of capture-mark-recapture studies. *Biometrics* **41**: 385–398.

[22] Darwin C. 1859. *On the Origin of Species by Means of Natural Selection.* John Murray, London.

[23] Dempster A. P., Laird N. M. & Rubin D. B. 1977. Maximum likelihood from incomplete data via the *EM* algorithm (with discussion). *Journal of the Royal Statistical Society. Series B. Methodological* **39**: 1–38.

[24] Desdevises Y., Legendre P., Azouzi L. & Morand S. 2003. Quantifying phylogenetically structured environmental variation. *Evolution* **57**: 2647–2652.

[25] Diaconis P. W. & Holmes S. P. 1998. Matchings and phylogenetic trees. *Proceedings of the National Academy of Sciences USA* **95**: 14600–14602.

[26] Diniz-Filho J. A. F., de Sant'Ana C. E. R. & Bini L. M. 1998. An eigenvector method for estimating phylogenetic inertia. *Evolution* **52**: 1247–1262.

[27] Edwards A. W. F. 1992. *Likelihood (Expanded Edition).* Johns Hopkins University Press, Baltimore.

[28] Edwards A. W. F. 1998. *History and Philosophy of Phylogeny Methods.* Talk at the EC Summer School Methods for Molecular Phylogenies, Newton Institute, Cambridge, UK.

[29] Efron B. 1981. Nonparametric estimates of standard error: the jacknife, the bootstrap and other methods. *Biometrika* **68**: 589–599.

[30] Efron B. 1998. R. A. Fisher in the 21st century (with discussion). *Statistical Science* **13**: 95–114.

[31] Efron B., Halloran E. & Holmes S. 1996. Bootstrap confidence levels for phylogenetic trees. *Proceedings of the National Academy of Sciences USA* **93**: 13429–13434.

[32] Efron B. & Tibshirani R. 1991. Statistical analysis in the computer age. *Science* **253**: 390–395.

[33] Emerson B., Paradis E. & Thbaud C. 2001. Revealing the demographic histories of species using DNA sequences. *Trends in Ecology & Evolution* **16**: 707–716.

[34] Felsenstein J. 1981. Evolutionary trees from DNA sequences: A maximum likelihood approach. *Journal of Molecular Evolution* **17**: 368–376.

[35] Felsenstein J. 1985. Confidence limits on phylogenies: An approach using the bootstrap. *Evolution* **39**: 783–791.

[36] Felsenstein J. 1985. Phylogenies and the comparative method. *American Naturalist* **125**: 1–15.

[37] Felsenstein J. 1988. Phylogenies and quantitative characters. *Annual Review of Ecology and Systematics* **19**: 445–471.

[38] Felsenstein J. 1993. *Phylip (Phylogeny Inference Package) Version 3.5c.* http://evolution.genetics.washington.edu/phylip/phylip.html. Department of Genetics, University of Washington, Seattle.

[39] Felsenstein J. 2004. *Inferring Phylogenies.* Sinauer Associates, Sunderland, MA.

[40] Felsenstein J. & Churchill G. A. 1996. A Hidden Markov model approach to variation among sites in rate of evolution. *Molecular Biology and Evolution* **13**: 93–104.

[41] Fry B. G. 2005. From genome to "venome": molecular origin and evolution of the snake venom proteome inferred from phylogenetic analysis of toxin sequences and related body proteins. *Genome Research* **15**: 403–420.

[42] Futuyma D. J. 1998. *Evolutionary Biology (Third Edition).* Sinauer Associates, Sunderland, MA.

[43] Galtier N. & Gouy M. 1995. Inferring phylogenies from DNA sequences of unequal base compositions. *Proceedings of the National Academy of Sciences USA* **92**: 11317–11321.

[44] Galtier N. & Gouy M. 1998. Inferring pattern and process: Maximum-likelihood implementation of a nonhomogeneous model of DNA sequence evolution for phylogenetic analysis. *Molecular Biology and Evolution* **15**: 871–879.

[45] Garland, Jr. T. & Adolph S. C. 1991. Physiological differentiation of vertebrate populations. *Annual Review of Ecology and Systematics* **22**: 193–228.

[46] Garland, Jr. T. & Carter P. A. 1994. Evolutionary physiology. *Annual Review of Physiology* **56**: 579–621.

[47] Garland, Jr. T., Dickerman A. W., Janis C. M. & Jones J. A. 1993. Phylogenetic analysis of covariance by computer simulation. *Systematic Biology* **42**: 265–292.

[48] Garland, Jr. T., Harvey P. H. & Ives A. R. 1992. Procedures for the analysis of comparative data using phylogenetically independent contrasts. *Systematic Biology* **41**: 18–32.

[49] Gentleman R. 2004. Some perspectives on statistical computing. *Canadian Journal of Satistics* **32**: 209–226.

[50] Giannini N. P. 2003. Canonical phylogenetic ordination. *Systematic Biology* **52**: 684–695.

[51] Gibson A., Gowri-Shankar V., Higgs P. G. & Rattray M. 2005. A comprehensive analysis of mammalian mitochondrial genome base composition and improved phylogenetic methods. *Molecular Biology and Evolution* **22**: 251–264.

[52] Gittleman J. L. 1986. Carnivore life history patterns: Allometric, phylogenetic and ecological associations. *American Naturalist* **127**: 744–771.

[53] Gittleman J. L. & Kot M. 1990. Adaptation: Statistics and a null model for estimating phylogenetic effects. *Systematic Zoology* **39**: 227–241.

[54] Grafen A. 1989. The phylogenetic regression. *Philosophical Transactions of the Royal Society of London. Series B. Biological Sciences* **326**: 119–157.

[55] Grafen A. & Ridley M. 1996. Statistical tests for discrete cross-species data. *Journal of Theoretical Biology* **183**: 255–267.

[56] Grafen A. & Ridley M. 1997. A new model for discrete character evolution. *Journal of Theoretical Biology* **184**: 7–14.

[57] Grafen A. & Ridley M. 1997. Non-independence in statistical tests for discrete cross-species data. *Journal of Theoretical Biology* **188**: 507–514.

[58] Guindon S. & Gascuel O. 2003. A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. *Systematic Biology* **52**: 696–704.

[59] Hackett S. J. 1996. Molecular phylogenetics and biogeography of tanagers in the genus *Ramphocelus* (Aves). *Molecular Phylogenetics and Evolution* **5**: 368–382.

[60] Hall B. G. 2004. *Phylogenetic Trees Made Easy: A how-to Manual (Second Edition).* Sinauer Associates, Sunderland, MA.

[61] Hansen T. F. 1997. Stabilizing selection and the comparative analysis of adaptation. *Evolution* **51**: 1341–1351.

[62] Hansen T. F. & Martins E. P. 1996. Translating between microevolutionary process and macroevolutionary patterns: The correlation structure of interspecific data. *Evolution* **50**: 1404–1417.

[63] Harvey P. H., May R. M. & Nee S. 1994. Phylogenies without fossils. *Evolution* **48**: 523–529.

[64] Harvey P. H. & Pagel M. D. 1991. *The comparative Method in Evolutionary Biology.* Oxford University Press, Oxford.

[65] Harvey P. H. & Purvis A. 1991. Comparative methods for explaining adaptations. *Nature* **351**: 619–624.

[66] Hasegawa M., Kishino H. & Yano T.-a. 1985. Dating of the human-ape splitting by a molecular clock of mitochondrial DNA. *Journal of Molecular Evolution* **22**: 160–174.

[67] Hebert P. D. N., Penton E. H., Burns J. M., Janzen D. H. & Hallwachs W. 2004. Ten species in one: DNA barcoding reveals cryptic species in the neotropical skipper butterfly *Astraptes fulgerator*. *Proceedings of the National Academy of Sciences USA* **101**: 14812–14817.

[68] Holder M. & Lewis P. O. 2003. Phylogeny estimation: Traditional and Bayesian approaches. *Nature Reviews Genetics* **4**: 275–284.

[69] Holmes S. 2003. Statistics for phylogenetic trees. *Theoretical Population Biology* **63**: 17–32.

[70] Housworth E. A., Martins E. P. & Lynch M. 2004. The phylogenetic mixed model. *American Naturalist* **163**: 84–96.

[71] Huelsenbeck J. P. & Rannala B. 1997. Phylogenetic methods come of age: testing hypotheses in an evolutionary context. *Science* **276**: 227–232.

[72] Huelsenbeck J. P., Rannala B. & Masly J. P. 2000. Accomodating phylogenetic uncertainty in evolutionary studies. *Science* **288**: 2349–2350.

[73] Hunter J. P. 1998. Key innovations and the ecology of macroevolution. *Trends in Ecology & Evolution* **13**: 31–36.

[74] Ihaka R. & Gentleman R. 1996. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics* **5**: 299–314.

[75] Johnson W. E. & O'Brien S. J. 1997. Phylogenetic reconstruction of the Felidae using 16S rRNA and NADH-5 mitochondrial genes. *Journal of Molecular Evolution* **44**: S98–S116.

[76] Jones K. E., Purvis A., MacLarnon A., Bininda-Emonds O. R. P. & Simmons N. B. 2002. A phylogenetic supertree of the bats (Mammalia: Chiroptera). *Biological Reviews of the Cambridge Philosophical Society* **77**: 223–259.

[77] Jukes T. H. & Cantor C. R. 1969. Evolution of protein molecules. In: *Mammalian Protein Metabolism*, Munro H. N., editor, pages 21–132. Academic Press, New York.

[78] Keiding N. 1975. Maximum likelihood estimation in the birth-and-death process. *Annals of Statistics* **3**: 363–372.

[79] Kendall D. G. 1948. On the generalized "birth-and-death" process. *Annals of Mathematical Statistics* **19**: 1–15.

[80] Kendall D. G. 1949. Stochastic processes and population growth. *Journal of the Royal Statistical Society. Series B. Methodological* **11**: 230–264.

[81] Kimura M. 1980. A simple method for estimating evolutionary rates of base substitutions through comparative studies of nucleotide sequences. *Journal of Molecular Evolution* **16**: 111–120.

[82] Kimura M. 1981. Estimation of evolutionary distances between homologous nucleotide sequences. *Proceedings of the National Academy of Sciences USA* **78**: 454–458.

[83] Kirkpatrick M. & Slatkin M. 1993. Searching for evolutionary patterns in the shape of a phylogenetic tree. *Evolution* **47**: 1171–1181.

[84] Kosakovsky Pond S. L. & Muse S. V. 2004. Column sorting: rapid calculation of the phylogenetic likelihood function. *Systematic Biology* **53**: 685–692.

[85] Kosiol C. & Goldman N. 2005. Different versions of the Dayhoff rate matrix. *Molecular Biology and Evolution* **22**: 193–199.

[86] Lachaud B. 2005. Cut-off and hitting times of a sample of Ornstein–Uhlenbeck processes and its average. *Journal of Applied Probability* **42**: 1069–1080.

[87] Lanave C., Preparata G., Saconne C. & Serio G. 1984. A new method for calculating evolutionary substitution rates. *Journal of Molecular Evolution* **20**: 86–93.

[88] Larget B., Simon D. L. & Kadane J. B. 2002. Bayesian phylogenetic inference from animal mitochondrial genome arrangements. *Journal of the Royal Statistical Society. Series B. Methodological* **64**: 681–693.

[89] Lecompte É., Granjon L., Peterhans J. K. & Denys C. 2002. Cytochrome b-based phylogeny of the *Praomys* group (Rodentia, Murinae): A new African radiation? *Comptes Rendus Biologies* **325**: 827–840.

[90] Legendre P. & Makarenkov V. 2002. Reconstruction of biogeographic and evolutionary networks using reticulograms. *Systematic Biology* **51**: 199–216.

[91] Leisch F. 2002. Dynamic generation of statistical reports using literate data analysis. In: *Compstat 2002—Proceedings in Computational Statistics*, Haerdle W. & Roenz B., editors, pages 575–580. Physika Verlag, Heidelberg.

[92] Liang K.-Y. & Zeger S. L. 1986. Longitudinal data analysis using generalized linear models. *Biometrika* **73**: 13–22.

[93] Losos J. B. & Adler F. R. 1995. Stumped by trees? A generalized null model for patterns of organismal diversity. *American Naturalist* **145**: 329–342.

[94] Lynch M. 1991. Methods for the analysis of comparative data in evolutionary biology. *Evolution* **45**: 1065–1080.

[95] Maddison D. R., Swofford D. L. & Maddison W. P. 1997. NEXUS: An extensible file format for systematic information. *Systematic Biology* **46**: 590–621.

[96] Martins E. P. & Hansen T. F. 1997. Phylogenies and the comparative method: A general approach to incorporating phylogenetic information into the analysis of interspecific data [erratum in vol. 153, no. 4, p. 488]. *American Naturalist* **149**: 646–667.

[97] McCullough B. D. 1999. Assessing the reliability of statistical software: Part II. *American Statistician* **53**: 149–159.

[98] McCullough B. D. & Vinod H. D. 1999. The numerical reliability of econometric software. *Journal of Economic Literature* **37**: 633–665.

[99] McLeod A. I. 1993. Parsimony, model adequacy and periodic correlation in time-series forecasting. *International Statistical Review* **61**: 387–393.

[100] Michaux J., Chevret P., Filipucci M.-G. & Macholan M. 2002. Phylogeny of the genus *Apodemus* with a special emphasis on the subgenus *Sylvaemus* using the nuclear IRBP gene and two mitochondrial markers: cytochrome *b* and 12S rRNA. *Molecular Phylogenetics and Evolution* **23**: 123–136.

[101] Minin V., Abdo Z., Joyce P. & Sullivan J. 2003. Performance-based selection of likelihood models for phylogeny estimation. *Systematic Biology* **52**: 674–683.

[102] Moran P. A. P. 1950. Notes on continuous stochastic phenomena. *Biometrika* **37**: 17–23.

[103] Nee S., Holmes E. C., Rambaut A. & Harvey P. H. 1995. Inferring population history from molecular phylogenies. *Philosophical Transactions of the Royal Society of London. Series B. Biological Sciences* **349**: 25–31.

[104] Nee S., May R. M. & Harvey P. H. 1994. The reconstructed evolutionary process. *Philosophical Transactions of the Royal Society of London. Series B. Biological Sciences* **344**: 305–311.

[105] Nee S., Mooers A. Ø. & Harvey P. H. 1992. Tempo and mode of evolution revealed from molecular phylogenies. *Proceedings of the National Academy of Sciences USA* **89**: 8322–8326.

[106] Nei M. & Kumar S. 2000. *Molecular Evolution and Phylogenetics.* Oxford University Press, Oxford.

[107] Oakley T. H. 2003. Maximum likelihood models of trait evolution. *Comments on Theoretical Biology* **8**: 1–17.

[108] Ollier S., Couteron P. & Chessel D. 2005. Orthonormal transform to decompose the variance of a life-history trait across a phylogenetic tree. *Biometrics* doi:10.1111/j.1541-0420.2005.00497.x.

[109] Pagel M. 1994. Detecting correlated evolution on phylogenies: A general method for the comparative analysis of discrete characters. *Proceedings of the Royal Society of London. Series B. Biological Sciences* **255**: 37–445.

[110] Pagel M. & Meade A. 2004. A phylogenetic mixture model for detecting pattern-heterogeneity in gene sequence or character-state data. *Systematic Biology* **53**: 571–581.

[111] Paradis E. 1997. Assessing temporal variations in diversification rates from phylogenies: Estimation and hypothesis testing. *Proceedings of the Royal Society of London. Series B. Biological Sciences* **264**: 1141–1147.

[112] Paradis E. 1998. Testing for constant diversification rates using molecular phylogenies: A general approach based on statistical tests for goodness of fit. *Molecular Biology and Evolution* **15**: 476–479.

[113] Paradis E. 2003. Analysis of diversification: Combining phylogenetic and taxonomic data. *Proceedings of the Royal Society of London. Series B. Biological Sciences* **270**: 2499–2505.

[114] Paradis E. 2004. Can extinction rates be estimated without fossils? *Journal of Theoretical Biology* **229**: 19–30.

[115] Paradis E. 2005. Statistical analysis of diversification with species traits. *Evolution* **59**: 1–12.

[116] Paradis E. & Claude J. 2002. Analysis of comparative data using generalized estimating equations. *Journal of Theoretical Biology* **218**: 175–185.

[117] Paradis E., Claude J. & Strimmer K. 2004. APE: Analyses of phylogenetics and evolution in R language. *Bioinformatics* **20**: 289–290.

[118] Penny D. & Hendy M. D. 1985. The use of tree comparison metrics. *Systematic Zoology* **34**: 75–82.

[119] Pinheiro J. C. & Bates D. M. 2000. *Mixed-Effects Models in S and S-PLUS*. Springer, New York.

[120] Posada D. & Buckley T. R. 2004. Model selection and model averaging in phylogenetics: Advantages of Akaike information criterion and Bayesian approaches over likelihood ratio tests. *Systematic Biology* **53**: 793–808.

[121] Posada D. & Crandall K. A. 1998. MODELTEST: Testing the model of DNA substitution. *Bioinformatics* **14**: 817–818.

[122] Posada D. & Crandall K. A. 2001. Selecting the best-fit model of nucleotide substitution. *Systematic Biology* **50**: 580–601.

[123] Pupko T., Huchon D., Cao Y., Okada N. & Hasegawa M. 2002. Combining multiple data sets in a likelihood analysis: Which models are the best? *Molecular Biology and Evolution* **19**: 2294–2307.

[124] Purvis A. & Garland, Jr. T. 1993. Polytomies in comparative analyses of continuous characters. *Systematic Biology* **42**: 569–575.

[125] Pybus O. G. & Harvey P. H. 2000. Testing macro-evolutionary models using incomplete molecular phylogenies. *Proceedings of the Royal Society of London. Series B. Biological Sciences* **267**: 2267–2272.

[126] Pybus O. G., Rambaut A., Holmes E. C. & Harvey P. H. 2002. New inferences from tree shape: Numbers of missing taxa and population growth rates. *Systematic Biology* **51**: 881–888.

[127] Quader S., Isvaran K., Hale R. E., Miner B. G. & Seavy N. E. 2004. Nonlinear relationships and phylogenetically independent contrasts. *Journal of Evolutionary Biology* **17**: 709–715.

[128] R Development Core Team. 2005. *R Language Definition. Version 2.2.0.* R Foundation for Statistical Computing, Vienna.

[129] R Development Core Team. 2005. *Writing R Extensions. Version 2.2.0.* R Foundation for Statistical Computing, Vienna.

[130] Read A. F. & Nee S. 1995. Inference from binary comparative data. *Journal of Theoretical Biology* **173**: 99–108.

[131] Ridley M. 1992. Darwin sound on comparative method. *Trends in Ecology & Evolution* **7**: 37.

[132] Rohlf F. J. 2001. Comparative methods for the analysis of continuous variables: Geometric interpretations. *Evolution* **55**: 2143–2160.

[133] Rzhetsky A. & Nei M. 1992. A simple method for estimating and testing minimum-evolution trees. *Molecular Biology and Evolution* **9**: 945–967.

[134] Saitou N. & Nei M. 1987. The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution* **4**: 406–425.

[135] Sanderson M. J. 1997. A nonparametric approach to estimating divergence times in the absence of rate constancy. *Molecular Biology and Evolution* **14**: 1218–1231.

[136] Sanderson M. J. 2002. Estimating absolute rates of molecular evolution and divergence times: A penalized likelihood approach. *Molecular Biology and Evolution* **19**: 101–109.

[137] Sanderson M. J., Purvis A. & Henze C. 1998. Phylogenetic supertrees: Assembling the trees of life. *Trends in Ecology & Evolution* **13**: 105–109.

[138] Schluter D., Price T., Mooers A. Ø. & Ludwig D. 1997. Likelihood of ancestor states in adaptive radiation. *Evolution* **51**: 1699–1711.

[139] Schnabel R. B., Koontz J. E. & Weiss B. E. 1985. A modular system of algorithms for unconstrained minimization. *ACM Transactions on Mathematical Software* **11**: 419–440.

[140] Sibley C. G. & Ahlquist J. E. 1990. *Phylogeny and Classification of Birds: A Study in Molecular Evolution.* Yale University Press, New Haven, CT.

[141] Sibley C. G. & Monroe, Jr. B. L. 1990. *Distribution and Taxonomy of Birds of the World.* Yale University Press, New Haven, CT.

[142] Skelton P., editor. 1993. *Evolution: A Biological and Palaeontological Approach.* Addison-Wesley and The Open University, Harlow, UK.

[143] Smith F. A., Lyons S. K., Ernest S. K. M., Jones K. E., Kaufman D. M., Dayan T., Marquet P. A., Brown J. H. & Haskell J. P. 2003. Body mass of late quaternary mammals. *Ecology* **84**: 3403.

[144] Stamatakis A., Ludwig T. & Meier H. 2005. RAxML-III: A fast program for maximum likelihood-based inference of large phylogenetic trees. *Bioinformatics* **21**: 456–463.

[145] Stauffer R. L., Walker A., Ryder O. A., Lyons-Weiler M. & Hedges S. B. 2001. Human and ape molecular clocks and constraints on paleontological hypotheses. *Journal of Heredity* **92**: 469–474.

[146] Stephens M. A. 1974. EDF statistics for goodness of fit and some comparisons. *Journal of American Statistical Association* **69**: 730–737.

[147] Stephens M. A. 1982. Anderson-Darling test for goodness of fit. In: *Encyclopedia of Statistical Science. Volume 1*, Kotz S. & Johnson N. L., editors, pages 81–85. John Wiley & Sons, New York.

[148] Suzuki Y., Glazko G. V. & Nei M. 2002. Overcredibility of molecular phylogenies obtained by Bayesian phylogenetics. *Proceedings of the National Academy of Sciences USA* **99**: 16138–16143.

[149] Tallis G. M. 1983. Goodness of fit. In: *Encyclopedia of Statistical Science. Volume 3*, Kotz S. & Johnson N. L., editors, pages 451–461. John Wiley & Sons, New York.

[150] Tamura K. 1992. Estimation of the number of nucleotide substitutions when there are strong transition-transversion and G+C-content biases. *Molecular Biology and Evolution* **9**: 678–687.

[151] Tamura K. & Nei M. 1993. Estimation of the number of nucleotide substitutions in the control region of mitochondrial DNA in humans and chimpanzees. *Molecular Biology and Evolution* **10**: 512–526.

[152] Tamura K., Nei M. & Kumar S. 2004. Prospects for inferring very large phylogenies by using the neighbor-joining method. *Proceedings of the National Academy of Sciences USA* **101**: 11030–11035.

[153] Thompson J. D., Gibson T. J., Plewniak F., Jeanmougin F. & Higgins D. G. 1997. The CLUSTAL_X windows interface: Flexible strategies for multiple sequence alignment aided by quality analysis tools. *Nucleic Acids Research* **25**: 4876–4882.

[154] Venables W. N. & Ripley B. D. 2002. *Modern Applied Statistics with S (Fourth Edition).* Springer, New York.

[155] Whelan S., Liò P. & Goldman N. 2001. Molecular phylogenetics: state-of-the-art methods for looking into the past. *Trends in Genetics* **17**: 262–272.

[156] Yang Z. 1994. Estimating the pattern of nucleotide substitution. *Journal of Molecular Evolution* **39**: 105–111.

[157] Yang Z. 1994. Maximum likelihood phylogenetic estimation from DNA sequences with variable rates over sites: Approximate methods. *Journal of Molecular Evolution* **39**: 306–314.

[158] Yang Z. 1996. Maximum-likelihood models for combined analyses of multiple sequence data. *Journal of Molecular Evolution* **42**: 587–596.

[159] Yang Z. 2000. Maximum likelihood estimation on large phylogenies and analysis of adaptive evolution in human influenza virus A. *Journal of Molecular Evolution* **51**: 423–432.

Index