Use R!

Analysis of Phylogenetics and Evolution with R

Second Edition

Springer

Use R!

Series Editors:
Robert Gentleman Kurt Hornik Giovanni G. Parmigiani

For further volumes:
http://www.springer.com/series/6991

Emmanuel Paradis

Analysis of Phylogenetics and Evolution with R

Second Edition

Springer

Emmanuel Paradis
Institut de Recherche pour le
Développement (IRD)
Institut des Sciences de l'Évolution
(ISEM) – UMR 226 IRD/CNRS/UM2
Jl. Taman Kemang 32B
Jakarta 12730
Indonesia
emmanuel.paradis@ird.fr

ISBN 978-1-4614-1742-2 e-ISBN 978-1-4614-1743-9
DOI 10.1007/978-1-4614-1743-9
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011940221

to Laure

Preface to the Second Edition

> Some kinds of problems may yield to sophisticated statistical technique; others will not. The goal of empirical research is—or should be—to increase our understanding of the phenomena, rather than displaying our mastery of technique.
>
> —Freedman [91]

The first edition of *APER* was published five years ago. The subject of the book has grown in importance and audience during this time, so this second edition is certainly timely. Moreover, the changes in `ape` and several related packages have been so important in the last few years, that, for some time, I have felt this new edition was required.

Though I acknowledge that the first edition of *APER* was quite appreciated by researchers, lecturers and students, it was not out of criticism. With this second edition, I have tried to improve on all possible points and issues as much as I was able to do. At the same time, I had to think over on the progress of the approach I presented five years ago. A difficult task was to review the development of the many new packages contributing to phylogenetics and evolutionary biology. It was my choice to focus on some of these packages with the aim to provide to the reader a consistent set of "techniques" to address a wide range of issues in phylogenetics and macroevolution. Even restricting our interest to these two fields, a number of packages are not considered in this book: this is a choice I fully assume. Besides, I decided to not tackle two related fields: population genetics and paleobiology. Both have seen critical developments in the form of several packages. A useful treatment of these would require two new books, so I have limited myself here to only mention some data structures in Chapter 3.

The general structure of this second edition is very similar to its predecessor. Chapters 1–4 focus on the "technique" of R and several specialized packages for phylogenetics and evolutionary biology with the aim to give to the readers the tools to manipulate, summarize, and display their data. Chapters 5 on estimation of phylogeny, and 6 on analysis of macroevolution,

are more concerned with "understanding the phenomena" of evolution. Compared to the first edition, these two chapters have been much expanded and emphasized. Though I have tried to go deeper in the inferential and biological processes, a complete treatment of these questions would require several volumes, and is therefore limited here to the essential points. The new Chapter 7 covers the simulation of evolutionary data, a topic which, I believe, deserves to be treated with more rigor than done until now. Chapter 8 concludes the book on the development of phylogenetic computational methods in R. This chapter has been expanded using essentially my experience over the last few years. Because the volume of the book has been nearly doubled, the number of the case studies has been reduced.

At the time of completing this second edition, a number of packages have been pre-released or are under development. These will, in the years to come, increase dramatically the quantity of software available to evolutionary biologists to analyze their data. Even the packages described in details in this book continue their development. For instance, ape will soon include most published methods to estimate phylogenies from incomplete distance matrices thanks to the effort of Andrei-Alin Popescu. Some other packages, such as ade4, diversitree, phangorn, or seqinr, have provided a range of tools and methods to many evolutionists for some time, and are under continuous development.

An interesting phenomenon emerged recently in the scientific literature: many authors provide R code as supplementary information to their articles. Though we can be worried about the proliferation of sources of R programs, this is very good news for our speciality as it marks a significant progress towards a wider acceptance of the concept of repeatable research in data analysis. This, and the above, testify that using R as a computational tool for phylogenetics and evolutionary biology is meant to last.

Several colleagues kindly read some parts of the manuscript of this second edition: Christophe Douady, Susan Holmes, Nicolas Hubert, Anthony Ives, and Errol Rowe. I am grateful to Thibaut Jombart for reading several chapters and for the stimulating discussions, and to Klaus Schliep for our many discussions. Many thanks also to my colleagues in Nescent for organizing the Hackathon in December 2007. Thank you to Régis Hocdé for helping me to obtain the web site ape.mpl.ird.fr, and the repository for the source of ape; thank you also to the IRD staff for maintaining them over these years. Writing a book is a tremendeous experience. Having the chance to correct and amend it is another unique experience. I thank warmly John Kimmel and Marc Strauss for giving me this opportunity. I am sincerely thankful to a long list of users, colleagues, and friends for their interest and enthusiasm.

I am grateful to Laure and Sinta for their constant support and patience.

Jakarta Emmanuel Paradis
August 2011

Preface to the First Edition

> As a result, the inference of phylogenies often seems divorced from any connection to other methods of analysis of scientific data.
>
> —Felsenstein [78]

> Once calculation became easy, the statistician's energies could be devoted to understanding his or her dataset.
>
> —Venables & Ripley [307]

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 [62, 63]. 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 [146]. 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

E. Paradis, *Analysis of Phylogenetics and Evolution with R*, Use R,
DOI 10.1007/978-1-4614-1743-9_1, © Springer Science+Business Media, LLC 2012

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. Besides, integrating phylogenetic methods with standard and leading-edge statistical methods is unique to R.

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 [38, 104]. 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

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

updated. It is thus very useful to automate such analyses, particularly if they are composed of a long series of smaller analyses.

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 8). R is an interpreted language meaning that there is no need to develop a full program to perform an analysis. An analysis may be done with 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 [104]. 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 [181] (see also `?Sweave` in R). The package odfWeave gives this functionality using ODF documents as used in LibreOffice or OpenOffice instead of LaTeX. Even without employing these sophisticated tools, writing R scripts is a must and should be taught in all courses on R. Several text editors allow to edit R scripts with enhanced efficiency (see Section 1.3).

Evolvability

Phylogenetic methods have considerably evolved for several decades, and this will 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 relatively 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 on most current laptops and personal computers, 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 machines.

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 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.[2] There have been numerous discussions, particularly on the Internet, about the advantages and inconveniences of free software [262]. 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.[3] 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 [104] 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`:

[2] Type `RShowDoc("COPYING")` in R for details.

[3] For obvious practical reasons, a limited number of persons, namely, the members of the R Core Team, can modify the original sources.

```
> 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:

```
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, computational statistical utilities, and tools for developing packages. Several thousand contributed packages add many specialized methods. 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. This selection of packages is motivated by the issues addressed in this book.

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:

Table 1.1. R packages used for evolutionary analyses

Name	Title	Chapter(s)	Ref.[a]
ade4	Analysis of environmental data	5,6	[41]
adegenet	Spatial and multivariate population genetics	3	[150]
adephylo	Phylogenetic comparative methods	6	[151]
ape	Analyses of phylogenetics and evolution	3–8	[235]
apTreeshape	Analyses of phylogenetic tree shape	6,7	[29]
BoSSA	Sequence retrieval	3	
distory	Distances among trees	5	
diversitree	Tests of diversification	3,6,7	
geiger	Evolutionary diversification	6	[125]
LAGOPUS	Bayesian molecular dating	5	
laser	Speciation and extinction estimation	6	[255]
pegas	Population genetics	3	[231]
phangorn	Phylogeny estimation	3,5,7	[278]
phyclust	Phylogenetic clustering and coalescence	7	
phylobase	Phylogenetic structures and comparative data	3,6	
phyloch	Tools for phylogenetics and taxonomy	3,5	
phyloclim	Phylogenetic climatic niche modeling	6	
picante	Phylogenies and community ecology	6	[158]
seqinr	Exploratory analyses of molecular sequences	3	[39]
spacodiR	Phylogenetic community structure	6	[59]
TreeSim	Tree simulations	7	

[a] `citation` gives how to cite a package (e.g., `citation("ape")`).

```
> library(ape)
> library(ade4)
> library(seqinr)
```

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 shall 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 from GenBank). The Internet connection also helps to keep updated one's system by checking the versions installed on the computer with those available on CRAN; this is done by typing regularly:

```
> update.packages()
```

Other programs may be required in some applications (e.g., Clustal, PhyML): the sites where to download them is indicated in the relevant sections.

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/

Under Windows, a user-friendly alternative to Emacs is Tinn-R:[4] this text editor allows syntax highlighting as well as sending selected lines of commands directly to R.

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. 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.[5] 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 (Appendix A).

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. If the path does start from the root, it is resolved from the working directory. The three following examples are similar to the above one but from different working directories:

[4] http://www.sciviews.org/_rgui/projects/Editors.html

[5] 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.

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

The last two examples show how to go back one or two levels in the directory hierarchy. This is useful when working on several projects and using a standard set of directory names.

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 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.

1.4 Other Readings

Two books come as natural companions of the present one. Felsenstein's *Inferring Phylogenies* [79] provides a thorough background on phylogenetics, including historical aspects and the rationale behind many methods. Freedman's *Statistical Models: Theory and Practice* [91] is probably one of the best presentation of the principles of statistical inference. His criticism on the current practice of some statistical techniques, though particularly focused on social and medical sciences, is very relevant for evolutionary biology.

I assume the reader has basic knowledge on statistics, and that the following concepts are mastered: distribution, variance, estimation, bias, least squares, and maximum likelihood. Wikipedia may be a good place to learn or update knowledge on these concepts out of context.

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. In the last few years, the practice and teaching of R has increased dramatically. So it is likely that most readers will already be familiar with R. The goal of this chapter is more to remind the most fundamental notions of R, rather than to give a formal introduction for new users. 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 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, genetics (quantitative trait loci, population analyses, etc.), morphometrics, ecological analyses, drawing maps including the use of geographic information systems (GIS) data or GoogleMaps, and interacting with a variety of other programs such as SQL and other forms of 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[1] (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.

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

E. Paradis, *Analysis of Phylogenetics and Evolution with R*, Use R,
DOI 10.1007/978-1-4614-1743-9_2, © Springer Science+Business Media, LLC 2012

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 (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 complementary 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 assign operator `<-`. 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 3.8.

The user does not see her data as in a spreadsheet editor (though this is possible) 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 <- 2.2:6
> 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
> 5:-5
 [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5
```

It happens from time to time that one forgets to assign the result of a command to an object. This may be worrying if the command took a long time to run (e.g., downloading a large number of molecular sequences through the Internet) and its results are only printed on the console. R actually stores the last evaluation in a hidden object called `.Last.value` and this may be copied into another object, for instance after typing the last command above:

```
> x <- .Last.value
> x
 [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5
```

2.2 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 unless they are quoted):

```
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 eventually leads to the help page of each function.

The help page of a function or operator is formatted in a standard way with sections, among others, 'Usage' giving all possible arguments and their

eventual default values, 'See also' that lists related help pages (as hyperlinks in HTML and PDF formats), and 'Examples' that illustrates some uses of the documented item(s). Finally, data sets delivered with a package are also documented: for instance, see `?sunspots` for a standard data set in R.

A lot of packages have now *vignettes*, PDF documents with additional information and written in a very free by package authors. The list of vignettes installed on a computer is printed with `vignette()`, and a specific one is viewed by giving its name within double quotes (e.g., `vignette("adephylo")`).

Electronic mailing lists have always been critical in all aspects of the development of R [88]. This phenomenon has amplified in recent years with the emergence of special interest groups hosted on the CRAN, including one in genetics (r-sig-genetics) and one in phylogenetics (r-sig-phylo). These lists play an important role in structuring the community of users in relation to many aspects of data analysis, including theoretical ones.

2.3 The Data Structures

We show here how data are stored in R, and how to manipulate them. Any serious R user must know the contents of this section by heart.

2.3.1 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. Vectors can be of five modes: numeric, logical (`TRUE` or `FALSE`), character, raw, and complex. Usually, only the modes numeric and character are used to store data. Logical vectors are useful to manipulate data. The modes raw and complex are seldom used and are 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
> x >= 3
[1] FALSE FALSE  TRUE  TRUE  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 (e.g., with `read.table` or `read.csv`).

R has a powerful and flexible mechanism to manipulate vectors (and other objects as well as we will see below): 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 (this is a major difference with numeric indexing); 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 >= 5
[1]  TRUE FALSE FALSE  TRUE  TRUE
> x[x >= 5]
[1] 10 10  5
```

A useful function in this context is `which` that takes a logical vector as argument and returns the numeric indices of the values that are `TRUE`:

```
> which(x >= 5)
[1] 1 4 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
```

2.3.2 Factor

A factor is a data structure derived from a vector, and is the basic way to store qualitative variables in R. It is of mode numeric 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 have not been 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 3.8 for examples). This can be avoided by using the option `as.is = TRUE`.

2.3.3 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.

R users often hit the problem that extracting a row or a column from a matrix returns a vector and not a single row, or single column, matrix. This is because the default behavior of the operator `[` is to return an object of the lowest possible dimension. This can be controled with the option `drop` which is `TRUE` by default:

```
> x[3, 2, drop = FALSE]
     [,1]
[1,]    6
> x[3, , drop = FALSE]
     [,1] [,2] [,3]
[1,]    3    6    9
> x[, 2, drop = FALSE]
     [,1]
[1,]    4
[2,]    5
[3,]    6
```

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
```

R has also a data structure called array which generalizes matrices to any number of dimensions.

2.3.4 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 3.8 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`, 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, modifying, or deleting, a column selectively with the operator `$`:

```
> DF$y
[1] 0 1 2 3
> DF$y <- NULL
> colnames(DF)
[1] "z"    "X4.1"
```

2.3.5 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" ""  ""
```

The concepts we have seen on indexing vectors apply also to lists. Additionally, an element of a list may be extracted, or deleted, 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"
```

Note the subtle, but crucial, difference between `[[` and `[`:

```
> str(L[[1]])
 chr [1:4] "order" "family" "genus" "species"
> str(L[1])
List of 1
 $ z: chr [1:4] "order" "family" "genus" "species"
```

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 (see `?device` to see those available on your machine), but mostly the following are available: encapsulated PostScript (EPS), PDF, JPEG, and PNG. These are the most useful formats for scientific publication.

EPS and PDF are the publishers' preferred formats because they are line-based (or vectorial) so they can be scaled without affecting the resolution of the graphics. Choosing between these two formats is not always trivial because they have mostly the same capacities. The main weakness of EPS is that it cannot handle color transparancy (see the option `alpha` in the function `rgb`) while PDF does. PDF files are usually more compact than the equivalent EPS files, but the latter are more easily modified (even by hand-editing since they are stored in simple text files). EPS files are easily converted into PDF with the 'epstoedit' utility provided with GhostScript.

JPEG and PNG are pixel-based (or raster) formats. They should be avoided for scientific publication because scaling-up graphics stored in such a format is likely to show the limit of resolution (an exception may be when a graphic contains a lot of elements so the EPS or PDF file may be too large). On the other hand, these two formats are well-suited for Web pages (PNG is usually better for line-graphics, whereas JPEG works better with pictures).

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. Similarly, for the other formats mentioned above, the function would be `pdf`, `jpeg`, or `png`.

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.copy2eps` and `dev.copy2pdf` which use EPS or PDF device (the file name must be specified by the user). Finally, `dev.print()` sends the current graphics to the printer. Under some operating systems, these commands can be called from the menus.

2.5 Saving and Restoring R Data

R uses two basic formats to save data: ASCII (simple text) and XDR (external data representation[2]). They are both cross-platform. The ASCII format is appropriate 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 latter is very flexible and has many options. The XDR format can store 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")
```

Some care must be taken before calling `load` because if some objects named `x`, `y`, or `z` are present, the above command will erase them without any warning.

In practice, .RData files are useful when you must interrupt your session and you want to continue your work at a later time. In that case, it is better to use `save.image()` or quit R and choose to save the workspace image:

```
> q()
Save workspace image? [y/n/c]: y
```

In both cases a file named '.RData' will be written in the current working directory will all objects in memory. Because these files may not be readable by many other programs, it is safe to keep your original data files (trees, molecular sequences, ...) in their own formats. However, an XDR file may be useful if you want to send a heterogeneous set of data to a colleague: in that case it will be easily loaded in memory with exactly the same attributes.

Apart from these two standard formats, the package foreign, installed by default with R, provides functions for reading and writing data files from a few common statistical computer programs. The CRAN has also some packages for reading a range of more or less specialized data formats (netCDF, HDF5,

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

various GIS and map data formats, medical image formats, PDB for 3-D molecular structures, ...)

All commands typed in R's CLI are stored into memory and can be displayed with the command `history()`, saved into a file (named '.Rhistory' by default) with `savehistory()`, or loaded into memory with `loadhistory()`.

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)
```

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, for instance, `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.

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`[3] 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:

```
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
    .....
}
```

[3] 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`.

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. If a list is structured (i.e., made of lists of objects), `rapply` is a recursive version of `lapply`, applying `FUN` to the non-list elements. By default, the results are returned as a vector, unless the option `how = "replace"` is used.

`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. `by` is a more elaborate function to apply a function `FUN` to the subsets of a vector or a data frame with respect to one or several factors given as a list; its syntax is:

```
by(data, INDICES, FUN, ..., simplify = TRUE)
```

`aggregate` does the same operation than `by` but returns the results in a table rather in a list.

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

```
> replicate(4, rnorm(1))
[1] -1.424699824  0.695066367  0.958153028  0.002594864
> replicate(5, rpois(3, 10))
     [,1] [,2] [,3] [,4] [,5]
[1,]    8   15   13   10   10
[2,]    5    8   14    3    6
[3,]   10   11    7    8    6
```

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: describe 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 text 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: [13]):

```
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 8.

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 reflecting the choices done to design 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 (Sections 2.6 and 8.1.1).

E. Paradis, *Analysis of Phylogenetics and Evolution with R*, Use R,
DOI 10.1007/978-1-4614-1743-9_3, © Springer Science+Business Media, LLC 2012

It is common that the same data can be stored in R with different classes, mainly because they are adapted to different analyses (often in different packages). Also it is common that a data structure evolves because it is realized that the same information can be stored in a better way, or it needs to be extended. The existence of different data classes should not be a problem for users: it is expected that R packages provide functions to convert among them painlessly. We may recall that in R, all actions are done on objects stored in the active memory of the computer. Consequently, different classes are not related to different data file formats.

3.1.1 Trees

ape uses a class called "phylo" to store phylogenetic trees. During the past few years, the majority of new phylogenetic R-packages have used this class so it has become an implicit standard. 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" which is also described below.

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. The package clue extends a wide range of clustering methods and implements its own classes.

The Class "phylo" (ape)

An object of class "phylo" is a list with the following components (see Section 8.6 for a technical description).

edge a two-column matrix where each row represents a branch (or edge) of the tree; the nodes and the tips are symbolized with integers; the n tips are numbered from 1 to n, and the m (internal) nodes from $n+1$ to $n+m$ (the root being $n+1$). For each row, the first column gives the ancestor.

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

tip.label a vector of mode character giving the labels of the tips; the order of these labels corresponds to the integers 1 to n in edge.

Nnode an integer value giving the number of nodes in the tree (m).

node.label (optional) a vector of mode character giving the labels of the nodes (ordered in the same way than tip.label).

`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 `"phylo4"` (phylobase)

This class is derived from `"phylo"` but using an S4 approach for its implementation instead of S3 (see Chapter 8). This makes almost no difference for end-users analyzing their data. The internal structure is similar to the one in `"phylo"`; the following elements are mandatory (and accessed with the operator `@` characteristic of S4):

`edge` it is similar to what is described above for `"phylo"` except that, for rooted trees, the root edge is always included.
`edge.length` same than above; may be empty (i.e., a numeric vector of length zero) if the tree has no branch length.
`label` a vector of mode character giving the labels of the tips (and the nodes if present).
`edge.label` the same for edges.
`order` a character string giving the order of the edges with respect to some algorithms and operations; may be `"unknown"`.
`annote` further information; may be an empty list.

The Class `"matching"` (ape)

Matchings have been introduced by Diaconis and Holmes [56] 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 [56]. 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.
`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).

Ford [87] further investigated coding trees with matchings and related structures, and developed algorithms for their efficient manipulation.

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.

The Class `"haploNet"` (pegas)

This is a class coding for simple networks without reticulations. The main difference with phylogenies is that observed data (e.g., sequences) may be at the node of a network while they are only at the tips of a phylogeny. The structure is based on an edge matrix but this can have additional columns with or without `colnames` like what is returned by the function `haploNet`. This function adds the attributes `"freq"` giving the numbers of each haplotype and `"labels"` with their respective labels.

3.1.2 Networks

Networks are usually coded in the same way than trees with nodes and edges linking them. The distinction is that the path between two nodes may not be unique. Networks are less common than trees in evolutionary studies, maybe because of the difficulty to draw them graphically. Very often, networks are analyzed as a sum of distinct trees: migration or horizontal gene transfer results in reticulations among populations or species but the genetic lineages have a tree-like dynamics. Other forms of networks (splits, consensus) are more conceptual and serve as graphical representations of phylogenetic uncertainty [137, 145].

There are two main classes dedicated to phylogenetic networks: `"evonet"` and `"networx"`, implemented in the packages ape and phangorn, respectively.

They are both derived from the class "phylo". The class "evonet" is built with a rooted phylogeny and an additional matrix named "reticulation" that is similar to the "edge" matrix. In "networx", the reticulations are included in the "edge" matrix. The latter class is used to code consensus networks.

The classes "network" and "igraph" from the packages of the same names are general classes for any kind of networks. Both classes use the same principle of an "edge list matrix" as the class "phylo", and ape provides functions to convert among these formats (p. 61).

BioConductor provides several packages to handle and plot networks using sophisticated algorithms for the layout of nodes.

3.1.3 Splits

A split (or bipartition) is a pair of two exclusive sets of tips (or taxa). For instance, in a phylogenetic tree, each branch defines a split.[1] The class "prop.part" in ape stores splits as a list of vectors of integers and an additional attribute "label" which is a vector of mode character storing the individual labels. Each vector in the list gives a subset of the taxa, so the complete split is given by this vector and the numbers of the taxa not included in it. The optional attribute "number" gives the number of times each partition was observed (e.g., in a bootstrap analysis; p. 171). Usually, in an object of class "prop.part", the first split includes all taxa and so is of the same length than the attribute "label".

The package phangorn uses the class "splits" which is derived from the previous one. In addition to the above elements, it includes an attribute named "weights" which is a numeric vector typically used to code the length of the edge defining each split. Another difference is that, most of the times, the trivial splits implied by terminal branches (e.g., A|BCD) are included in objects of class "splits" while they are not in those of class "prop.part".

The function designSplits in phangorn uses another strategy to code splits: the possible splits (there are $2^{n-1}-1$ for n taxa) are numbered sequentially. For instance, for $n = 4$, we display all possible splits (actually, like for "prop.part" only one of the two subsets):

```
> designSplits(4)$Splits
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    1    1    0    0
[4,]    0    0    1    0
[5,]    1    0    1    0
[6,]    0    1    1    0
[7,]    1    1    1    0
```

[1] Splits are traditionally denoted as, e.g., AB|CD for a a set of four taxa {A,B,C,D}.

You notice that the splits are coded in a similar way than integers are binary coded in computers, but from left to right instead of from right to left (in computers `0001` = 1, `0010` = 2, `0011` = 3, `0100` = 4, ...) So for the four taxa {A,B,C,D}, the split AD|BC will be numbered 6.

3.1.4 Molecular Sequences

Sequences of nucleotides (DNA) or amino acids (proteins) can be stored in R using character vectors or matrices and the conventional codes for these two types of molecules. However, this is quite memory-consuming and not optimal for computing. Several classes have been developed to store DNA sequences using less memory and allowing faster computation. Currently, no specific class has been proposed for protein sequences and they are stored as simple character data.

The Class `"DNAbin"` (ape)

Each nucleotide is coded specifically by a single byte arranged in a vector (single sequence), a matrix (sequences of the same length), or a list of vectors. Individual sequences may be identified with labels stored in `rownames` (matrix) or `names` (list). The coding of the individual bytes has been designed to allow fast comparison of two nucleotides; its principle is shown on Fig. 3.1 and the coding is detailed in Table 3.1.

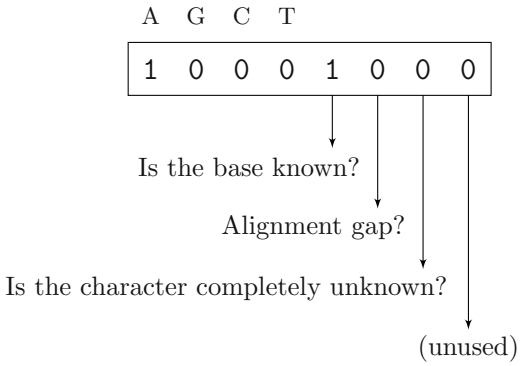

Fig. 3.1. Principles of the binary coding system used in `"DNAbin"`. Each bit of the byte gives information in the form of a Yes / No answer

The Class `"haplotype"` (pegas)

It is directly derived from the previous class, and is the set of unique sequences that can be extracted from an object of class `"DNAbin"` by the func-

Table 3.1. The bit-level coding scheme for nucleotides used in `"DNAbin"`

Nucleotide	IUPAC code	Bit-level code
A	A	10001000
G	G	01001000
C	C	00101000
T	T	00011000
A or G	R	11000000
A or C	M	10100000
A or T	W	10010000
G or C	S	01100000
G or T	K	01010000
C or T	Y	00110000
A or G or C	V	11100000
A or C or T	H	10110000
A or G or T	D	11010000
G or C or T	B	01110000
A or G or C or T	N	11110000
Alignement gap (–)		00000100
Unknown character (?)		00000010

tion `"haplotype"`. The attribute `"index"` identifies the individual sequences belonging to each haplotype.

The Class `"alignment"` (seqinr)

This class is designed to store a set of aligned sequences. The data are in a list with four elements:

`nb` the number of sequences.
`seq` the sequences as a vector of mode character where each sequence is a single string, and all strings have the same number of characters.
`nam` of vector of mode character giving the individual labels.
`com` optional comments (may be `NA`).

The Class `"phyDat"` (phangorn)

This class can be viewed as a "preparation" of sequence data for phylogeny estimation (Chapter 5). The sequences must be aligned, and only the patterns of variation among sequences are stored, corresponding to the unique columns in the class `"DNAbin"`. This class may be used for any discrete valued variable (DNA, protein, codon, morphological, ...) The data are stored as integers in a list with as many vectors as sequences. The following attributes give the necessary information:

`"weight"` the number of sites observed in each pattern.

`"index"` identifies to which pattern belongs the original sites.
`"levels"` a character vector giving the different possible states.
`"type"` one of `"DNA"`, `"AA"`, or `"USER"`.
`"contrast"` a matrix of integers giving the correspondance between the codes in the matrix (as rows) and the states given in `"levels"` (as columns).

If `type = "DNA"` or `"AA"`, the attribute `"levels"` will be set automatically. In the case of other kinds of data, a typical use of the function `phyDat` would be (with `x` being a matrix or a data frame):

```
phyDat(x, "USER", levels = unique(x))
```

If some levels have not been observed in `x`, this may be specified by changing `levels`.

3.1.5 Allelic Data

The Class `"loci"` (pegas)

This class is a direct extension of a data frame with an attribute named `"locicol"` identifying the columns that are loci; the other columns are additional variables that may be of any kind. The loci are coded with factors (p. 16) where the different levels are the observed genotypes with the alleles separated with a forward slash, for instance, 'A/A' for a classical genotype, or '132/148' for a microsatellite locus. The `rownames` are used as individual labels, and the `colnames` as labels for the loci and the other variables. Any level of ploidy can be handled.

The Class `"genind"` (adegenet)

In this class, allelic information is arranged in a numeric matrix where the rows are individuals and each column is an allele, so a locus may be represented by several columns. The entries of this matrix, named `tab`, are the relative frequencies of the allele for an individual: a homozygote has one value 1 and the others 0, whereas a heterozygote has two values 0.5 and the others 0. This is an S4 class, so the elements are accessed with the `@` operator (e.g., `x@tab`). Additional information are stored in other slots (`@ind.names`, `@pop`, ...) The details can be found with `class?genind`. This data structure is well-suited for multivariate analyses.

The Classes `"SNPbin"` and `"genlight"` (adegenet)

These two classes code for single nucleotide polymorphism (SNP) data for an individual or a set of genotypes, respectively. To optimize memory usage, each SNP is stored on a single bit, so one byte can store eight SNPs. Any level of ploidy is supported. These are S4 classes. There are a number of functions for their manipulation which are documented in `?SNPbin` and `?genlight` as well as a vignette accessible with `vignette("adegenet-genomics")`.

3.1.6 Phenotypic Data

Generally, the standard R data structures are sufficient to store phenotypic data. Continuous and discrete variables can be coded with numeric vectors and factors, respectively, and a set of such variables may be grouped in a matrix or, more generally, in a data frame. A list may also be used for repeated measurements with different numbers of repetitions for each species. The attributes `rownames` (or `names` for a vector, a factor, or a list) and `colnames` are used to identify the individuals and the variables.

Some standard R functions may be used to check the congruence of several series of `*names`. Most functions analyzing phenotypes and a phylogeny (e.g., `pic`) check the consistency of labels and reorder the phenotypic data if necessary. The rows of a data frame `X` may be reordered with respect to a phylogeny `tr`:

```
X <- X[tr$tip.label, ]
```

This will also drop the rows of `X` which are not in `tr`. The problem of managing and matching labels is discussed on page 62.

phylobase has two S4 classes: `"pdata"` for storing phenotypic data as a data frame with additional elements `type`, `comment` and `metadata`, and `"phylo4d"` to associate phenotypic data (as a data frame) with a phylogeny.

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.[2] 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 [192].

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")
```

[2] 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 `"multiPhylo"`, and is a list of objects of class `"phylo"` (see p. 55).

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`.

`read.tree` can read all kinds of Newick strings as long as they have labels, including star-like trees such as (a,b,c); or trees with a single tip such as (a); however, it cannot read trees only specified as a "skeleton" made of parentheses and commas usch as (((,(,)),),(,)).

All characters between straight brackets are ignored as these are considered as comments in the Newick syntax. If the Newick string is prefixed by some characters, these are read as tree names unless the option `tree.names` is specified.

3.2.2 Molecular Sequences

DNA sequences can be read with the `ape` function `read.dna` which reads files in FASTA, Clustal, interleaved, or sequential format (these formats are described in the help page of `read.dna`) returning the data as an object of class `"DNAbin"`. The function `read.nexus.data` reads DNA or protein sequence alignment in a NEXUS file, returning them as a list of character vectors. The function `read.aa` in `phangorn` reads protein sequences in FASTA, interleaved, or sequential format. This package has `read.phyDat` that calls either `read.dna` or `read.nexus.data` and returns the data as an object of class `"phyDat"`.

The function `read.GenBank` can read sequences 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"`.

seqinr has two functions which 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. A few other options allow the user to control the format of the returned object.

`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 object returned is of class `"alignment"`.

seqinr has an elaborate mechanism for retrieving sequences from molecular databanks. This works through the ACNUC repository.[3] 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"
 [4] "swissprot"       "ensembl"         "refseq"
 [7] "refseqViruses"   "nrsub"           "hobacnucl"
[10] "hobacprot"       "hovergendna"     "hovergen"
[13] "hovergen49dna"   "hovergen49"      "hogenom4"
[16] "hogenom4dna"     "hogenom"         "hogenomdna"
[19] "homolens"        "homolensdna"     "greviews"
[22] "polymorphix"     "emglib"          "taxobacgen"
[25] "apis"            "anopheles"       "caenorhabditis"
[28] "cionasavignyi"   "danio"           "drosophila"
[31] "felis"           "gallus"          "human"
[34] "mouse"           "saccharomyces"   "tetraodon"
[37] "xenopus"
```

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

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

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* [116]:

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

```
> query("rampho", "sp=Ramphocelus@")
> rampho
42 SQ for sp=Ramphocelus@
```

By default `query` uses the last opened database. If several databases are open at the same time, the option `socket = s$socket` may be used here. This command returns an object named `"rampho"` which lists the sequences meeting the selection criteria.[4] 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 42 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]]
      name     length      frame     ncbicg
"AF310048"      "921"        "0"        "2"
```

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

```
> x <- getSequence(rampho)
> length(x)
[1] 42
> length(x[[1]])
[1] 921
> x[[1]][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 first 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[[1]], as.character(y[[1]]))
[1] TRUE
```

seqinr comes with a very extensive (and entertaining) manual and numerous example data files. We'll see again some of its functionalities later in this book.

The package BoSSA can read PDB (protein data bank) files with the function `read.PDB`. The PDB file format is a standard to store three dimensional molecular structures, often obtained by crystallography. As an example, the

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

file 'bdna.pdb' stores the 3-D structure of a 12-base pair molecule of B-DNA (this file is included in the supplementary material of the book):

```
> bdna <- read.PDB("bdna.pdb")
> names(bdna)
[1] "header"   "compound" "atom"     "sequence"
> bdna$header
[1] "B-DNA
```

The element named `sequence` is a list with the sequence(s):

```
> bdna$sequence
$ref_A
 [1]  1  2  3  4  5  6  7  8  9 10 11 12

$chain_A
 [1] "  G" "  C" "  G" "  A" "  A" "  A" "  T" "  T" "  T"
[10] "  G" "  C" "  G"

$ref_B
 [1]  1  2  3  4  5  6  7  8  9 10 11 12

$chain_B
 [1] "  C" "  G" "  C" "  A" "  A" "  A" "  T" "  T" "  T"
[10] "  C" "  G" "  C"
```

The element `atom` stores the 3-D structure in a data frame with seven columns:

```
> names(bdna$atom)
[1] "atom"  "aa"    "chain" "naa"   "X"     "Y"     "Z"
```

The last three columns contain the atomic spatial coordinates. A nice representation can be done with the rgl package (Fig. 3.2):

```
> library(rgl)
> points3d(bdna$atom$X, bdna$atom$Y, bdna$atom$Z)
```

3.2.3 Allelic Data

It is common to store allelic data in a standard tabular form with individuals as rows and genes (or loci) as columns, each entry giving the corresponding genotype. pegas has the function `read.loci` to read such files. It is modeled after the standard R function `read.table` and thus has the same flexibility. It assumed that alleles are separated by a forwardslash but this can be changed with the option `allele.sep`.

However, several computer programs widely used in population genetics have their own data file format. Fortunately, adegenet provides four functions, `read.fstat`, `read.genetix`, `read.genepop`, and `read.structure`, to

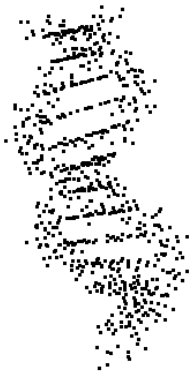

Fig. 3.2. Three dimensional structure of a B-DNA molecule plotted with rgl. Once plotted in R, the image may be rotated and zoomed in / out easily

read files from the programs of the same name. pegas includes a detailed guide explaining step-by-step how to proceed to read all the allelic data files, including some tips about reading Excel files. It can be open from R with:

```
vignette("ReadingFiles")
```

3.2.4 Reading Data Over the Internet

There has been a tremendeous increase in the number of databases and data sets available on the Internet in the past ten years. This may take various forms, such as formal databases with sophisticated browsable interfaces, or simple data files accompanying a publication. With the requirement from most evolutionary journals to deposit data related to publication in data archives [33], the trend will surely continue for some time.

Some issues arise with the widespread availability of numerous data sets on the Internet:

1. How to localize a particular data set?
2. How to read a data set for analysis?
3. How data sets from different sources can be combined in an analysis?
4. How to assess the quality of the data with, for instance, its metadata?

The aim of this section is to show some solutions to the second and third issues. The first and fourth ones are important but not obvious to treat in a systematic way.

Most R functions used to input data (e.g., `scan`, `read.table`, ...) can read data from files located on the Internet: the `file` argument is then the complete URL of the file such as `"http://..."` or `"ftp://..."`.[5] As an example, we

[5] URLs with https are not supported by R; see `?connections`.

read the data file compiled by Ernest [71] containing values of life-history traits of non-flying placental mammals (i.e., excluding egg-laying mammals, marsupials, and bats). For clarity, we first store the URL in a string:[6]

```
> a <- "http://www.esapubs.org/archive/ecol/E084/093/"
> b <- "Mammal_lifehistories_v2.txt"
> ref <- paste(a, b, sep = "")
> X <- read.delim(ref)
> names(X)
 [1] "order"         "family"        "Genus"
 [4] "species"       "mass.g."       "gestation.mo."
 [7] "newborn.g."    "weaning.mo."   "wean.mass.g."
[10] "AFR.mo."       "max..life.mo." "litter.size"
[13] "litters.year"  "refs"
```

Once the data have been read into R, they can be manipulated as usual, for instance, to make a character vector with the genus and species names in a standard way which may be used to cross this data frame with a tree (see Section 3.6):

```
> paste(X$Genus, X$species, sep = "_")[1:5]
[1] "Antilocapra_americana" "Addax_nasomaculatus"
[3] "Aepyceros_melampus"    "Alcelaphus_buselaphus"
[5] "Ammodorcas_clarkei"
```

Fortunately, the input functions in other packages have the possibility to read remote files. For instance, it is possible to read PDB files from the Research Collaboratory for Structural Bioinformatics (RCSB) web site, for instance to read an activation intermediate of cathepsin E "1TZS":

```
> library(BoSSA)
> tzs <- read.PDB("http://www.rcsb.org/pdb/files/1TZS.pdb")
> tzs$sequence$chain_P
[1] "GLY" "SER" "LEU" "HIS" "ARG" "VAL" "PRO" "LEU" "ARG"
```

The 'pdb/files/' directory on this site stores the PDB files, so it's only needed to change the part between this and the '.pdb' suffix, for instance, to read the structure of CrgA from *Neisseria meningitidis* "3HHF":

```
> hhf <- read.PDB("http://www.rcsb.org/pdb/files/3HHF.pdb")
```

This makes possible to write a function to read any of these PDB files where the reference number as argument:

[6] For the sake of keeping the command lines outside the margins, long strings are built in two steps. This is not needed, but the user must be aware that a linebreak inserted in a string is read as it is.

```
read.rcsb <- function(ref)
{
    url <- paste("http://www.rcsb.org/pdb/files/",
                 ref, ".pdb", sep = "")
    read.PDB(url)
}
```

Pfam is another large database of proteins located on the Sanger Institute web site [84]. This site allows one to download various information for thousands of protein families, including sequence alignment and trees. Here is an example reading the chitin binding peritrophin-A "metagenomic" tree:

```
> a <- "http://pfam.sanger.ac.uk/family/tree/"
> b <- "download?alnType=meta&acc=PF01607"
> ref <- paste(a, b, sep = "")
> tr <- read.tree(ref)
> tr
Phylogenetic tree with 6 tips and 4 internal nodes.

Tip labels:
[1] "EBG17252.1/78-131"  "EBB56650.1/58-112"
[3] "EBG17252.1/134-186" "ECX18707.1/2-42"
[5] "ECR19709.1/127-178" "ECR19709.1/65-116"
Node labels:
[1] ""      "0.600" "0.670" "0.860"

Unrooted; includes branch lengths.
```

Here, the 'alnType=meta' specifies that the metagenomic tree is read. The option 'alnType=' can take three other values: 'seed', 'full', or 'ncbi'. The bit of text 'acc=PF01607' specifies the reference number of the family.

These brief examples show only the flavor of the potentialities of reading data over the Internet. The Exercices invite the reader to explore further these ideas.

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 8).

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 one or several trees in Newick format. It takes as main argument a `"phylo"` or `"multiPhylo"` 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);"
```

If a list of trees is used, then a vector with several Newick strings is returned. If the list has names they are not written in the file unless the option `tree.names = TRUE` is used, or a vector of names is given here. To save the tree(s) 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.

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 [192]). If several trees are written, they must have the same tip labels, and must be given either as seperated by commas, or as a list:

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

If the list of trees has names `write.nexus` writes them in the file.

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). The function `write.nexus.data` writes DNA sequences in a NEXUS file.

Splits can be written in a NEXUS file with `write.nexus.splits` in phangorn:

```
> write.nexus.splits(as.splits(tr))
#NEXUS

[Splits block for Spectronet]
[generated by phangorn:
Schliep K (2011). "phangorn: phylogenetic analysis in R."
_Bioinformatics_, *27*(4), pp. 592-593. ]

BEGIN TAXA;
DIMENSIONS NTAX=2;
TAXLABELS a b ;
END;

BEGIN ST_SPLITS;
DIMENSIONS NSPLITS=3;
FORMAT LABELS WEIGHTS;
```

```
MATRIX
 1 1 1 ,
 2 1 2 ,
 3 0 1 2 ,
;
END;
```

Allelic data are written with `write.loci` from `pegas`: this function calls `write.table`, so the data are written in a tabular way and the formatting is controlled by several specific options (`loci.sep`, `allele.sep`) as well as the usual options of `write.table`.

3.4 Manipulating Data

This section includes materials on phylogenetic data manipulation in R ranging from basic to elaborate procedures. This will interest most users. Most of this section is devoted to the manipulation of trees since the manipulation of molecular, allelic, or phenotypic is usually done using basic R's tools.

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` (Fig. 3.3).

```
> 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, 3:4, trim.internal = FALSE))
[1] "((a:1,b:1):1,NA:1);"
```

It is often convenient to identify tips with numbers, but you must be very careful that many operations are likely to change these numbers (essentially because they must be numbered without gaps). In all situations, it is safest to identify tips with their labels.

This function has three additional options to control how the branches are removed: `subtree`, `root.edge`, and `rooted` (a logical value to force to consider the tree as (un)rooted.

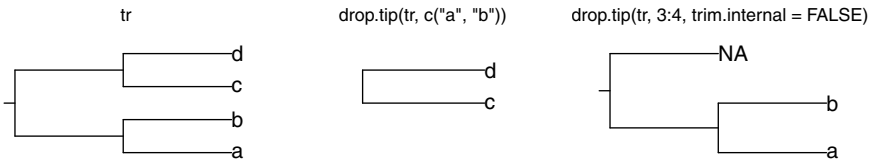

Fig. 3.3. Examples of the use of `drop.tip`

`extract.clade` does the opposite operation than the previous function: it keeps all the tips and nodes descending from a node specified by its number or its label.

```
> write.tree(extract.clade(tr, node = 6))
[1] "(a:1,b:1);"
```

`drop.fossil` is similar to `drop.tip` but dropping only the tips that are not contemporaneous (as measured by the distances on the tree) with the most distant ones from the root.

`bind.tree` is used to build a tree from two trees where the second one is grafted on the first one. 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 `where`. 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 `position` to specify where on the branch the tree is to be bound (Fig. 3.4).

```
> t1 <- read.tree(text = "(a:1,b:1):0.5;")
> t2 <- read.tree(text = "(c:1,d:1):0.5;")
> write.tree(bind.tree(t1, t2))
[1] "(a:1,b:1,(c:1,d:1):0.5):0.5;"
> write.tree(bind.tree(t1, t2, position = 1))
[1] "((a:1,b:1):0.5,(c:1,d:1):0.5):0;"
```

The operator `+` applied to two trees is a short-cut for this last example, eventually adjusting to the root edge length of the first tree:

```
> write.tree(t1 + t2)
[1] "((a:1,b:1):0.5,(c:1,d:1):0.5):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 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.

```
> write.tree(rotate(t1, 3))
[1] "(b:1,a:1):0.5;"
```

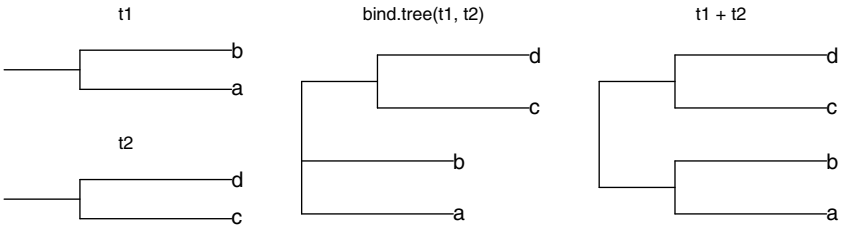

Fig. 3.4. Examples of the use of `bind.tree`

For non-binary nodes, the option `polytom` specifies which of the two clades to swap.

`ladderize` rotates the internal branches of a tree in order to arrange the sister-clades so that the smallest one is on the right-hand side when plotted upwards. The option `right = FALSE` ladderizes the tree in the other direction.

`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 [97, 113]: `tr <- compute.brlen(tr, 1)`. This is likely to be useful in comparative analyses when only a phylogeny with no branch lengths is available.

`compute.brtime` has a similar action than the previous function but the returned tree is ultrametric; the argument `method` may be:

- A character string specifying the method to be used to generate the branching times (e.g., `"coalescent"` which is the default).
- A numeric vector of the same length than the number of nodes.

The option `force.positive` allows the user to avoid returning a tree with negative branch lengths.

3.4.2 Rooted *Versus* Unrooted Trees

The Newick parenthetic format can represent both rooted and unrooted trees. In the latter, all nodes are made by the connection of at least three 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.

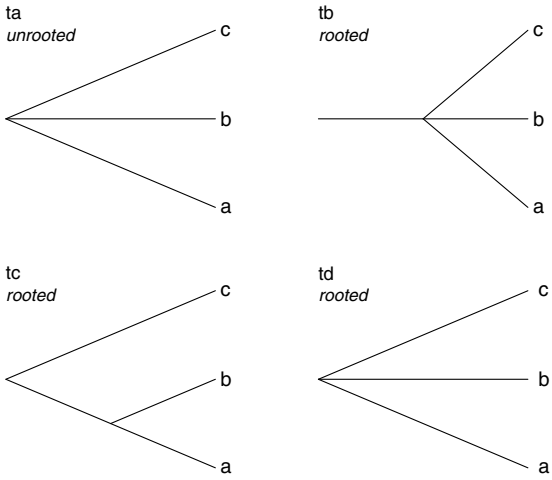

Fig. 3.5. The four trees used in the text to illustrate the difference between rooted and unrooted trees

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 (Fig. 3.5).

The function `root` (re)roots 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 may not be the same as the original one. The specified outgroup must be monophyletic, otherwise the operation fails and an error message is printed. The returned tree will have a multichotomous root; the option `resolve.root = TRUE` forces the root to be dichotomous be adding a zero-length branch below the ingroup.

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

The function `midpoint` in `phangorn` does a midpoint rooting by placing the root so to minimize the distances from this node to all tips.

3.4.3 Graphical Tree Manipulation

It is fundamental to allow tree manipulation from the command line because in many situations the same operation must be done on several trees and this is practically done via a script or some command lines. But in exploratory analyses, it may be a bit tedious to pass the correct arguments to these functions, and the user may prefer to interact more closely with the graphical display of the tree. The functions `drop.tip`, `extract.clade`, `bind.tree`, and `root` have an option `interactive = FALSE` which, when switched to `TRUE`, invites the user to click on the plotted tree to identify the argument `tip` or `node`. The user must take care that the correct tree has been plotted, as well as assigning the returned tree in a possibly different object.

It is possible to write sophisticated, though very short, scripts that enhance the degree of interactivity. For instance, the next two lines of command plot the tree `tr`, call `root` interactively, save the result in `ta`, and plot the latter.

```
> plot(tr)
> plot(ta <- root(tr, interactive = TRUE))
Click close to a node of the tree...
You have set resolve.root = FALSE
> # back to the prompt after clicking
```

This may be tried with, say, `tr <- rtree(5)`.

The power of these interactive functions may be increased by combining them with the function `identify.phylo` which is introduced in the next chapter (p. 98).

3.4.4 Dichotomous *Versus* Multichotomous Trees

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

$$(\ldots,(\mathrm{A},\mathrm{B},\mathrm{C}),\ldots);$$

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

```
...
20 1
20 2
20 3
...
```

where 1, 2, 3 would be the numbers of the tips A, B, C, and 20 the number of their ancestor.

As shown in the next chapters, some methods deal only with dichotomous (i.e., binary) trees, thus it is 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 the number of labelled rooted topologies with n tips (`howmanytrees(n)` in R). For only three possibilities with $n = 3$, there are 105 with $n = 5$, and 34,459,425 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 it is required to conserve the same topology in all repetitions.

apTreeshape has a different mechanism to solve multichotomies randomly. It is used when converting trees of class `"phylo"` with multichotomies (Section 3.5). The function `as.treeshape` has the option `model` that can take the following values: `"biased"`, `"pda"`, `"aldous"`, or `"yule"`. This specifies the model used to resolve the multichotomies. These models are explained in Section 7.1.

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.

3.4.5 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 or as specified by the user.

`balance` returns, for a fully binary tree, the number of descendants of both sister-lineages from each node (see Section 6.3.7 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 numeric vectors the comparison is done considering the numerical precision of the computer. For `"phylo"` objects, two options control the way the two trees are compared: `use.edge.length` and `use.tip.label`, both being `TRUE` by default. If `use.edge.length = FALSE`, only the labeled topologies are compared; if both options are set to `FALSE`, then the unlabeled topologies are compared.

```
> 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
> t3 <- read.tree(text = "(c:1,(a:1,b:1):1);")
> all.equal(t1, t3)
[1] FALSE
> all.equal(t1, t3, use.edge.length = FALSE)
[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:

```
isTRUE(all.equal(t1, t2))
```

A graphical comparison is possible with the functions `compare.phylo` (in the package phyloch) or `phylo.diff` (in distory). The first one is especially useful when comparing two ultrametric trees because it performs a comparison of the branching times between the two trees; a graphical summary is also done (Fig. 3.6).

```
> ta <- read.tree(text = "(((a:1,b:1):1,c:2):1,d:3);")
> tb <- read.tree(text = "(((a:0.5,b:0.5):1.5,d:2):1,c:3);")
> compare.phylo(ta, tb, presCol = "white", absCol = "black",
```

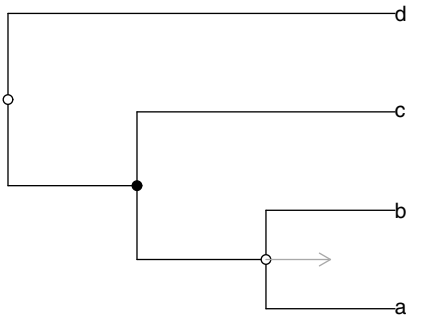

Fig. 3.6. Comparison of two trees with `compare.phylo`

```
+               tsCol="darkgrey", font = 1)

 0 tips from x are missing in y
 0 tips from y are missing in x

Summary of the time shift of internal nodes:
 max: 0
 min: -0.5
 median: -0.25
 mean: -0.25
 standard diviation: 0.3535534

....

$shared_nodes
          5    7
xnodes    5  7.0
ynodes    5  7.0
timeshift 0 -0.5
```

`phylo.diff` performs a general comparison of the topologies of two phylogenies, plotting them side by side and highlighting their differences (the colors cannot be changed in the current version). It relies on the function `distinct.edges` in distory:

```
> distinct.edges(ta, tb)
[1] 1
> distinct.edges(ta, ta)
NULL
```

3.4.6 Manipulating Lists of Trees

Lists of trees of class "multiPhylo" are manipulated in R's standard way. These lists can actually be seen as vectors of trees. The operators [, [[, $ are used in the usual way, and this works both for extracting tree(s) from a list and for assigning some to a list.

The objects of class "multiPhylo" can exist in two versions: either as a simple list of trees, or, if all trees have the same tip labels and none of them are duplicated, the list has an attribute "TipLabel" and the tip numbers in the matrix edge are the same in all trees. For instance, we can simulate a list of ten trees:

```
> TR <- rmtree(10, 5)
> attr(TR, "TipLabel")
NULL
```

TR may be 'compressed' with .compressTipLabel:

```
> TRcompr <- .compressTipLabel(TR)
> attr(TRcompr, "TipLabel")
[1] "t1" "t4" "t3" "t5" "t2"
```

This results in a compact and efficient way to store lists of trees. The function .uncompressTipLabel does the reverse operation. Here, TR and TRcompr contain the same information, but if we try to change one tree in TRcompr, the tip labels of the new tree will be checked and the operation will work only if they are the same than in the list (the tip numbers will eventually be renumbered):

```
> TR[[1]] <- rtree(10)
> TRcompr[[1]] <- rtree(10)
Error in `[[<-.multiPhylo`(`*tmp*`, 1, value = ....
  tree with different number of tips than those in the list
  (which all have the same labels; maybe you want
   to uncompress them)
```

Because the tip labels are not compressed in TR, this list can accept any tree, but after the above operation, it cannot be compressed:

```
> .compressTipLabel(TR)
Error in .compressTipLabel(TR) :
  tree no. 2 has a different number of tips
```

Checking tip labels is also done when adding new trees to the list:

```
> TRcompr[11:20] <- rmtree(10, 10)
Error in `[[<-.multiPhylo`(`*tmp*`, 1, value = ....
  tree with different number of tips than those in the list
```

```
  (which all have the same labels; maybe you want
   to uncompress them)
> TRcompr[11:20] <- rmtree(10, 5)
```

Table 3.2 recapitulates the use of the operators for lists of trees.

Table 3.2. Suppose `TR` is a list of class `"multiPhylo"` with three trees. Objects in uppercase are of class `"multiPhylo"`; those in lowercase are of class `"phylo"`. All replacement operations check the tip labels of the right-hand term if the labels of `TR` are compressed

Example	Comment
`tr <- TR[[1]]`	extract one tree
`tr <- TR$A`	id.
`tr <- TR["A"]`	id.
`TB <- TR[1]`	id. but as a list
`TB <- TR[1:2]`	extract two trees
`TR[[1]] <- tr`	replace one tree
`TR[1] <- list(tr)` *or* `c(tr)`	id.
`TR[1:2] <- c(tr, ts)`	replace two trees
`TR[[4]] <- tr`	appends a tree to the list
`TR[5:6] <- c(tr, ts)`	appends two trees

3.4.7 Molecular Sequences

DNA sequences read with `read.dna` are stored in R as a list or a matrix (if aligned) where each element represents a nucleotide. This allows manipulation of DNA data with R's standard tools. Typically, the labels read in the file will be used as names if the data are stored in a list, or rownames if in a matrix. The operators `[`, `[[`, and `$` can be used in the usual way. A useful application is when selecting sites with respect to codon positions in a coding sequence. For instance, for a matrix `x`, and assuming that the reading frame of the alignment is correct, the following will select the third, the sixth, the ninth, ... columns:

```
y <- x[, c(FALSE, FALSE, TRUE)]
```

This will work whatever the number of columns because logical indices are recycled (p. 13). You must be careful to not forget the comma so that indexing will be applied on the columns only. To do the complementary operation of selecting the first and second codon positions, one can either use a logical vector with the first two values set to `TRUE`, or, more simply, invert the values of the above vector with the `!` operator; in that case it is more convenient to prepare the logical vector beforehand:

```
s <- c(FALSE, FALSE, TRUE)
y <- x[, s]
z <- x[, !s]
```

Additionally, `rbind` and `cbind` can be used to combine matrices rowwise or columnwise. The latter function has an option `fill.with.gaps` to possibly create 'supermatrices' where missing data are filled with alignment gaps, as well as the option `check.labels`.

Sequences stored in a list can be converted into a matrix with `as.matrix` if they have the same length: the names and other attributes will be set appropriately. However, many functions in `ape` (e.g., `dist.dna`) accept lists of sequences and test whether they all have the same length. The reverse operation, from a matrix to a list, can be done with `del.gaps`: this will remove the eventual alignment gaps. If the latter must be kept, `as.list` can be used instead.

`seqinr` has some sophisticated functions for manipulating molecular sequences. `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 with names `"a"`, `"c"`, `"g"`, and `"t"`. The option `freq = TRUE` allows the user to return the counts instead of the proportions. The option `all = TRUE` results in counting all 17 possible values (Table 3.1).
- `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.
- `Ftab` computes the contingency table of base frequencies from a pair of sequences.

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 amino acids. For instance, taking a protein sequence distributed with `seqinr` (Fig. 3.7):

```
> 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
....
```

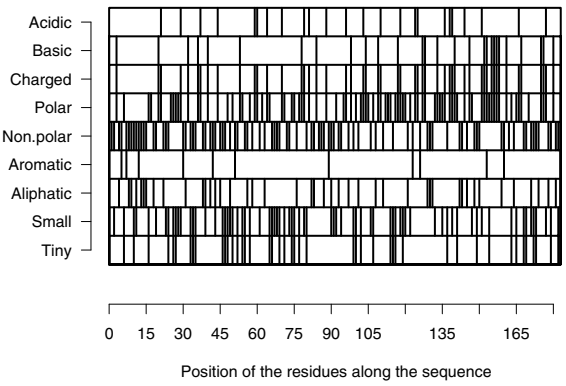

Fig. 3.7. Plot of the distribution of amino acid categories along the sequence of a protein

3.4.8 Allelic Data

Objects of class `"loci"` can be manipulated easily since they are a direct extension of data frames. `pegas` has a few method functions adapted to this class: `rbind`, `cbind`, and the indexing operator `[`. Some other functions, such as `subset`, `split`, `rownames`, `colnames`, or the `$` operator, can be used without problem since they respect additional attributes. Additionally, `pegas` allows to edit a `"loci"` object with `edit` or `fix`. The command `edit(x)` opens the data in R's spreadsheet-like data editor, and returns the possibly modified object. On the other hand, `fix(x)` modifies directly the object `x`.

`adegenet` allows to edit an object of class `"genind"`, but since this is an S4 class, the elements are accessed with the `@` operator. The help page `?genind` describes them in details. A few functions are provided to ease the manipulation of `"genind"` objects because setting all elements by hand may be tedious: `seploc` splits the data with respect to each locus, `seppop` does the same with respect to each population, and `repool` allows to do the opposite operation.

It is also possible to select a part of a `"genind"` object with the usual indexing operator `[`, the indices will apply to the rows and/or columns of the `@tab` slot, and the other slots will be modified accordingly. Some care should be paid when using numerical indexing on columns because something like `x[, 1]` will only select one allele column, eventually excluding other alleles of the same locus. It is safer to use the option `loc` which specifies the loci to be selected, and so will select the appropriate allele columns. Further details are available in `?pop` as well as information on some other functions.

3.5 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 or analyses can be done on some classes but not others. Table 3.3 gives details on how to convert among the classes discussed here. Only direct possible conversions are indicated; others 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.

Table 3.3. Conversion among the different classes of tree objects in R (`x` is the object of the original class)

From	To	Command	Package
phylo	phylo4	`as(x, "phylo4")`	phylobase
	matching	`as.matching(x)`	ape
	treeshape	`as.treeshape(x)`	apTreeshape
	hclust	`as.hclust(x)`	ape
	prop.part	`prop.part(x)`	ape
	splits	`as.splits(x)`	phangorn
	evonet	`evonet(x, from, to)`	ape
	network	`as.network(x)`	ape
	igraph	`as.igraph(x)`	ape
phylo4	phylo	`as(x, "phylo")`	phylobase
matching	phylo	`as.phylo(x)`	ape
treeshape	phylo	`as.phylo(x)`	apTreeshape
splits	phylo	`as.phylo(x)`	phangorn
	networx	`as.networx(x)`	phangorn
evonet	phylo	`as.phylo(x)`	ape
	networx	`as.networx(x)`	ape
	network	`as.network(x)`	ape
	igraph	`as.igraph(x)`	ape
haploNet	network	`as.network(x)`	pegas
	igraph	`as.igraph(x)`	pegas
hclust	phylo	`as.phylo(x)`	ape
	dendrogram	`as.dendrogram(x)`	stats

Table 3.4 gives the same information for genetic and phenotypic data objects.

Table 3.4. Conversion among the different classes of DNA, allelic, and standard objects in R (`x` is the object of the original class)

From	To	Command	Package
DNAbin	character	`as.character(x)`	ape
	alignment	`as.alignment(x)`	ape
	phyDat	`as.phyDat(x)`	phangorn
	genind	`DNAbin2genind(x)`	adegenet
character	DNAbin	`as.DNAbin(x)`	ape
	loci	`as.loci(x)`	pegas
alignment	DNAbin	`as.DNAbin(x)`	ape
	phyDat	`as.phyDat(x)`	phangorn
	character	`as.matrix(x)`	seqinr
	genind	`alignment2genind(x)`	adegenet
phyDat	DNAbin	`as.DNAbin(x)`	phangorn
	character	`as.character(x)`	phangorn
loci	genind	`loci2genind(x)`	pegas
	data frame	`class(x) <- "data.frame"`	
genind	loci	`genind2loci(x)`	pegas
data frame	phyDat	`as.phyDat(x)`	phangorn
	loci	`as.loci(x)`	pegas
	genind	`df2genind(x)`	adegenet
matrix	phyDat	`as.phyDat(x)`	phangorn

3.6 Managing Labels and Linking Data Sets

There are two ways to link distinct data objects in R:

- Observations are ordered similarly in the different objects.
- Observations are identified by labels.

The first way is efficient because the data are easily located. The second way is safer when putting different data sets together, but it is not so straightforward because labels may be duplicated, or have typing errors, or come from computers with different locales of encoding systems (e.g., geographic localities with accented characters, or even encoded in non-Roman characters).

Often, a user handles two sets of labels for the same object because one set may refer to the labels used in, say, a database, or the user wants to have at the same time a full description of the data as well as a shortened version for plotting. Three functions in `ape` help to modify labels. `makeLabel` modifies labels to shorten them, make them unique, delete some characters considered as "illegal", and possibly quote them. The options are:

```
makeLabel(x, len = 99, space = "_", make.unique = TRUE,
          illegal = "():;,[]", quote = FALSE, ...)
```

It is a generic function. The options above are those when `x` is a character vector. The options when `x` is of class `"phylo"` are:

```
makeLabel(x, tips = TRUE, nodes = TRUE, ...)
```

The '...' permits to use the same options than above; the two other options specify whether to apply the operation on the tip and / or node labels.

When creating several sets of labels for the same observations, it is better to keep them together in a data.frame, for instance, if we wish to use external programs with our data. PhyML accepts labels up to 99 character long, Clustal up to 30, and Phylip up to 10. Suppose we have an alignment of DNA sequences `X`, we could do:

```
LABELS <- data.frame(original = rownames(X),
                     phyml = makeLabel(rownames(X)),
                     clustal = makeLabel(rownames(X), 30),
                     phylip = makeLabel(rownames(X), 10),
                     stringsAsFactors = FALSE)
```

The data are prepared and written on the disk for an analysis with PhyML, such as:

```
rownames(X) <- LABELS$phyml
write.dna(X, "X.txt")
```

After estimating a tree with PhyML and reading it into R (say `tr`), we want to replace its labels with the original ones:

```
o <- match(tr$tip.label, LABELS$phyml)
tr$tip.label <- LABELS$original[o]
rownames(X) <- LABELS$original
```

`tr` now has the same labels than `X`. This mechanism based on the base R function `match` may be used to link any type of objects were observations are identified with labels, names, or rownames.

Labels in R are virtually illimited in size but long labels often need to be formatted when plotted. `mixedFontLabel` helps for this task:

```
mixedFontLabel(..., sep = " ", italic = 0, bold = 0,
               parenthesis = 0,
               always.roman = c("sp.", "spp.", "ssp."))
```

Here '...' is not for passing arguments among functions but because various numbers of vectors may be passed. These vectors are then paste together separated with the string(s) given in `sep`. The three next options specify the formatting of each element—note than an element may be in bold and italics if its number is passed to both options. The last option specifies that some strings are never printed in italics even if the vector they belong to is in `italic`. This function returns a vector of R expressions which is not readable to the human eye, but if in the `tip.label` element of a `"phylo"` object, `plot.phylo` will interpret it correctly. For instance:

```
tr$tip.label <- mixedFontLabel(tr$tip.label, geo, paren = 2)
plot(tr)
```

The last of these three functions is `makeNodeLabel`: it creates labels for the noded of a `"phylo"` tree. The second argument, `method`, gives the method to create these labels. This is simple for `"number"` (the default) and `"md5sum"`; the third possibility, `"user"`, though more complicated to use, is potentially powerful. Suppose `tr` is a large tree with all species of mammals and we want to label the clade of all primate species "Primates" and the one of all rodent species "Rodentia" on the appropriate nodes. Suppose also that we have two character vectors listing all species of these two orders (`primates` and `rodents`). Then the following command will create the labels:

```
tr <- makeNodeLabel(tr, "user",
    nodeList = list(Primates = primates, Rodentia = rodents))
```

The mechanism used inside `makeNodeLabel` is based on regular expressions, so even a list of genera is enough. Previous node labels are not erased, it is thus possible to use similar commands repeatedly. The regular expressions make possible to identify clades with other bits of text present in the labels of the tips (geographic locations, strains, ...) Appendix A gives a summary of the use of regular expressions.

The option `method = "md5sum"` creates node labels that are unique for a given set of tip labels. This is done by writting into a temporary file the labels of tips descendant from each node (after sorting them alphabetically), then the md5sum of this file is extracted and used as label. This results in a 32-character string which is unique—even accross trees—for a given set of tip labels. This is useful for crossing information among trees, and is used in the function `node.trans` in `phyloch`.

3.7 Sequence Alignment

Sequence alignment is an essential step in evolutionary analyses of molecular sequences because it identifies homologous sites in spite of insertions and deletions (indels) that blurs this identification. A good introduction and summary of this subject are provided by Higgins and Lemey [134] as well as a listing of computer programs to perform multiple sequence alignment with different methods. These authors stress that two rules should be observed:

- Protein-coding sequences of nucleotides must first be aligned at the amino acid level.
- At least two computer programs should be used to check the robustness of the alignment.

Methods of sequence alignment vary greatly but they are mainly based on scoring the dissimilarity between sequences and balancing this scoring against

gap opening in the sequences. Clustal X [299] is the most widely used program of sequence alignment and is available for most operating systems.[7] There are also several interfaces to the Clustal computing engine, such as the Web-interface ClustalWWW [40].

Programs for sequence alignment are often large applications that are costly to re-program *de novo* in a new language such as R. On the other hand, most of them can be called with a CLI which makes them relatively simple to interface with R using system calls. This is what is implemented in several functions of phyloch. The programs currently available are the most recommended by Higgins and Lemey [134]: Clustal, Mafft [156],[8] Muscle [60],[9] T-Coffee [220],[10], as well as Prank[11] not listed by Higgins and Lemey. All these programs but the last one are also distributed in various GNU/Linux distributions. The interfaces of `mafft` and `prank` are:

```
mafft(x, method = "localpair", maxiterate = 1000, op = 1.53,
      ep = 0.123)
prank(x, outfile, guidetree = NULL, gaprate = 0.025,
      gapext = 0.75)
```

where `x` is an object of class `"DNAbin"` and the other options control the alignment procedure. Both functions return an object of this class.

Similar functions can be found in `ape` for Clustal, Muscle, and T-Coffee:

```
clustal(x, pw.gapopen = 10, pw.gapext = 0.1,
        gapopen = 10, gapext = 0.2, exec = NULL,
        MoreArgs = "", quiet = TRUE)
muscle(x, exec = "muscle", MoreArgs = "", quiet = TRUE)
tcoffee(x, exec = "t_coffee", MoreArgs = "", quiet = TRUE)
```

Once an alignment has been performed, it is recommended to check its quality by visual inspection. Clustal X includes a graphical display to this end. Furthermore, `ape` has a method `image` for `"DNAbin"` objects which allows one to plot a nucleotide alignment in a flexible way. The advantages compared to a classical graphical display are that the whole alignment can be visualized globally, and it is possible to plot a selected base (or several). The selection of colors is straightforward so that it can be adapted to color-blind users. Furthermore, I found in my experience that even with well-contrasted colors, the simultaneous plot of the four bases leads to an overload of information. The possibility to plot a single base (e.g., with `image(x, "a")`) helps to better visualize the portions of the alignment rich or poor in this base. The distribution of alignement gaps is also easily displayed with `image(x, "-")`.

[7] http://www-igbmc.u-strasbg.fr/BioInfo/ClustalX/
[8] http://align.bmr.kyushu-u.ac.jp/mafft/software/
[9] http://www.drive5.com/muscle/
[10] http://www.tcoffee.org/
[11] http://www.ebi.ac.uk/goldman-srv/prank/prank/

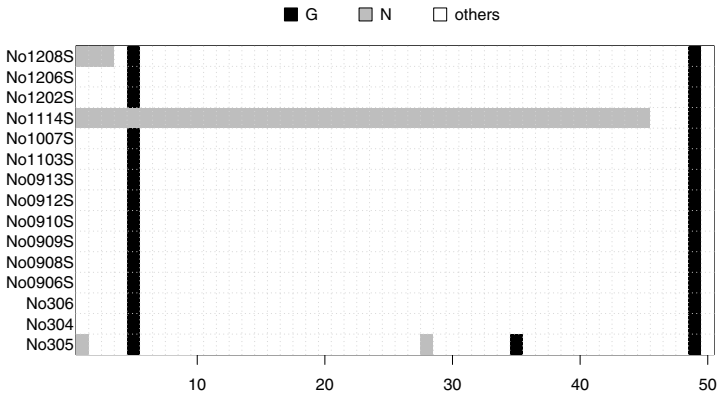

Fig. 3.8. Plot of the fifty first sites of the woodmouse data displaying the guanine and the missing data

Thanks to the efficient subsetting of `"DNAbin"` objects (p. 56), it is easy to zoom on a portion of an alignment. Figure 3.8 shows the first 50 nucleotides of the woodmouse data displaying the guanine together with the missing data. A grid is added to ease reading the graph:

```
x <- woodmouse[, 1:50]
image(x, c("g", "n"), c("black", "grey"))
grid(ncol(x), nrow(x), col = "lightgrey")
```

Finally, the function `del.gaps` in `ape` is useful here because it removes all inserted gaps (`-`) and returns the sequences as a list.

3.8 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.

3.8.1 *Sylvia* Warblers

Böhning-Gaese et al. [27] 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 using 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 printing the returned object:

```
> sylvia.seq
25 DNA sequences in binary format stored in a list.

Mean sequence length: 1134.84
   Shortest sequence: 1041
    Longest sequence: 1143

Labels: Z73494 AJ534526 AJ534527 AJ534528 AJ534529 AJ534530...

Base composition:
    a     c     g     t
0.270 0.354 0.131 0.245
```

We have effectively a list with 25 sequences: 23 of them have 1143 nucleotides, and 2 have 1041. This necessitates an alignment operation which is done with Clustal:

```
> sylvia.clus <- clustal(sylvia.seq)
```

The alignment operation shows that there are 102 missing nucleotides in the last two sequences (which could be easily visualized with `image(sylvia.clus)`). To check the stability of the alignment produced by Clustal, we perform the same operation with MAFFT:

```
> library(phyloch)
> sylvia.maff <- mafft(sylvia.seq)
blosum 62
generating 200PAM scoring matrix for nucleotides ... done
scoremtx = -1
....
```

Because the external programs will likely reorder the sequences, we compare their outputs after reordering the rows:

```
> identical(sylvia.clus[x, ], sylvia.maff[x, ])
[1] TRUE
```

Note that we kept the original (unaligned) sequences from GenBank because they have the species names. To save some memory, we can keep the latter in a separate vector whose names are the accession numbers—we see later the advantage of using this structure—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:

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

Böhning-Gaese et al. [27] 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
 "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 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.clu, taxa.sylvia, sylvia.eco,
+      file = "sylvia.RData")
```

3.8.2 Mammalian Mitochondrial Genomes

In the first edition of this book, this case study used a data set downloaded in April 2005 representing 109 species. For this second edition, the analyses have been updated with a larger data set. Except for a few changes—number of lines to skip or read, number of species—the commands below are virtually identical to those in the first edition, thus demonstrating the concept of "programmability" discussed in Chapter 1.

Gibson et al. [106] 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.[12] All mammalian mtGenomes available in the database were downloaded in March 2011. This represents 233 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                               #
################################################

#ORYCUNMIT : _Oryctolagus cuniculus_ (rabbit) : MITOCHONDRION
#OCHPRIMIT : _Ochotona princeps_ (American pika) : MITOCH....
....
```

After the 233 species names and codes, the sequences are printed in FASTA format. For instance, the lines 240–242 are:

```
>ORYCUNMIT(ATP6)
atgaacgaaaatttattctcctctttcgctaccccaacactaatagggctccctatt....
caactttactatttccctcccctagccgactaattaacaaccgactagtctcaaccc....
....
```

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

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:

- Skip the first four lines,
- Read only 233 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 = 233, 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 #ORYCUNMIT :  Oryctolagus cuniculus NA
2 #OCHPRIMIT :      Ochotona princeps NA
3 #OCHCOLMIT :      Ochotona collaris NA
4 #LEPEURMIT :        Lepus europaeus NA
5 #EREGRAMIT :      Eremitalpa granti 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 ORYCUNMIT Oryctolagus cuniculus
```

```
2 OCHPRIMIT     Ochotona princeps
3 OCHCOLMIT     Ochotona collaris
4 LEPEURMIT       Lepus europaeus
5 EREGRAMIT     Eremitalpa granti
```

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 = 239)
> length(mtgen)
[1] 9087
```

We now check the names of the first ten sequences:

```
> names(mtgen)[1:10]
 [1] "ORYCUNMIT(ATP6)" "OCHPRIMIT(ATP6)" "OCHCOLMIT(ATP6)"
 [4] "LEPEURMIT(ATP6)" "EREGRAMIT(ATP6)" "PONBLAMIT(ATP6)"
 [7] "CAPHIRMIT(ATP6)" "MONMONMIT(ATP6)" "BALOMUMIT(ATP6)"
[10] "CAPMARMIT(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 (Appendix A).

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

```
> unique(genes)[15]
[1] "RNL)Sequence does not exist"
```

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

```
> i <- grep("Sequence does not exist", names(mtgen))
> str(i)
 int [1:519] 3137 4149 4382 4615 4848 5081 5547 5780 ...
```

There are thus 519 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)
```

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

```
> table(genes)
genes
         ATP6          ATP8          COX1          COX2
          233           233           233           233
         COX3          CYTB           ND1           ND2
          233           233           233           233
          ND3           ND4          ND4L           ND5
          233           233           233           233
          ND6           RNL           RNS      tRNA-Ala
          233           232           233           233
     tRNA-Arg      tRNA-Asn      tRNA-Asp      tRNA-Cys
          233           232           232           232
     tRNA-Gln      tRNA-Glu      tRNA-Gly      tRNA-His
          232           232           233           232
     tRNA-Ile      tRNA-Leu tRNA-Leu(CUN) tRNA-Leu(UUR)
          232            16           217           217
     tRNA-Lys      tRNA-Met      tRNA-Phe      tRNA-Pro
          232           232           230           227
     tRNA-Ser tRNA-Ser(AGY) tRNA-Ser(UCN)      tRNA-Thr
           17           216           218           232
     tRNA-Trp      tRNA-Tyr      tRNA-Val
          232           232           233
```

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 233 rows and 4 columns that will store the results:

```
> BF.sp <- matrix(NA, nrow = 233, 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 233 species.

A simple approach is to use a `for` loop where a variable, say `i`, will vary from 1 to 233: this will be used as index for both `BF.sp` and `mtgen.taxa$code`. The commands are relatively simple 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 1:233) {
+     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. 3.9):

```
> matplot(BF.sp, type = "l", col = 1, xlab = "Species",
+         ylab = "Base frequency")
> legend(0, 0.22, c("A", "C", "G", "T"), lty=1:4, bty="n")
```

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 39 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 = 39, ncol = 4)
> rownames(BF.gene) <- unique(genes)
> colnames(BF.gene) <- c("A", "C", "G", "T")
> for (i in 1:nrow(BF.gene)) {
+   x <- grep(rownames(BF.gene)[i], names(mtgen), fixed=TRUE)
+   BF.gene[i, ] <- base.freq(mtgen[x])
+ }
```

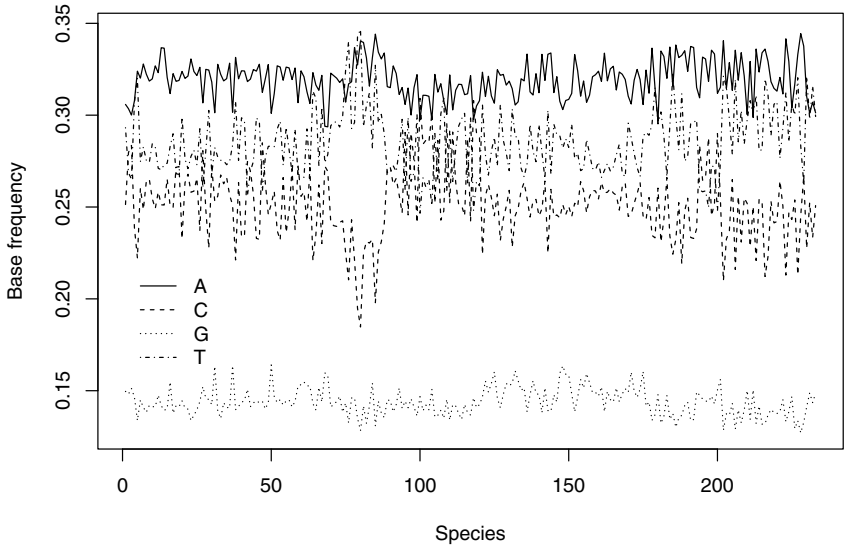

Fig. 3.9. Plot of the base frequencies of the mitochondrial genome of 233 species of mammals

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. 3.10):

```
> par(mar = c(8, 3, 3, 2))
> barplot(t(BF.gene), las = 2, legend = TRUE,
+         args.legend = list(horiz = TRUE, bg = "white"))
```

For the third analysis—between sites for a single gene—we focus on the genes of the cytochrome oxydase 1 whose code is `COX1`. We first extract the sequences of this gene by taking the appropriate indices in the way seen above:

```
> cox1 <- mtgen[grep("COX1", names(mtgen))]
> cox1
233 DNA sequences in binary format stored in a list.

Mean sequence length: 1545.056
   Shortest sequence: 1539
    Longest sequence: 1557
```

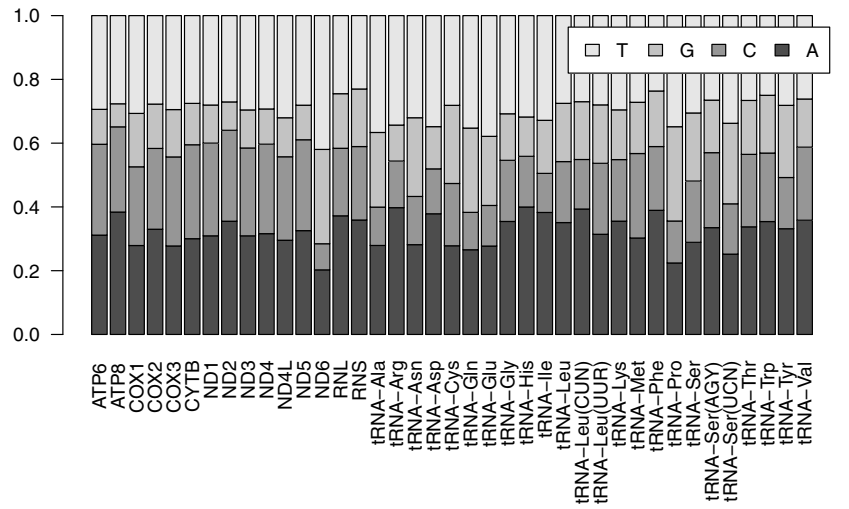

Fig. 3.10. Plot of the base frequencies of the mitochondrial genome of 233 species of mammals for each gene

```
Labels: ORYCUNMIT(COX1) OCHPRIMIT(COX1) OCHCOLMIT(COX1)
        LEPEURMIT(COX1) EREGRAMIT(COX1) PONBLAMIT(COX1) ...

Base composition:
    a     c     g     t
0.279 0.247 0.167 0.307
```

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

```
> table(sapply(cox1, length))

1539 1540 1541 1542 1543 1545 1546 1548 1551 1554 1557
   1    5    1   57    4  126    2    7   23    6    1
```

The majority of these sequences has 1545 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(cox1)`) suggests this is true for the whole sequences. We align the sequences with Muscle and MAFFT:

```
> cox1.muscle <- muscle(cox1)
> cox1.mafft <- mafft(cox1)
```

The two returned alignments differ in size:

```
> dim(cox1.muscle)
[1]  233 1569
> dim(cox1.mafft)
[1]  233 1565
```

A visualization with `image` shows that the difference comes from the insertion of gaps towards the 3' end of the sequences. Indeed, both alignments agree over 1532 nucleotides, and we select this portion (which covers 99.16% of the average sequence lenght) for our subsequent analyses.

```
> nm <- rownames(cox1.muscle)
> identical(cox1.muscle[, 1:1532], cox1.mafft[nm, 1:1532])
[1] TRUE
> cox1.ali <- cox1.muscle[, 1:1532]
```

To extract the first, second, or third codon position we use logical indixing on the columns as explained on page 56. So, we make a loop where at each iteration a vector with three values `FALSE` is created and the appropriate element is changed to `TRUE`. This is then used to subset the columns of the alignment and compute the base frequencies. (Compared to the first edition, this step is considerably simplified because we work direcly with the alignment.)

```
> BF.cox1 <- matrix(NA, 3, 4)
> rownames(BF.cox1) <- paste("codon position", 1:3)
> colnames(BF.cox1) <- c("A", "C", "G", "T")
> for (i in 1:3) {
+     s <- rep(FALSE, 3)
+     s[i] <- TRUE
+     BF.cox1[i, ] <- base.freq(cox1.ali[, s])
+ }
> BF.cox1
                          A         C          G         T
codon position 1 0.2655149 0.2037661 0.29134156 0.2393775
codon position 2 0.1814517 0.2610573 0.14905679 0.4084343
codon position 3 0.3829284 0.2784842 0.06391537 0.2746720
```

We plot the results again using `barplot` but adding a few annotations to present the figure (Fig. 3.11):

```
> par(mar = c(2.5, 4.1, 4.1, 1))
> barplot(t(BF.cox1), main = "Cytochrome oxydase I",
+         ylab = "Base frequency")
> par(cex = 2)
> text(0.7, BF.cox1[1, 1]/2, "A", col = "white")
> text(0.7, BF.cox1[1,1]+BF.cox1[1,2]/2, "C", col = "white")
> text(0.7, sum(BF.cox1[1, 1:2]) + BF.cox1[1, 3]/2, "G")
> text(0.7, sum(BF.cox1[1, 1:3]) + BF.cox1[1, 4]/2, "T")
```

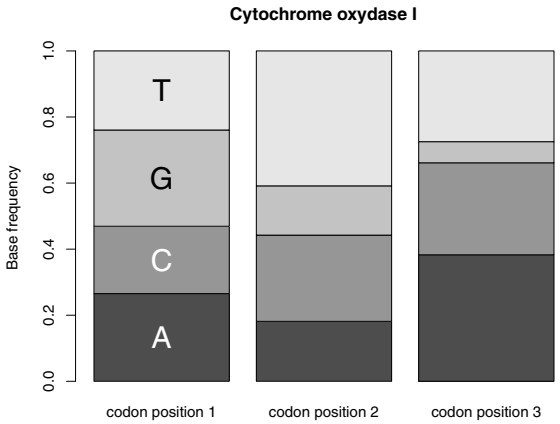

Fig. 3.11. Plot of the base frequencies at the three codon positions of the gene of the cytochrome oxydase I for 233 species of mammals

3.8.3 Butterfly DNA Barcodes

Hebert et al. [131] 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 [131] 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(sapply(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 use Clustal:

```
> astraptes.seq.ali <- clustal(astraptes.seq)
```

A graph with `image(astraptes.seq.ali, "-")` shows that all alignment gaps have been inserted at the beginning or the end of the sequences.

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._INGCUPnumt     Astraptes_sp._LOHAMP
                      66                       47
    Astraptes_sp._LONCHO       Astraptes_sp._MYST
                      41                        3
    Astraptes_sp._SENNOV      Astraptes_sp._TRIGO
                     105                       51
    Astraptes_sp._YESENN
                      79
```

All specimens were thus attributed to *Astraptes* sp. with further information given as a code (explained in [131]). 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")
```

3.9 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 plot 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. (a) Write a function that will read trees from the Pfam database, so that so we can use it with:
   ```
   read.pfamtree(accnum, type = "full")
   ```
   where `accnum` is the accession number of the family, and `type` is the type of the alignment (see p. 44).
   (b) Extract the tree #1000 in Pfam. 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: [116]).
   (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. (a) Write a program that will extract single nucleotide polymorphism (SNP) from a sequence alignment. The output will include the position of the SNPs along the sequence and the observed bases (alleles). You will include an option to output the sequence of the constant sites.
   (b) Write a second program that will transform the above alignment into an object of class `"loci"`.

4

Plotting Phylogenies

> But at least a few computer graphics only evoke the response "Isn't remarkable that the computer can be programmed to draw like that?" instead of "My, what interesting data.".
>
> —Edward R. Tufte [301]

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* [49]. 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 [79, 119, 218, 317]. 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 [211].

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 five kinds of trees: phylograms (also called rectangular cladograms), cladograms (triangular cladograms), unrooted trees (dendrograms), radial trees, and circular (fan) trees. This function is a method: it

E. Paradis, *Analysis of Phylogenetics and Evolution with R*, Use R,
DOI 10.1007/978-1-4614-1743-9_4, © Springer Science+Business Media, LLC 2012

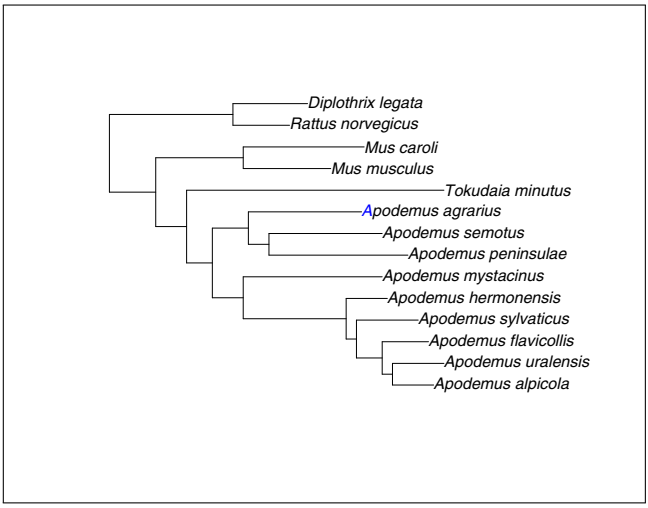

Fig. 4.1. A simple use of `plot(tr)`

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 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. [205]. 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,
```

[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"`, `"f"`
`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	`"black"` (d), a vector of strings giving the colors
`edge.width`	The line thickness of the edges	1 (d), a vector of numeric values
`edge.lty`	The line type of the edges	1 (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"`
`tip.color`	The colors of the tip labels	`"black"` (d), a vector of strings giving the colors
`draw`	Draw the tree	`TRUE` (d), `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,
     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

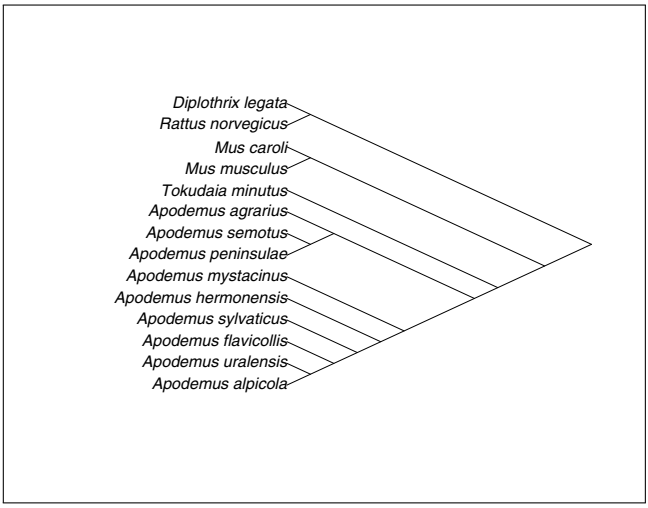

Fig. 4.2. A leftwards cladogram with default label justification

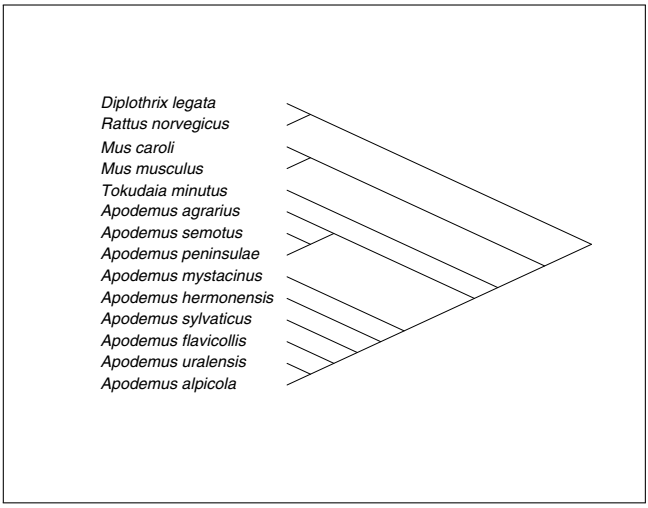

Fig. 4.3. A leftwards cladogram with left-justified labels

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 Murrell's excellent textbook [211], 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.

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" "font"
 [7] "cex"             "adj"             "srt"
[10] "no.margin"       "label.offset"    "x.lim"
[13] "y.lim"           "direction"       "tip.color"
[16] "Ntip"            "Nnode"
> tr.sett$x.lim
[1] 0.0000000 0.1326358
> tr.sett$y.lim
[1]  1 14
```

This shows that the horizontal axis of the plot in Fig. 4.1 ranges from 0 to 0.132. 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:

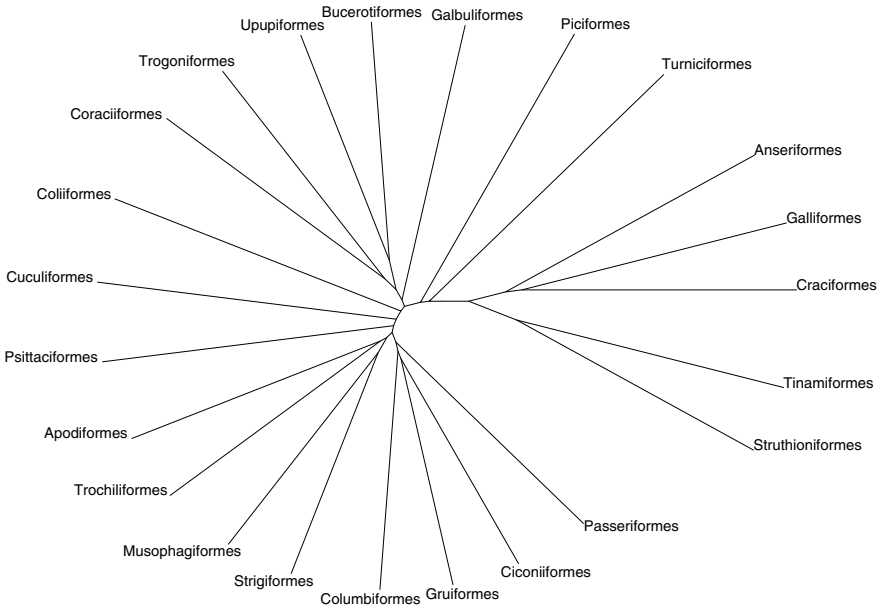

Fig. 4.4. An unrooted tree of the bird orders

```
plot(tr, x.lim = c(0, 0.264))
plot(tr, x.lim = c(-0.132, 0.132))
```

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 [79]. 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 `"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 [284]. The command used is:

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

Circular trees can be drawn with `plot(tr, "fan")` (Fig. 4.5). `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 [284]. This representation can be used for rooted and unrooted trees.

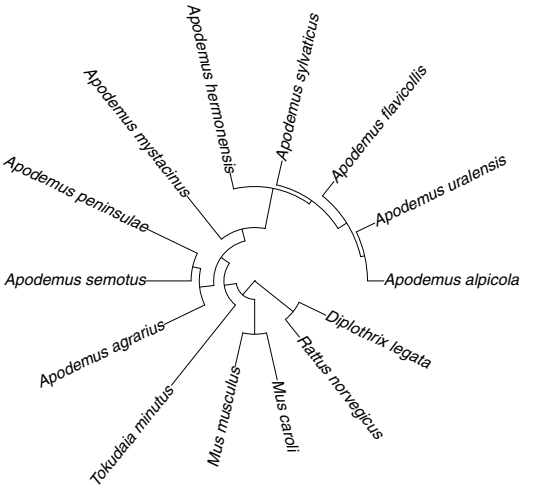

Fig. 4.5. A circular tree with `plot(tr, "fan")`

It has the advantages of being easily computed; the lines have no chance to cross and the tips are equally spaced. It should be noted that circular trees are not appropriate for displaying information on branch lengths because the edges are represented with different angles which is a well-known confusing effect in graphical display [301].

4.1.1 Node, Tip and Edge Annotations

`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.

Three functions offer a flexible way to add labels on a tree: `nodelabels`, `tiplabels`, and `edgelabels`. Their respective names tell us where the labels will be drawn. The point of these functions is to identify the nodes, tips, or edges using the numbering system of the class `"phylo"`(Section 3.1.1), then R finds itself where to exactly draw the text or symbols. They are low-level plotting functions: the labels are added on a previously plotted tree. They can print text (like the function `text`), symbols (like `points`), "thermometers" (like `symbols`), or pie charts (like `pie`) on all or some 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.

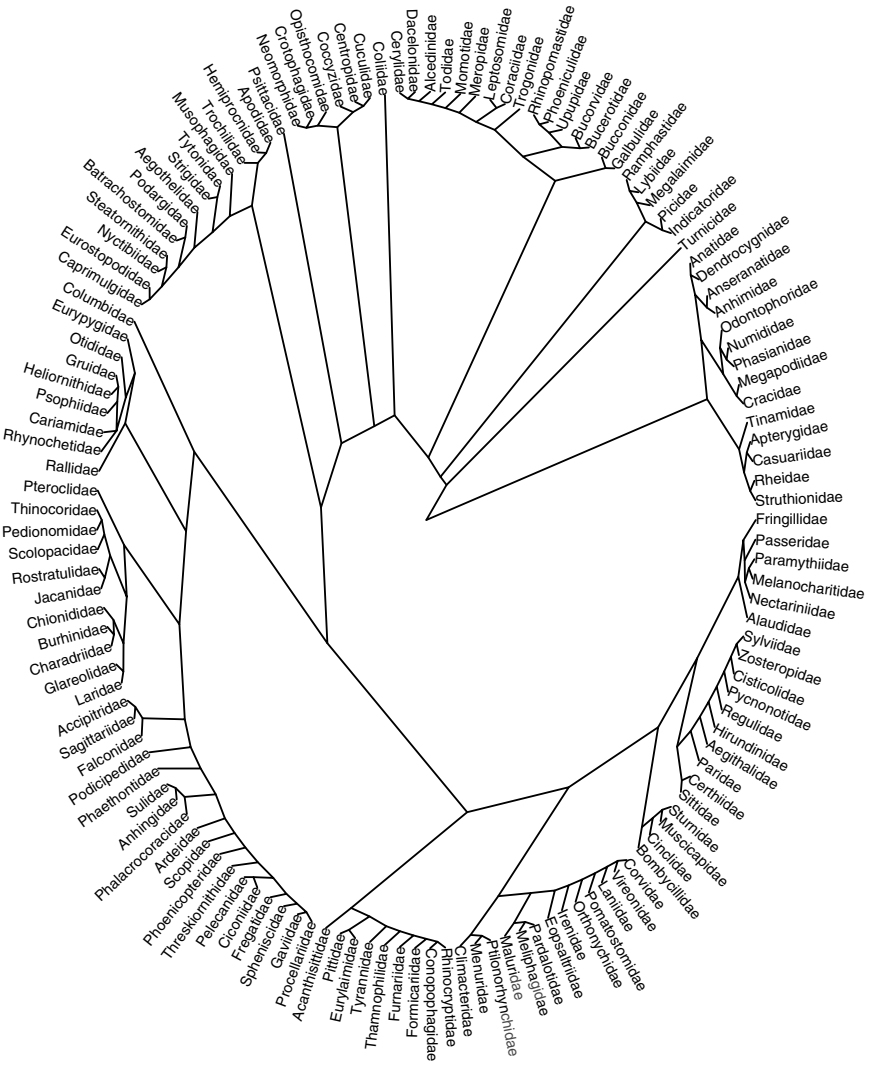

Fig. 4.6. A circular tree using `type = "radial"` in `plot.phylo`

The number of options of these three functions is quite small (Table 4.2), but they take 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 to another function, in the present case either `text` or `points` (see

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

Option	Effect	Possible values
`text`	Text to be printed	A character vector; can be left missing (d)
`node`, `tip`, or `edge`	Where to print	A numeric vector; can be left missing (d)
`adj`	Position with respect to the node, tip or edge	One or two numeric values; centered (d)
`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	A numeric vector or matrix
`pie`	Draw pie charts	id.
`piecol`	Colors for thermo or pie	A vector of string or a color code
`col`	Colors for text or symbol	id.
`bg`	Colors for the background of the frame or the symbol	id.
`...`	Further arguments	`cex =` , `font =` , `vfont =` `offset =` , `pos =`

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 these functions.

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—in the case of `nodelabels`—where the labels are printed are specified with `node`: this is done using the numbers of the `edge` element of the `"phylo"` object. 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). To display all numbers, we could do:

```
plot(tr); nodelabels(); tiplabels(); edgelabels()
```

That will plot the node numbers on a blue background, the tip numbers on a yellow background, and the edges numbers on a green background. Note that edge numbers are the row numbers of the `edge` matrix of the tree.

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. This will be true if the values to be plotted come from an analysis done on the tree like bootstrap values (Section 5.5) or ancestral reconstructions (Section 6.2), or node labels output from another program. In that case, the argument `node` can be left missing—or similarly `tip` or `edge` for the two other functions.

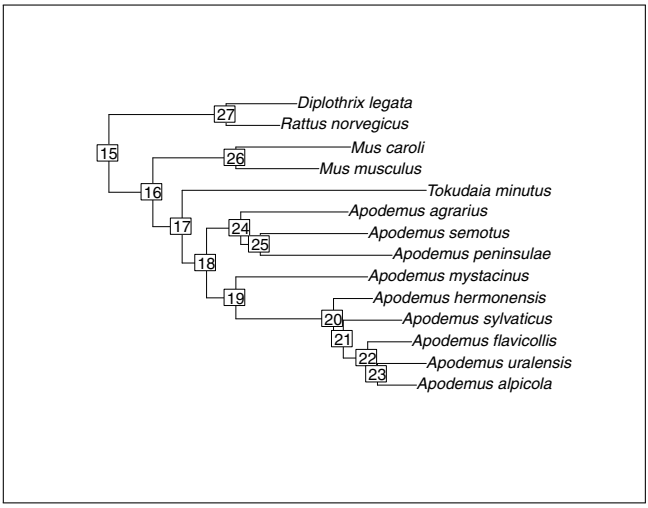

Fig. 4.7. Display of node numbers with `nodelabels()`

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. [289]:

```
trape <- read.tree(text = "((Homo,Pan),Gorilla);")
plot(trape, x.lim = c(-0.1, 2.2))
nodelabels("6.4 Ma", 4, frame = "c", bg = "white")
nodelabels("5.4 Ma", 5, 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 4 (number of tips + 1). Similarly, the second node is obviously numbered 5. 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 nodes. It is common to print the bootstrap values right-justified with respect to the nodes and without frames which can be done with:

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

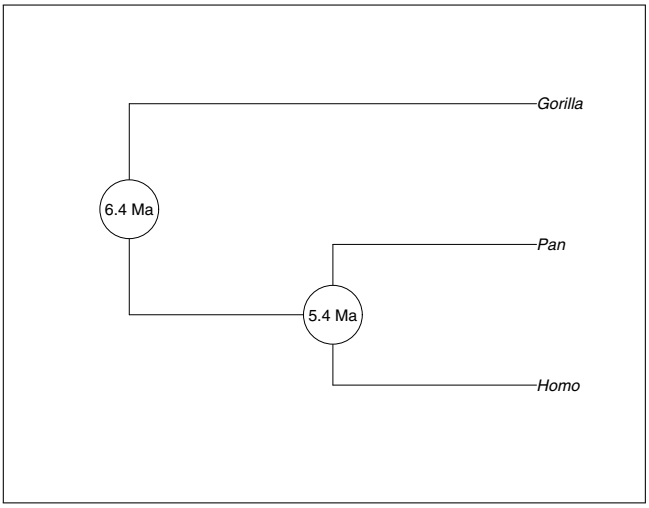

Fig. 4.8. Adding dates with `nodelabels`

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. [205] 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 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()
```

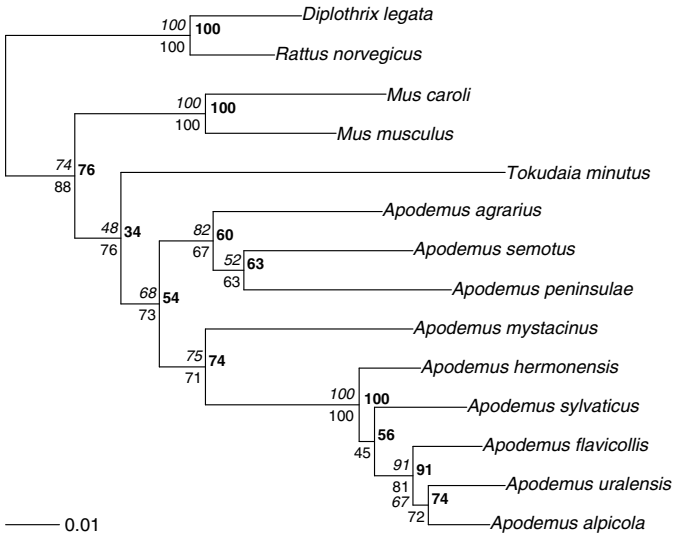

Fig. 4.9. Adding bootstrap values

```
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 (they will usually be read from the Newick file as node labels). 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):

```
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

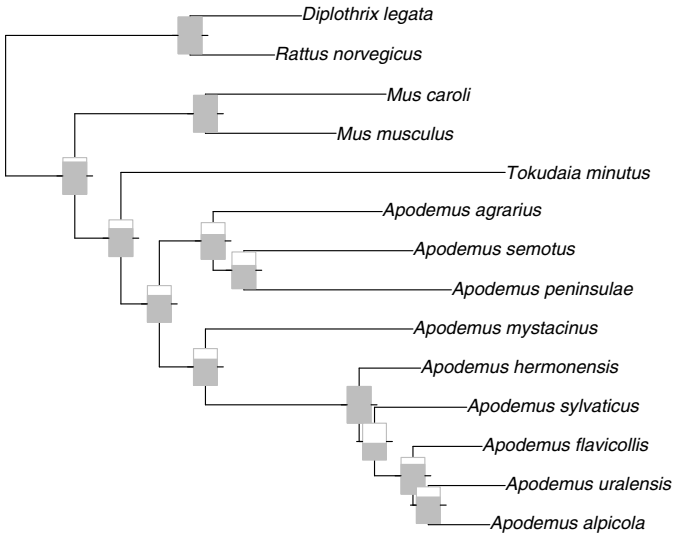

Fig. 4.10. Plotting proportions on nodes with thermometers

categories as a filled thermometer. This representation is less usual than circular symbols such as piecharts (option `pie`), 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, piecol = "grey")
```

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.

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

This is neat, but it is convenient to use a more flexible code:

```
co <- c("black", "grey", "white")
```

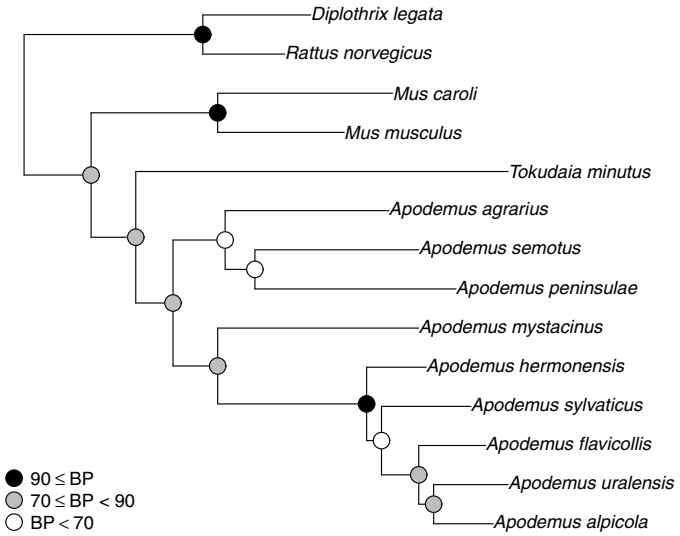

Fig. 4.11. Plotting symbols on nodes

```
p <- character(length(bs.ml))
p[bs.ml >= 90] <- co[1]
p[bs.ml < 90 & bs.ml >= 70] <- co[2]
p[bs.ml < 70] <- co[3]
```

The result is exactly the same but the advantage is that the vector `co` is easily modified, for instance, if the user wants to make a colored version of her graph. 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 = 16:27, 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 which can be done manually by two calls to `points` and `text`:

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

A more sophisticated solution is to use `legend`; we also pass the text of the legend as R expressions so they are plotted in a more elaborate way (Fig. 4.11):

```
legend("bottomleft", legend = expression(90 <= BP,
                         70 <= BP * " < 90", BP < 70),
```

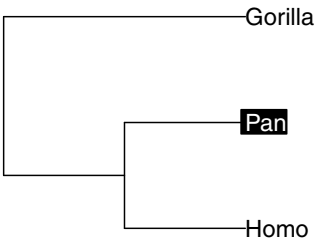

Fig. 4.12. Highlighting labels with `tiplabels`

```
pch = 21,  pt.bg = co, pt.cex = 2, bty = "n")
```

We have focused on `nodelabels` in this section, but the ideas developed here apply exactly to `tiplabels` and `edgelabels` as well. These functions allow us to print some labels highlighted on a colored background as can be seen sometimes in some publications for some tip labels. To achieve this, we first plot the tree without displaying the tip labels but we must leave some space to print them later. There are several ways to do this; here we plot the tree without options and save the settings that we then use to set the `x.lim` option:

```
o <- plot(trape)
plot(trape, show.tip.label = FALSE, x.lim = o$x.lim)
```

We can now print the tip labels with a single command where the options `bg` and `col` are set so that only the second tip label is highlighted:

```
tiplabels(trape$tip.label, adj = 0, bg = c("white", "black",
          "white"), col = c("black", "white", "black"))
```

We do not need to use the option `tip` because the tip labels are obviously correctly ordered in the element `tip.label` of the tree. With this command all labels would appear with a frame around them which may not be aesthetically the best result. To avoid this we have to call `tiplabels` twice using `tip` this time and `frame = "n"` for the labels not highlighted (Fig. 4.12):

```
tiplabels(trape$tip.label[2], 2, adj = 0,
          bg = "black", col = "white")
tiplabels(trape$tip.label[-2], c(1, 3), frame = "n", adj = 0)
```

We will see more details in Section 4.1.5 on how to use this kind of tools to explore phylogenetic data.

4.1.2 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`. This location may be found interactively with `add.scale.bar(ask = TRUE)`: the user is then asked to click on the graph to indicate where to draw the bar. 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). In all cases, the bar is drawn parallel to the branches of the tree.

`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. 4.18 and 4.20). 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.

`axisGeo` in phyloch adds a geological time scale below a plotted tree with branch lengths scaled in million of years. This has many options such as specifying which levels of the geological periods to print.

4.1.3 Manual and Interactive Annotation

R's low-level plotting commands can be used to annotate a tree manually 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 adds various annotations (Fig. 4.13):

```
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)
```

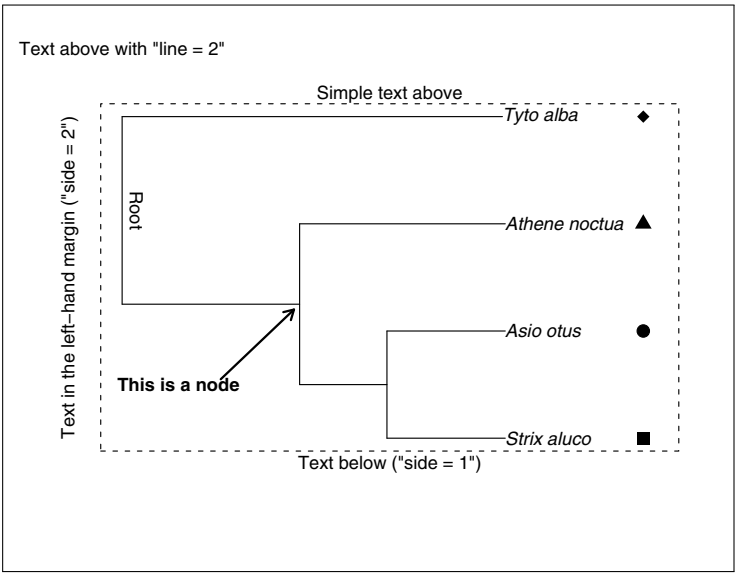

Fig. 4.13. Manual annotation of a tree

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.

All the above examples show how to add graphical elements once the coordinates or the numbers of nodes or tips are known. It is indeed important to be able to make graphics with lines of commands because they may be saved for future use, or they may be programmed for more complex analyses of exploration of data. On the other hand, in many situations it may be tedious to find all these coordinates or numbers, and a more interactive way of drawing may be preferable.

The coordinates of any location on a plot may be obtained with the base graphics function `locator`. It may be used without option: the user then left-clicks on the plot up to 512 times; a right-click interrupts the acquisition and a list is then returned with two vectors `x` and `y` containing the coordinates of all the locations previously clicked. It is not unusual to forget to assign such a procedure (i.e., `XY <- locator()`), so `.Last.value` (Section 2.1) may be useful here. The list returned by `locator` can in fact be used directly by most graphical functions, so it is not even needed to print these coordinates, even to save them. For instance, if you type:

```
text(locator(1), "Some text")
```

You then click on the plotting region, and "Some text" will be printed at this location.

`identify` is another function to interact with a plot. It is a generic function: in its standard form it is used after a bivariate plot, say `plot(x, y)`, with `identify(x, y)`. As above, the user is invited silently to click on the plot, and interrupts the process with a right-click. The indices of the data points that are closest to the clicked locations are returned, and these points are identified on the plot (this may be switched off with the option `plot = FALSE`).

There is an `identify` method for the class `"phylo"` (so it is documented at `?identify.phylo`). Its options are:

```
identify(tr, nodes = TRUE, tips = FALSE, labels = FALSE)
```

By contrast to the default method, the user is allowed to click only once on the tree: this will identify a node by its number. For instance, if we do the following command and click close to the node marked with the arrow on Fig. 4.13:

```
> identify(tree.owls)
Click close to a node of the tree...
$nodes
[1] 6
```

The returned object is a named list. Switching `tips = TRUE` results in returning the indices of the node and the tips descending from it:

```
> identify(tree.owls, tips = TRUE)
Click close to a node of the tree...
$tips
[1] 1 2 3

$nodes
[1] 6
```

Using `labels = TRUE` returns the labels instead of the numbers:

```
> identify(tree.owls, tips = TRUE, labels = TRUE)
Click close to a node of the tree...
$tips
[1] "Strix_aluco"   "Asio_otus"   "Athene_noctua"

$nodes
[1] 6
```

This tree has no node labels, so the node is still identified by its number.

The call to `identify` may be embedded in `nodelabels` or `tiplabels`:

```
nodelabels("Some text", identify(tree.owls)$nodes)
```

This command will print "Some text" at the node closest to the clicked location.

4.1.4 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 facing 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 coordinates of the tips of the tree on the y-axis are 1, 2, and so on. This may be helpful in specifying the coordinates of the vertical bars. Figure 4.14 shows a simple example with a phylogeny of bird orders; the commands used were:

```
data(bird.orders)
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.

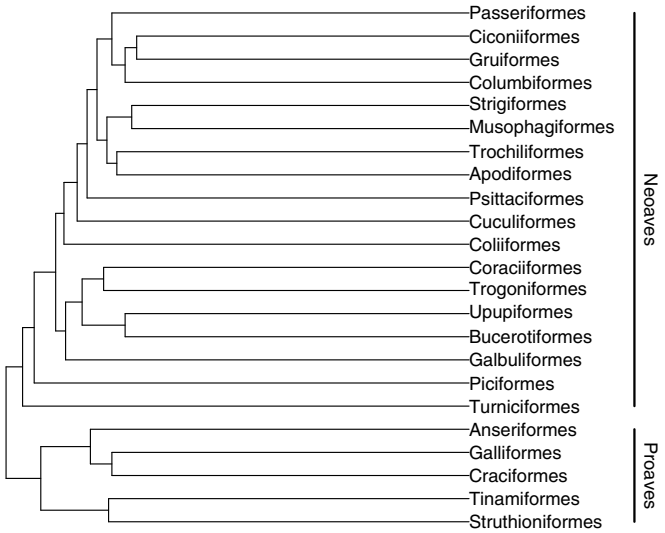

Fig. 4.14. Simple bars

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`, `edge.width`, and `edge.lty` allow us to specify the color, width, and line type 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 its `edge` matrix, say `tr$edge`, 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", Nedge(bird.orders))
```

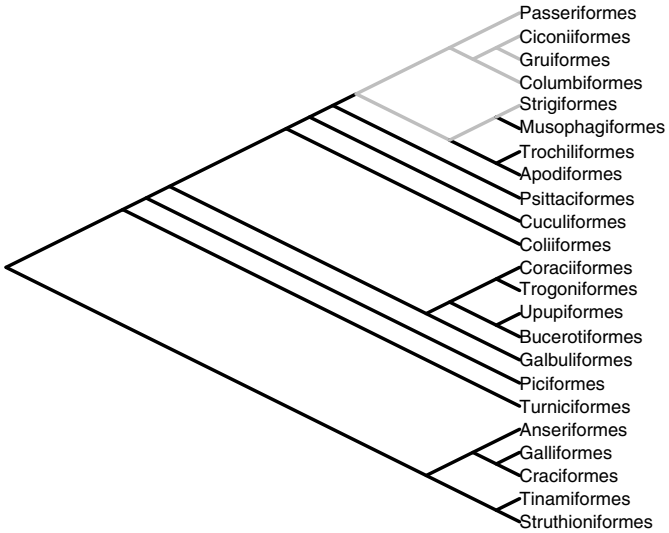

Fig. 4.15. Simple edge colors

The function `Nedge` extracts the number of edges of a tree: 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"
```

The tree can now be drawn. We use wider lines to display the difference in colors better (Fig. 4.15):

```
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 `locator`.

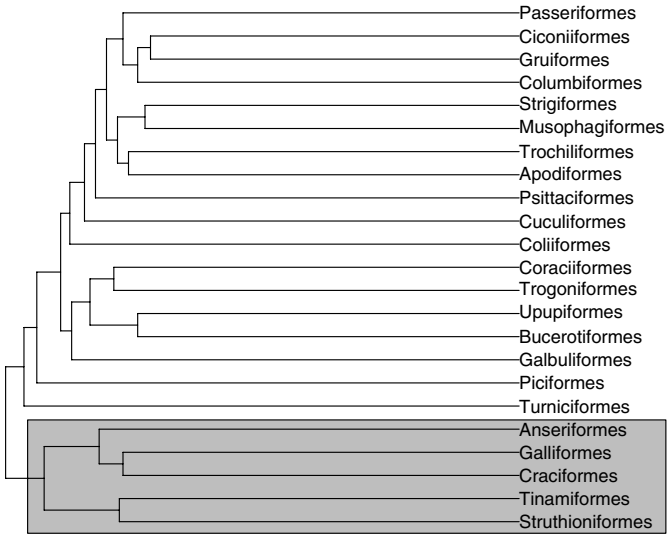

Fig. 4.16. A framed clade

A more flexible solution is to first call `plot.phylo` with its option `draw = FALSE`: the result is to set the graphics as if the tree was plotted. Low-level plotting commands can then be called as usual: the results will be plotted on a blank window. The tree can be plotted on top of the graph. The trick here is to set the graphical parameter `new` as `TRUE`: this tells R that the current graphical device has been freshly opened, and the output of the next high-level plotting command can plotted on it (Fig. 4.16):

```
plot(bird.orders, font = 1, no.margin = TRUE, draw = FALSE)
rect(1.2, 0.5, 36, 5.4, col = "lightgrey")
par(new = TRUE)
plot(bird.orders, font = 1, no.margin = TRUE)
```

After calling a high-level plotting command, such as `plot`, the `new` is automatically set to `FALSE`. Setting it to `TRUE` actually forces high-level plotting commands to behave like low-level ones.

An interesting feature of setting `draw = FALSE` is that the coordinates are computed and saved in a special environment. This gives many possibilities to program one's own graphical function.

4.1.5 Plotting Phylogenetic Variables

We have seen various tools to plot a tree with colors and other annotations with text and symbols. These serve for the appearance of the tree, but can be tools for representing variables linked to a tree. As such, these may be powerful

exploratory analysis tools. The basic idea is that variables associated to a tree will match the indexing and numbering system of the class `"phylo"` (p. 30). For instance, a variable associated to the n tips of a tree will be in vector of length n so that its first element will be the value of the first tip, and so on. Similarly, a variable associated with the branches will be in a vector of the same length than the number of branches (just like the `edge.length` element in the class `"phylo"`). Such variables may be passed to the appropriate arguments of some functions previously described, either directly or after some manipulations that we will illustrate in this section.

The functions to be used are `plot.phylo`—with its options `tip.color`, `cex`, `font` for the tips, and `edge.color`, `edge.width`, `edge.lty` for the branches (Table 4.1)—`tiplabels`, `nodelabels`, and `edgelabels`—with their options `text`, `pch`, `thermo`, `pie`, `col`, `bg`, and `cex` (Table 4.2). All these arguments use the "recycling rule" of R which says that if the passed element is shorter than what is required, then it is repeated as many times as necessary. For instance, `plot(tr, tip.color = c("blue", "red"))` will color the tip labels alternatively in blue and red.

The variables to be represented may be passed directly—possibly after rescaling—to some of the above options (`edge.width`, `thermo`, `pie`, `cex`), whereas a coding is likely to be necessary in the other cases.

The possibilities are vast and some examples are given in the next chapters.

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. adephylo 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.

4.2.1 Tree–Variable Coplot

`table.phylo4d` in adephylo plots multivariate data sets facing a tree. All variables are centered and scaled by default. This function requires the data to be in an object of class `"phylo4d"` which is easily done with the function of the same name. We illustrate this with random data (Fig. 4.17):

```
X <- phylo4d(rcoal(20), matrix(rnorm(100), 20))
table.phylo4d(X, box = FALSE)
```

This function has also several options to customize the appearance of the plot.

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

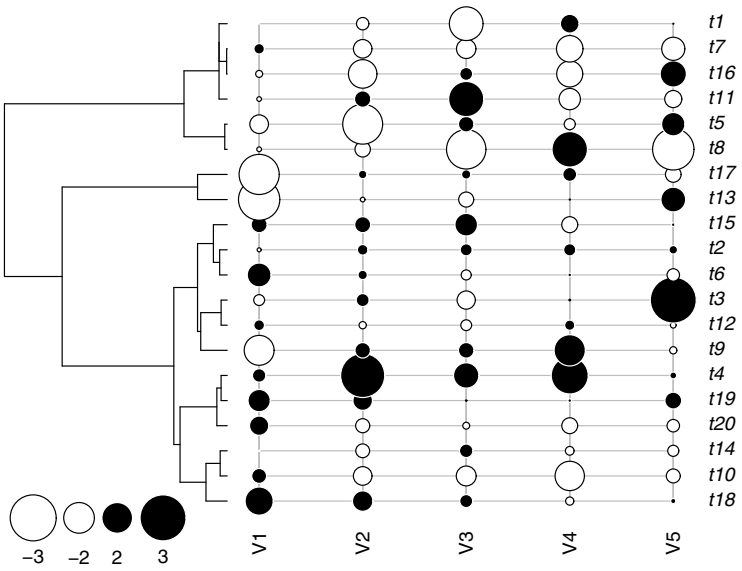

Fig. 4.17. The function `table.phylo4d` with random data

when plotting a phylogram or a cladogram, the tips have—in most cases—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
     Galliformes     Anseriformes    Turniciformes
```

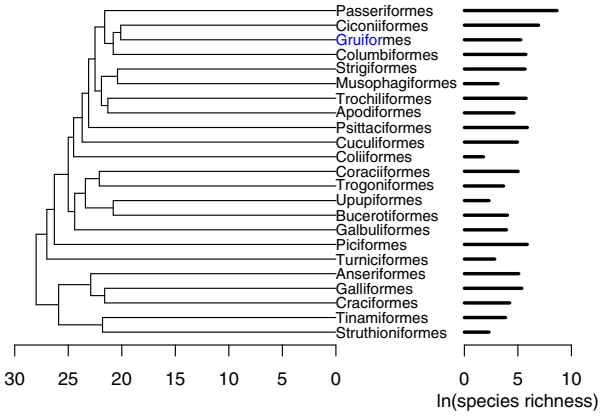

Fig. 4.18. Bars in the face of a tree plotted with `plot.phylo`

```
          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.[2] We can thus proceed in a simple manner (Fig. 4.18):

```
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)
axisPhylo()
```

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

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:[3]

```
>  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
```

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

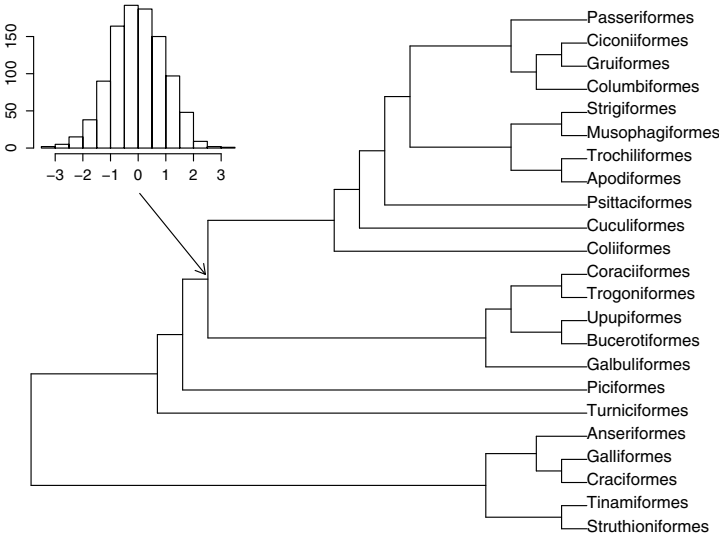

Fig. 4.19. Insert an histogram

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
```

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:[4]

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

4.2.2 Cophylogenetic Plot

To illustrate the possibilities given by `layout`, 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. [205] estimated divergence dates on their tree using a molecular clock. The tree on Fig. 4.1 could also be analyzed with the penalized likelihood method of Sanderson [276] using the calibration point of 12 Ma (million years ago) for the divergence *Mus–Rattus*. This is done with the function `chronopl` (Section 5.6). 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 needed commands are:

```
trk <- read.tree("rodent_clock.tre")
trc <- chronopl(tr, lambda = 2, age.min = 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. 4.20.

```
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 = 26, "?", adj = 2, bg = "white")
axisPhylo()
```

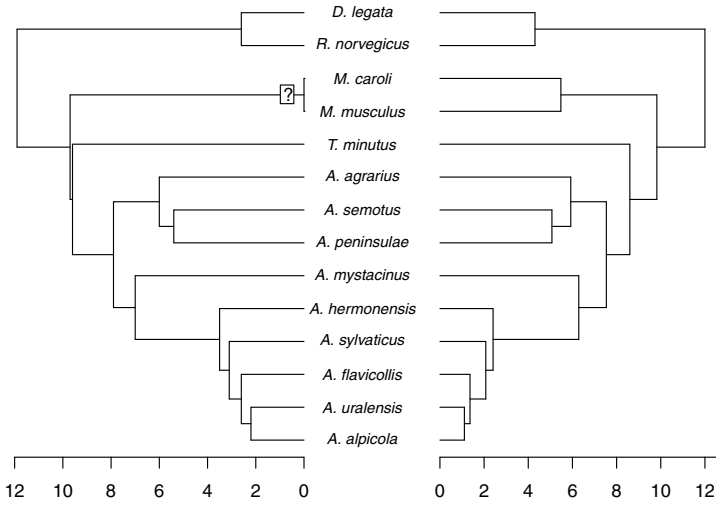

Fig. 4.20. Facing trees

```
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 the one below, so we cannot use the option `no.margin` of `plot.phylo`: instead we use the `par` function. The call to `nodelabels` is used to indicate that one node (the divergence between the two species of *Mus*) was not dated by Michaux et al. [205]. Finally, we draw the axis below each tree using `axisPhylo`.

It is debatable whether Fig. 4.20 is the best way to represent these results. Because the topologies are identical, the tree may be plotted together in the same direction for an easy comparison:

```
layout(matrix(1:2, 2, 1))
plot(trk); title("trk"); axisPhylo()
plot(trc); title("trc"); axisPhylo()
```

Besides, a more accurate comparison of the branching times may be obtained with the function `branching.times` (see Exercises).

The use of `layout` is easily generalized for plotting more than two trees. Additionally, if the trees are in a list of class `"multiPhylo"`, there is a `plot` method which second argument is `layout` specifying the number of trees plotted at the same time.

ape has the function cophyloplot that is devoted to plot two facing trees with links between their tips. The two trees may have distinct labels. As an example we will take a subset of the data from Light and Hafner [184] who studied the phylogeny of several species of rodents and their lice of the genus *Fahrenholzia*. We take two small clades from their Fig. 4:

```
TR <- read.tree("host_parasite.tre")
```

The file 'host_parasite.tre' contains the two trees, the first one is the host phylogeny. cophyloplot also needs a two-column matrix where each row is a link between the two trees:

```
> A
     col1              col2
[1,] "C.hispidus"      "F.zacatecae-C.h."
[2,] "C.formosus"      "F.reducta-C.f."
[3,] "C.eremicus"      "F.zacatecae-C.e."
[4,] "C.intermedius"   "F.zacatecae-C.i."
[5,] "C.californicus" "F.tribulosa-C.c."
[6,] "C.baileyi"       "F.reducta-C.b."
```

We can now call the function (Fig. 4.21):

```
cophyloplot(TR[[1]], TR[[2]], A, space = 40,
            length.line = -3, lty = 2)
```

It is often necessary to allow a large space between the two trees via the space argument. By default standard lines are drawn to represent the links, so we use the option lty = 2 to have dashed lines.

Several trees can be plotted on the same (temporal) scale with kronoviz. The main argument is a list of trees, for instance with six simulated coalescent trees (Fig. 4.22):

```
TR <- replicate(6, rcoal(10), simplify = FALSE)
kronoviz(TR, horiz = FALSE, type = "c", show.tip.label=FALSE)
```

The option layout (not used here) controls how many trees are plotted at the same time (all by default), and horiz controls the direction of the trees (TRUE by default). All other options are passed to plot.phylo.

4.3 Large Phylogenies

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.

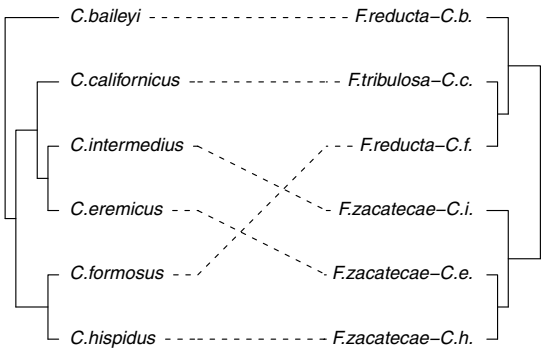

Fig. 4.21. Cophylogeny of six spexies of *Chaetodipus* and their parasites *Fahrenholzia* [184]

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 possibly 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

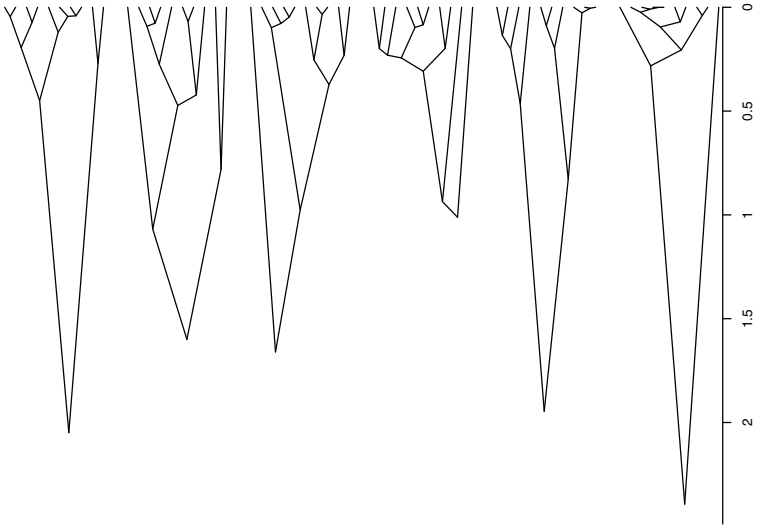

Fig. 4.22. Use of `kronoviz`

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 [154]. Our goal is to extract a subtree with the first 15 tips. The tree has 916 tips, thus the second argument to `drop.tip` could either be `16:916` or `chiroptera$tiplabel[-(1:15)]` with exactly the same result. We then plot the extracted tree (Fig. 4.23):

```
data(chiroptera)
tr <- drop.tip(chiroptera, 16:916, subtree = TRUE)
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.

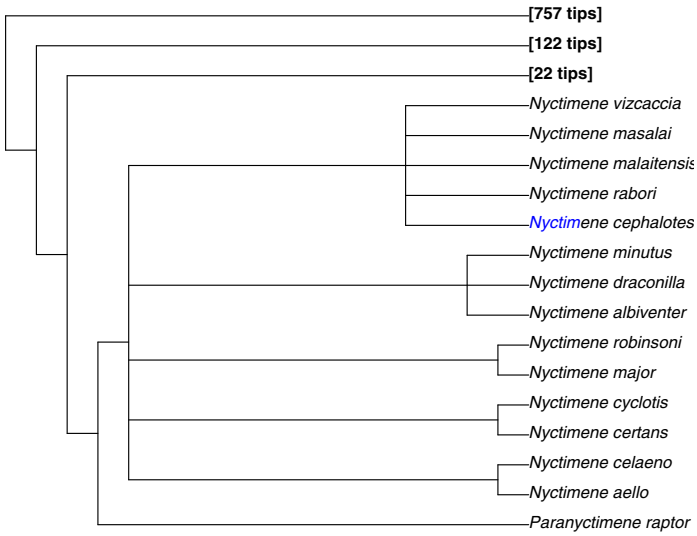

Fig. 4.23. Extracting a subtree

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.[5]

`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` 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 '`...`' argument (see p. 88).

A simple example of the use of `zoom` could be (Fig. 4.24):

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

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

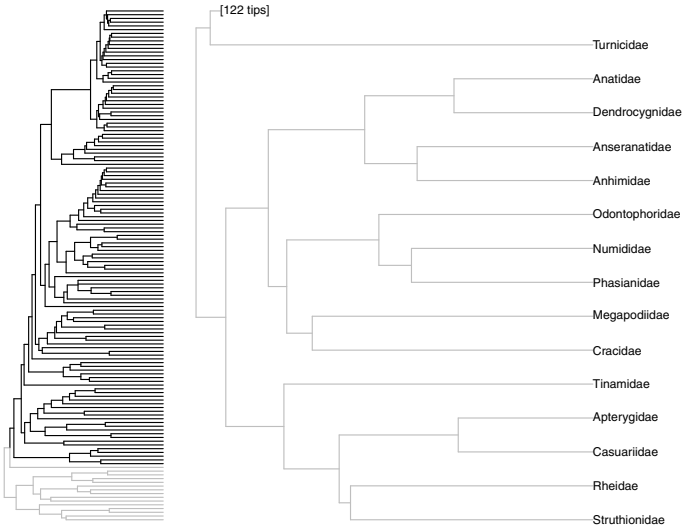

Fig. 4.24. Using `zoom`

```
     font = 1, 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 '`...`' 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. 4.25):

```
zoom(bird.families, list(1:15, 38:48),
     col = c("lightgrey", "slategrey"),
     no.margin = TRUE, font = 1, subtree = TRUE)
```

The function `trex` (*tree exploration*) can be seen as an interactive muti-window version of `zoom`. The idea is simple: you plot a tree, say `plot(tr)`, then call `trex(tr)` and left-click on any node and the subtree is plotted on a new window. You can click as many times as you want: this refreshes the new window. A right-click exits from the operation. There are options such as a default title and a different background color. Each time `trex` is called the subtree is plotted on a new window without closing or deleting those possibly already plotted. They may be distinguished with titles and / or background colors.

4.4 Networks

Networks of the class `"evonet"` and `"networx"` have their own `plot` method. Plotting networks is not trivial when branch lengths are involved because this

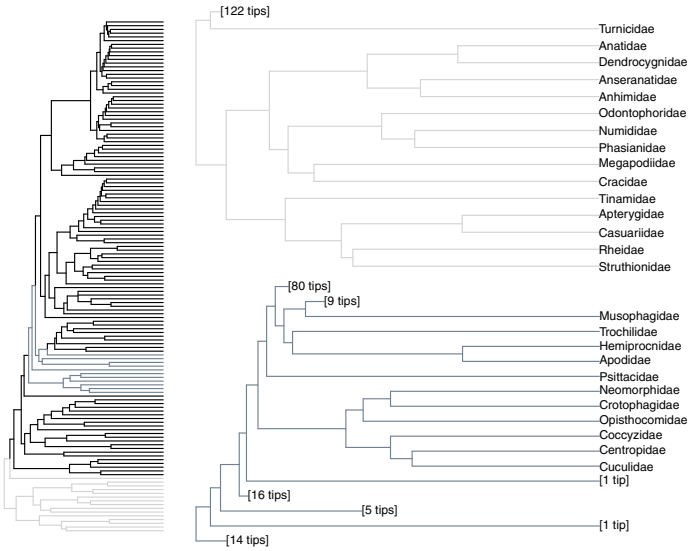

Fig. 4.25. Using `zoom` to show two groups

might not be physically possible. With the class `"evonet"`, this problem is solved simply by plotting the base tree of the network with `plot.phylo` and representing the additional reticulations in a different way. Possible information on the reticulations can be represented with line width or color. We take a simple network from a rooted tree:

```
> net <- evonet(stree(4, "balanced"), 6, 7)
> net

    Evolutionary network with 1 reticulation

               --- Base tree ---
Phylogenetic tree with 4 tips and 3 internal nodes.

Tip labels:
[1] "t1" "t2" "t3" "t4"

Rooted; no branch lengths.
```

The plot method calls `plot.phylo` to draw the tree; some arguments can be passed with '...'. Other arguments help to format the reticulations (Fig. 4.26):

```
plot(net, type = "c", col = "darkgrey", alpha = 1,
     arrows = 3, arrow.type = "h")
```

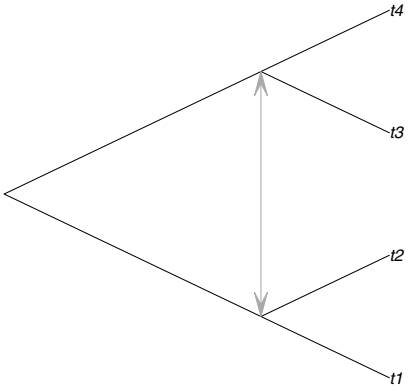

Fig. 4.26. A phylogenetic network (class `"evonet"`)

The argument `alpha` codes for the color transparency of the reticulations (0.5 by default). `arrows` follows the same syntax than the function of the same name, but if `arrow.type` is used a specific function, `fancyarrows`, from `ape` is used: the two possible choices are `"triangle"` and `"harpoon"` (or an abbreviation of these).

The class `"networx"` is appropriate to code consensus networks where reticulations represent uncertainty (instead of evolutionary events such as hybridization, gene transfer, or migration; see Section 5.5.4 on page 180). Nonetheless, a network of class `"evonet"` can be converted into this class but this requires the network to have branch lengths, so we create them with `compute.brlen` (Fig. 4.27):

```
ntx <- as.networx(compute.brlen(net, 1))
plot(ntx, "2D", edge.width = 1, tip.color = "black")
```

By default, `plot.networx` plots the network in 3-D with the package rgl (see an example in the next section); a standard plot is done with the option `"2D"`. This function uses the Kamada–Kawai algorithm which optimizes the spacing of the branches and the nodes. It is a random algorithm, so each call of the `plot` function produces a different output.[6]

It is possible to plot elaborate networks with the packages network and igraph. When converting trees of class `"phylo"` into objects of class `"network"` or `"igraph"`, the node labels, if present, are used as labels in the returned object together with the tip labels. Both classes have a `plot` method. Note that each repetition of the commands below will result in a different layout of the nodes (Fig. 4.28):

```
library(network)
library(igraph)
```

[6] See details in `?layout.kamada.kawai` from the package igraph.

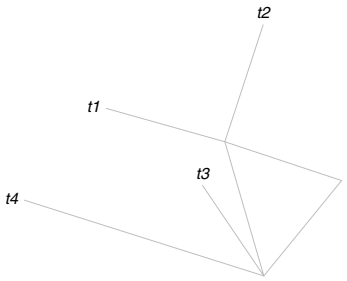

Fig. 4.27. A phylogenetic network (class `"networx"`)

```
layout(matrix(1:2, 1))
plot(as.network(net), vertex.col="white", displaylabels=TRUE)
plot(as.igraph(net), vertex.color = "white")
```

The parameters controlling the appearance of these plots are documented in `?plot.network` and `?igraph.plotting`, respectively.

In addition to its standard plotting function, igraph has the function `tkplot` which plots the network on a special device using the package tcltk: the graph can then be edited with the computer mouse moving the nodes freely, and the features of the nodes and edges (color, size, line type and width) can be edited like in a common graphical user interface with left- and right-clicks. The layout of the nodes can also be changed by calling an algorithm from the menu. Most importantly, the coordinates of the nodes can be saved with the function `tkplot.getcoords`.

A rich set of graphics for networks is available in Rgraphviz which is part of BioConductor. The package pegas has also a `plot` method for the class `"haploNet"`.

4.5 Data Exploration with Animations

Data exploration often requires to make many graphics and plots. Animations are useful tools to explore large-scale, high-dimensional data sets such as lists of phylogenies or molecular sequence alignments. There are a number of specialized computer programs dedicated to animated data exploration (e.g., GGobi [45]). Fortunately for us, several packages are available to perform similar tasks directly from R. We will focus on two of them: rgl and animation.

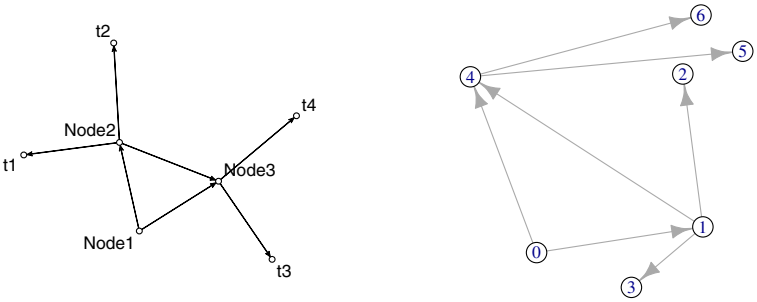

Fig. 4.28. Plotting a network with the packages network (left) and igraph (right). Note the root of the tree labeled "Node1" or "0" which is a node of degree 2

rgl is a port of the OpenGL library to R: it uses a specific graphical device (called "RGL") where the plot is rotated and zoomed in / out with the computer mouse. We come back to the network of class `"networx"` used in the previous section:

```
library(rgl)
open3d()
plot(ntx, edge.width = 1, tip.color = "black")
```

By default, the network is plotted under a dark grey background with blue tip labels, so we change these settings slightly. After the last command, the RGL can be manipulated freely by the user. An alternative is to animate the device with functions from rgl. A typical use is:

```
play3d(spin3d())
```

This will rotate the graph until the user stops it (`play3d` has the option `duration = Inf`). `spin3d` has two options which are, with their default values, `axis = c(0, 0, 1)` and `rpm = 5` to control the rotation.

Instead of playing the animation, the views can be saved in files in PNG format with (Fig. 4.29):

```
movie3d(spin3d(), 12, fps = 1, convert = FALSE, dir = ".")
```

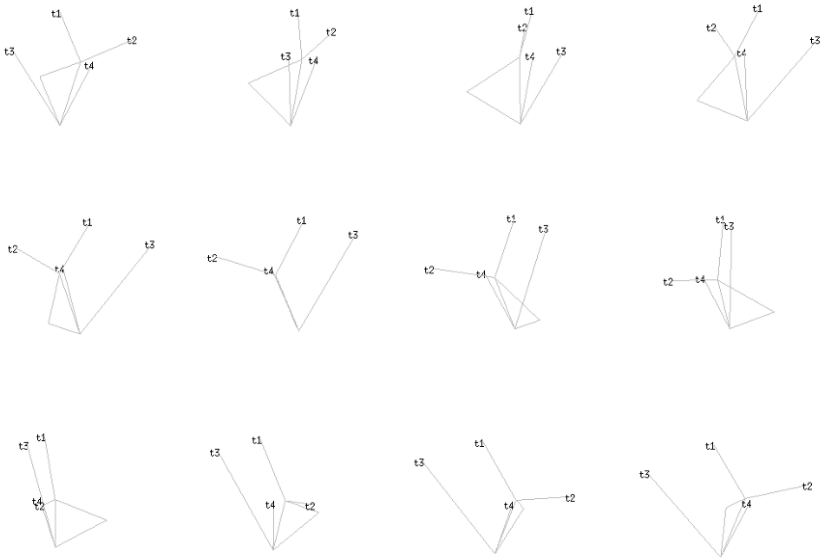

Fig. 4.29. Animation with phangorn and rgl

The second argument is the duration of the animation (here 12 s), `fps` is the number of frame(s) per second, `convert = FALSE` prevents to convert the PNG files into a single animation file (GIF by default which requires an external program such as ImageMagick), and `dir = "."` says to save the files in the current working directory (by default, a temporary directory is created).

The package animation provides several tools to create animations in various formats, some of them requiring external programs to be fully effective. Thus we will focus on `saveHTML` which creates a HTML file with animation coded in JavaScript. The individual files are saved in PNG. As an example, suppose we wish to visualize ten trees:

```
> TR <- rmtree(10, 10)
> library(animation)
> saveHTML(lapply(TR, plot))
HTML file created at: /tmp/Rtmp53QHfk/index.html
```

The files are stored in the temporary directory '/tmp/Rtmp53QHfk/' (this includes some subdirectories): the files can be accessed as long as the R session is open. This then opens a Web browser displaying sequentially the PNG files (Fig. 4.30). The parameters of the animation are controled with buttons.

The syntax is `saveHTML(<some R code>, <some options>)`, where the first argument can be any number of R commands (possibly included within

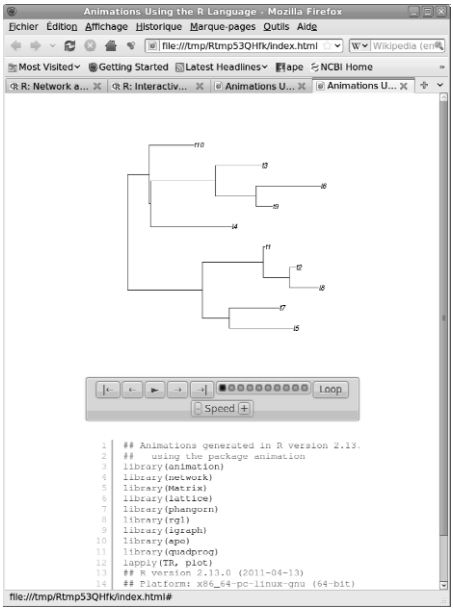

Fig. 4.30. HTML animation with animation

{ }). Often, it is useful to increase the size of the graphics, especially with high resolution screens (the default is 480×480 pixels). In the example below, we create an animation with the woodmouse data visualizing the four bases, gaps, and missing data (p. 66). The plot size is 800×1200 pixels.

```
saveHTML(
    for (b in c("a", "g", "c", "t", "-", "n"))
         image(woodmouse, b, "blue"),
    ani.height = 800, ani.width = 1200)
```

Clearly, this gives a lot of possibilities with very little programming. Below, we visualize the whole alignment in a standard way (all bases colored at the same time) displaying fifty columns:

```
saveHTML(
    for (i in seq(1, 901, 50)) {
       image(woodmouse[, i:(i + 49)])
       mtext(c(i, i + 50), at = c(1, 50), line = 1, font = 2)
    },
    ani.height = 800, ani.width = 1200)
```

For LaTeX users, `saveLatex` creates a PDF file with animations and has a similar syntax than the previous function, though the options are different; for instance:

```
saveLatex(
    for (i in seq(1, 901, 50)) {
       image(woodmouse[, i:(i + 49)])
       mtext(c(i, i + 50), at = c(1, 50), line = 1, font = 2)
    },
    ani.opts = "controls,loop,width=0.95\\textwidth",
    latex.filename = "woodmouse.tex",
    interval = 0.1, nmax = 10, ani.dev = "pdf",
    ani.type = "pdf", ani.width = 7, ani.height = 7)
```

A simpler alternative is to use the base function `Sys.sleep` which stops R execution during the number of seconds given as argument. This may be inserted in series of command such as:

```
for (i in seq(1, 901, 50)) {
    image(woodmouse[, i:(i + 49)])
    mtext(c(i, i + 50), at = c(1, 50), line = 1, font = 2)
    Sys.sleep(1)
}
```

4.6 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 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.

4. Consider the trees `trc` and `trk`. Do a comparison of branching times as suggested above. This will include a bivariate plot with a dotted line $x = y$. You will also indicate the node by their numbers on the plot.

5. Create a list of trees simulated using a Yule process with $\lambda = 0.1$ (see Chapter 7). Sort the trees in increasing order of number of tips. Create an animation to visualize sequentially the tree on top and its lineage-through time plot (Chapter 6) on bottom.

5

Phylogeny Estimation

> When I took up my post in Pavia in October 1961, [...] Cavalli-Sforza introduced me to his idea of developing methods for a computer-based construction of evolutionary trees from contemporary data [...]. Initially, I was skeptical. I had studied linkage estimation theory under Fisher himself and I knew how difficult it was. I had visions of evolutionary tree estimation being much the same but with the addition of the need to estimate the form of the tree itself, surely a fatal complexity: my intuition was that there would be insufficient data for the task. Cavalli-Sforza was very persuasive, however.
>
> —Edwards [63]

Reconstructing the evolutionary relationships among living species is one of the oldest problems in biology. 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 [43].
- 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 [12].
- It is often necessary to estimate many parameters simultaneously though only some of them are of interest [135].
- 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 [138].
- Some confusion arises because phylogeny estimation methods are also used for systematics (i.e., classification of species) rather than estimating evolutionary parameters.

E. Paradis, *Analysis of Phylogenetics and Evolution with R*, Use R,
DOI 10.1007/978-1-4614-1743-9_5, © Springer Science+Business Media, LLC 2012

- 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 [138].
- The different methods, models, and algorithms for phylogeny estimation are available in distinct programs resulting in several practical difficulties [169, 203, 202].

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. 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.

All phylogeny estimation methods are characterized by:

1. A numerical criterion to measure the adequacy of a given tree topology compared to another.
2. A formula to calculate the branch lengths.
3. An algorithm to explore the tree space.

The development of phylogeny estimation in R is very recent, and significant 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 [77], it can be viewed as a valid nonparametric method [138]. Bayesian methods enjoy a current success, but some critics pointed out the limitations of this approach [79, 293]. 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 as we will see later in this chapter.

5.1 Distance Methods

Distance methods have a long history because they are generally tractable even with a large amount of data [298]. Consider a phylogenetic tree $\mathcal{T}$ among a set of species. $\mathcal{T}$ defines a unique matrix of pairwise distances, Δ, given by the paths between species (the patristic distances):

$$\mathcal{T} \longrightarrow \Delta$$

However, for a given distance matrix, many trees may be defined depending on the criterion used to define them:

$$\Delta \longrightarrow \mathcal{T}_1, \mathcal{T}_2, \ldots$$

All the problematic of distance-based methods is to find the tree for a given distance matrix. The driving idea is that the distances in Δ give information on the relative closeness of observations. The distances are used to estimate both the topology of the tree and its branch lengths. There are two main strategies to find a tree: aggregating the most closely related observations, or splitting the most distant sets of observations. We will see below that for a given data set (sequences, measurements, ...) there are several possible distance matrices depending on the method used to compute them.

There has been considerable progress in implementing these methods in R, particularly based on the numerous methods to compute distances in R from various types of data which are the focus of the first section.

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

There are several ways to define a distance between two observations using numerical data. It would be too long to detail all of them here. Fortunately, the help pages of the functions mentioned below include the formulae of the different methods.

`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 strings: `"euclidean"` (the default), `"maximum"`, `"manhattan"`, `"canberra"`, `"binary"`, or `"minkowski"`. As a simple example, consider three variables taken on three individuals:

```
> X <- matrix(c(0, 1, 5), 3, 3)
> rownames(X) <- LETTERS[1:3]
> X
```

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.quant`	Continous
ape	`dist.gene`	Discrete
	`dist.dna`	Aligned DNA sequences
	`weight.taxo`	'Taxonomic' levels
	`cophenetic`	An object of class `"phylo"`
adephylo	`distTips`	Objects of class `"phylo"`, `"phylo4"` or `"phylo4d"`
adegenet	`dist.genpop`[a]	An object of class `"genpop"`
phangorn	`dist.ml`	An alignment of amino acids

[a]Replaces `dist.genet` from ade4

```
  [,1] [,2] [,3]
A    0    0    0
B    1    1    1
C    5    5    5
```

These may be viewed as three points in a 3-d space where A would be at the origin of the space.

```
> 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(X, "binary")
  A B
B 1
C 1 0
```

`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). This class can be converted into a matrix using the generic function `as.matrix`, and a matrix 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. Three metrics are available via the option `metric`: `"euclidean"`, `"manhattan"`, or `"gower"`. The last one implements Gower's coefficient of similarity for mixed data types [112]. The types of the variables are either identified with respect to the class of the columns, or specified with the option `type` (see `?daisy` for details). In the example below, we convert the columns of `X` as factors to build the data frame `Y`:

```
> daisy(X, "gower")
Dissimilarities :
    A   B
B 0.2
C 1.0 0.8

Metric :  mixed ;  Types = I, I, I
Number of objects : 3
> Y <- as.data.frame(apply(X, 2, factor))
> Y
  V1 V2 V3
A  0  0  0
B  1  1  1
C  5  5  5
> daisy(Y, "gower")
Dissimilarities :
  A B
B 1
C 1 1

Metric :  mixed ;  Types = N, N, N
Number of objects : 3
```

`dist.quant` in ade4 implements three metrics (canonical or Euclidean, Joreskog, and Mahalanobis) specified with an integer between 1 and 3:

```
> dist.quant(X, 1) # canonical
          A        B
```

```
B 1.732051
C 8.660254 6.928203
> dist.quant(X, 2) # Joreskog
          A         B
B 0.8017837
C 4.0089186 3.2071349
> dist.quant(X, 3) # Mahalanobis
          A         B
B 0.4629100
C 2.3145502 1.8516402
```

`dist` and `daisy` handle missing data and will usually correct the distances for their presence, but the returned distance may be `NA` if there are too many missing values. On the other hand, `dist.quant` does not accept missing values.

Evolutionary Distances

`dist.gene` provides a simple interface to compute the distance between two haplotypes using a simple binomial distribution of the pairwise differences. This allows one to compute easily the variance of the estimated distances using 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. Missing values are accepted and handled through the option `pairwise.deletion`. By default, the columns with at least one `NA` are deleted before computing (global deletion), so that all distances are computed with the same columns. If a few missing values are spread over all columns, this procedure may result in very few (even none) variables left for computing the distances. If `pairwise.deletion = TRUE`, columns are deleted on a pairwise basis and the distances are adjusted with the respective numbers of variables used.

`dist.dna` provides a comprehensive function for the estimation of distances from aligned DNA sequences using substitution models or counts of the number of differences (`"n"`, `"raw"` scales by the sequence length), transitions (`"ts"`), transversions (`"tv"`), or alignement gaps (`"indel"`, `"indelblock"` counts contiguous gaps as a single unit). The options are detaild in Table 5.2. The substitution models are described in Section 5.2.1.[1] 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. Missing and ambiguous nucleotides are taken into account with the option `pairwise.deletion` as described in the previous paragraph.

`dist.genpop` takes as input an object of class `"genpop"`. Five methods are available to compute these distances: standard (or Nei), angular (or Edwards),

[1] `dist.dna` calculates distances for the models whose an analytical formula has been found. See page 144 for a general numerical formula.

Table 5.2. Options of the function `dist.dna`. The values marked with (d) are the default ones

Options	Effect	Possible Values
`model`	Specifies the substitution model	`"raw"`, `"n"`, `"ts"`, `"tv"`, `"indel"`, `"indelblock"`, `"JC69"`, `"K80"` (d), `"K81"`, `"F81"`, `"F84"`, `"BH87"`, `"T92"`, `"TN93"`, `"GG95"`, `"logdet"`, `"paralin"`
`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`

Reynolds, Rogers, and Provesti. This is specified with the option `method` which takes an integer value between 1 and 5.

These three functions return an object of class `"dist"`, except `dist.dna` with `model = "BH87"` because the Barry–Hartigan model is asymmetric [21].

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.

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.

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"`. Additionally, adephylo has the function `distTips` computing various distances from a tree (patristic, node path, Abouheif's, and sum of descendants of nodes on path). ade4 and vegan have a comprehensive list of ecological distances.

5.1.2 Exploring and Assessing Distances

The choice of a particular distance is often dictated by the type of data at hand, but in most situations it is very useful to compare different distances computed with the same data. Because distances are easily computed, they may be potentially useful exploratory tools that have been underexploited in phylogenetic analyses. It is important to emphasize that even if a distance-based method is not used, pairwise distances contain information that could be explored before tree estimation.

A simple graphical exploration of the distances may be revealing. Because an object of class `"dist"` (say `d`) contains only the lower triangle of the distance matrix, it can be plotted with `hist(d)`. It is well-known that the way classes are defined in a histogram may hide patterns in the data, so it is useful to compare the default number of classes used by `hist` (approximately 20) with, for instance, `hist(d, 50)`. An alternative is to draw a curve of the empirical density on top of the histogram plotted as densities, instead of numbers, so its total area equals one:

```
hist(d, freq = FALSE)
lines(density(d))
```

lattice gives a nice alternative with `densityplot(~ d)`. Ideally, with the objective of phylogeny estimation in mind, one wants to have the distances spread as much as possible in order to obtain a resolved phylogeny. One has to be careful on the scale of the x-axis here because similarly shaped histograms may have different scales. Small distances may indicate observations that will be difficult to pull apart in the phylogeny (typically, < 0.005 for a sequence of 1000 sites). On the other hand, large distances will lead to a similar problem because of mutation saturation: the JC69 distance cannot exceed 0.75, but in practice distances greater than 0.3 lead to difficulties.

A second useful, though very simple, graphical exploration tool is to plot two sets of distances computed on the same data with the goal to contrast the effect of different methods. A form of this approach is well-known as the 'saturation plot' where the number of transitions or transversions is plotted against the K80 distance (see p. 191), but this can be generalized. For instance, the F81 model differs from the JC69 one by considering unbalanced base frequencies (p. 142). So plotting these two distances together will show the impact of this assumption:

```
djc69 <- dist.dna(x, "JC69")
dk81 <- dist.dna(x, "K81")
plot(djc69, dk81)
abline(b = 1, a = 0)
```

We take advantage of the fact that the observations are ordered in the same way in both `"dist"` objects. In case of doubt this may be checked with:

```
all(attr(djc69, "Labels") == attr(dk81, "Labels"))
```

Distance matrices may be characterized by a number of properties that may give helpful indications on the evolutionary information in the data. We will consider only a few of these properties. A particularly interesting one is *additivity*. A distance matrix is said to be additive if it satisfies the *four-point condition*. Consider a quartet of observations $q = \{x, y, u, v\}$, and define:

$$d_{xy|uv} = d_{xy} + d_{uv}.$$

Then the four-point condition is satisfied for q if among the three quantities $d_{xy|uv}$, $d_{xu|yv}$, and $d_{xv|yu}$ the two largest ones are equal. Additive distances may be represented appropriately with a tree structure. Holland et al. [136] developed an exploratory method to assess the addivity of a distance matrix. They define the ratio:

$$\delta_q = \frac{d_{xv|yu} - d_{xu|yv}}{d_{xv|yu} - d_{xy|uv}},$$

under the condition that $d_{xy|uv} \leq d_{xu|yv} \leq d_{xv|yu}$. If the distances are additive δ_q will be close to zero. The function `delta.plot` implements this method. δ_q is computed for all possible quartets, and their distribution is examined with a histogram. Additionally, a mean value $\bar{\delta}$ is computed for each observation taking all δ_q values where this observation is involved. For illustration, we take a distance matrix computed from a set of fifteen sequences of cytochrome b, and another matrix computed from a random set of five variables for fifteen observations (Fig. 5.1):

```
> data(woodmouse)
> dw <- dist.dna(woodmouse)
> x <- replicate(5, rnorm(15))
> dx <- dist(x)
> delta.plot(dw)
> delta.plot(dx)
```

A difficulty with δ-plots is that the shape of the distribution of the δ_q depends on several factors, including the number of variables (e.g., if `x` had only one column in the above example, then $\delta_q = 0$ for all quartets). It may thus be appropriate to examine some hypothetical scenarios with simulated data.

Another relevant property is whether a distance is *Euclidean*. Euclidean distances are measured in a purely isotropic space, i.e., a space where classical geometry applies. Most multivariate methods, such as principal components analysis, requires variables to be represented in a Euclidean space so that rotations are meaningful. `ade4` has the function `is.euclid` that returns `TRUE` or `FALSE`. Note that for a given data set, a distance matrix may or may not be Euclidean depending on the method used. A non-Euclidean distance matrix can be transformed into a Euclidean one by multiplying the distances

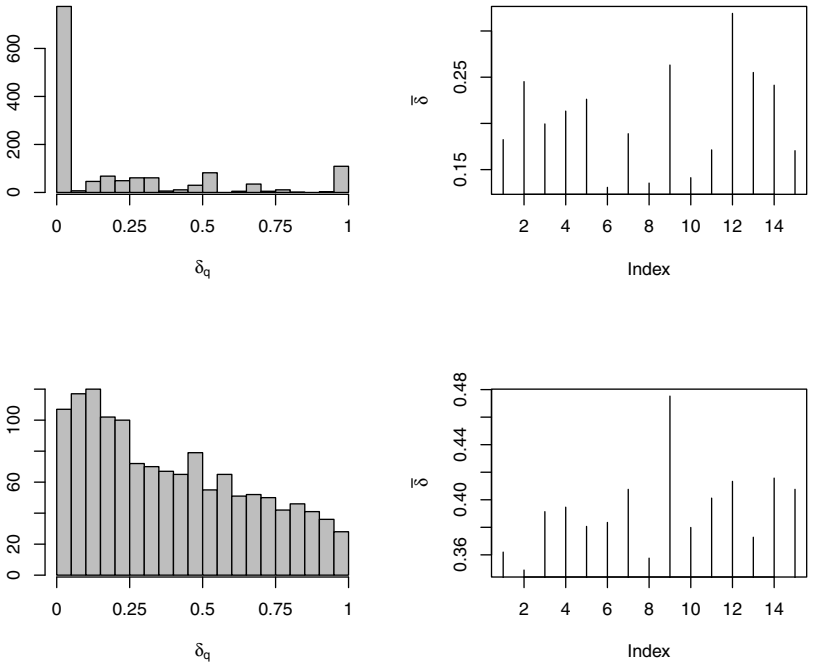

Fig. 5.1. δ-plot of fifteen sequences of cytochrome b (top) and fifteen random observations (bottom)

by a coefficient that is found from an eigen decomposition of the original matrix. Two functions are available in `ade4`: `cailliez` that does a simple linear transformation of the distances, and `lingoes` that does the same on the squared distances. Note also the function `quasieuclid` to 'clean' an already Euclidean matrix which is not completely so because of numerical approximations (Thibaut Jombart, personal communication). Recently, de Vienne, Aguileta and Ollier [51] showed that taking the square root of the distances make them Euclidean. This simple transformation, done with `sqrt(d)`, implies less distortion of the original distances compared to the Caillez transform.

A well-known property of distances in evolutionary biology is *ultrametricity*. A distance matrix is ultrametric if it satisfies the following inequality:

$$d_{xy} \leq \max(d_{xz}, d_{yz}) \ ,$$

for all triplets $\{x, y, z\}$. Consider the case where there are only three observations: it is clear that the above condition is met only if all distances are equal. So if we represent these three observations with a tree, it would have

three equal branch lengths, and all tips would be equally distant from the root which is the definition of an ultrametric tree. Ultrametricity is appealing for evolutionists because if evolution is homogeneous across lineages—but not necessarily through time—then distances among them are expected to be ultrametric. Several decades of research have shown that this will be not generally true with real data. Fortunately, a distance matrix may be non-ultrametric but still be additive. The reader is invited to explore further these ideas in the Exercices of this chapter.

Campbell, Legendre and Lapointe [36, 37] developed a test of the congruence among ultrametric distance matrices. It is implemented in the functions `CADM.global` and `CADM.post` in `ape`.

5.1.3 Simple Clustering, UPGMA, and WPGMA

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 aggregative. The first step is to find the two observations with the shortest pairwise distance in the matrix: this distance is used to calculate the 'height' at which they aggregate, that is the distance from the node to each of these tips. Since no other observation come into this height calculation, it is not possible to assume different rates of change in both branches, so they have the same length. The second step is to calculate the distances from the inferred node to the remaining observations: this is where clustering methods differ because there are many ways to recalculate distances. Once this is done, the two steps are repeated until no observation remains.

R has a reasonably large number of functions that perform clustering [307]. They mostly work on a distance (also called dissimilarity) matrix, but some of them work directly on the original data matrix (observations and variables). The unweighted pair-group method using arithmetic average (UPGMA) is similar to a hierarchical clustering with the average method as implemented in the `hclust` function. The only difference is that the branch lengths are halved in the UPGMA method. This is implemented in the function `upgma` in `phangorn`, for instance:

```
M <- dist.dna(woodmouse)
tr <- upgma(M)
```

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(woodmouse) # K80 is the default
tr1 <- upgma(M1)
M2 <- dist.dna(woodmouse, model = "F84")
```

```
tr2 <- upgma(M2)
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 5.8.

`upgma` has an option `method` that specifies the method used to update the distances in the second step described above. The default is `"average"`; six other methods are available and described in the help page `?hclust`. Setting `method = "mcquitty"` is the same thing that using the function `wpgma` that implements the weighted version or WPGMA. The topology and branch lengths of the tree are substantially affected by the choice of this method (see Exercices).

5.1.4 Neighbor-Joining

The neighbor-joining (NJ) is one of the most widely used methods of phylogeny estimation. It is a splitting method. The first step is to build a tree with a single internal branch where one node is linked to two observations (the neighbors), and the other is linked to all the others. All possible pairs of neighbors are considered: the tree with the smallest total branch length is selected. The second step is to update the distance matrix by removing the two neighbors and calculating the distance from the new node to the remaining observations. The two steps are repeated until the tree is dichotomous. The branch lengths are estimated by least squares, so the rates of evolution may vary among branches.

`ape` has the function `nj` that performs the NJ algorithm. Its use is 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.

Because branch lengths are estimated by least squares, it happens sometimes that some of them may be negative. This is the case with the woodmouse data:

```
> trw <- nj(dist.dna(woodmouse))
> which(trw$edge.length < 0)
[1] 5
> trw$edge.length[5]
[1] -3.160774e-05
```

Usually, these negative branch lengths are tiny (the mean branch length of `trw` is 2.43×10^{-3}), and it seems reasonable to consider them as equal to zero if needed in subsequent analyses [102, 172].

The properties of NJ have been extensively studied in the literature [e.g., 100, 103, 273]. Particularly, NJ has good statistical properties when distances are unbiased.

5.1.5 Extensions of Neighbor-Joining: UNJ and BIONJ

We have seen above that clustering methods differ essentially in the way the distance matrix is updated after two observations have been clustered. Similarly, the NJ method may vary with respect to the way distances are updated after a splitting step. In the original version of the method, the new distances are calculated with:

$$d_{ui} = \frac{d_{xi} + d_{yi} - d_{xy}}{2} \qquad i \neq x, y \ ,$$

where x and y are the two neighbors, and u is the new node to be added in the matrix. During the first iteration of the NJ algorithm, x and y are original observations (species, populations, ...) but in the subsequent iterations they can be nodes created during the previous steps and so representing more than one observations. The unweighted neighbor-joining (UNJ) takes this into account by correcting the above formula with the number of observations (i.e., tips) in x and y [99]. It is unweighted in the sense that all observations are given the same weight [101]. The UNJ method is implemented in the function `UNJ` in phangorn.

So far, we have ignored the variances and covariances of the distances which are considered as independent and with equal variance. This is unlikely to be generally true because of the shared evolutionary history which makes distances non-independent (a theme we will revisit extensively in the next chapter), and the longer distances will tend to have higher variances. However, computing the variances and covariances of the distances is computationally complicated. Gascuel [98] proposed an approximation in the case where distances are from molecular sequences: the variances are then calculated with the distances read in the matrix divided by the sequence length. These approximations are used to calculate d_{ui} in a way to minimize the variance of the distances. This is the BIONJ method, and implemented in the function `bionj` in `ape`.

5.1.6 Minimum Evolution

Desper and Gascuel [55] developed a method that aims to alleviate some of the limitations of NJ. Because their method is based on the principle of minimum evolution, and it is faster than others, the authors named it FastME. They developed two versions of it. In the OLS (ordinary least squares) version, a

tree is first built agglomeratively using a fast algorithm. In a second step, a method of tree rearrangements based on nearest-neighbors interchange (NNI; Fig. 5.2) is applied until the shortest tree has been found. These two steps do not require the calculation of the branch lengths—in contrast to NJ which does so at each iteration. Finally, branch lengths are estimated on the final tree by least squares. This method is implemented in the function `fastme.ols` in `ape`:

```
fastme.ols(X, nni = TRUE)
```

The second version of FastME is based on a "balanced" approach to computing distances. This uses a formula established by Pauplin [236] to calculate the length of a tree without requiring to calculate its branch lengths. The procedure is otherwise close to the OLS version. However, the balanced approach allows to use more complex tree rearrangements, namely subtree pruning and regrafting (SPR), and tree bisection and reconnection (TBR).[2] This is implemented in `fastme.bal` also in `ape`:

```
fastme.bal(X, nni = TRUE, spr = TRUE, tbr = TRUE)
```

Desper and Gascuel wrote that FastME is faster than NJ [55]. In practice with the present implementation, however, `fastme.ols` is faster than `nj` while `fastme.bal` is slower, particularly if TBR rearrangements are used. Nevertheless, the overall computing time of the latter remains reasonable even with a large data set (≈ 15 s with 500 observations on a modern laptop). On the other hand, Desper and Gascuel claimed a greater accuracy of the balanced version of FastME compared to NJ, which should make this method an attractive alternative to NJ with moderate-sized data sets.

We have seen repeatedly the importance of least squares in distance methods. These may be formally defined with:

$$\sum_{i<j}(d_{ij} - t_{ij})^2 \,, \tag{5.1}$$

where i and j are the observations, d_{ij} is as before, and t_{ij} is the distance from the tree. This quantity is used for estimation in several methods, but it can also be used as a measure of the goodness-of-fit of the estimated tree. The generic function `cophenetic` returns the distances from a tree: it is then possible to compare them with the original distances. For instance, taking the woodmouse data and estimating an NJ tree:

```
d <- dist.dna(woodmouse)
tr.nj <- nj(d)
dt.nj <- cophenetic(tr.nj)
```

[2] At the time of writing of this book, these improvements are not yet published in the literature (Olivier Gascuel, personal communication).

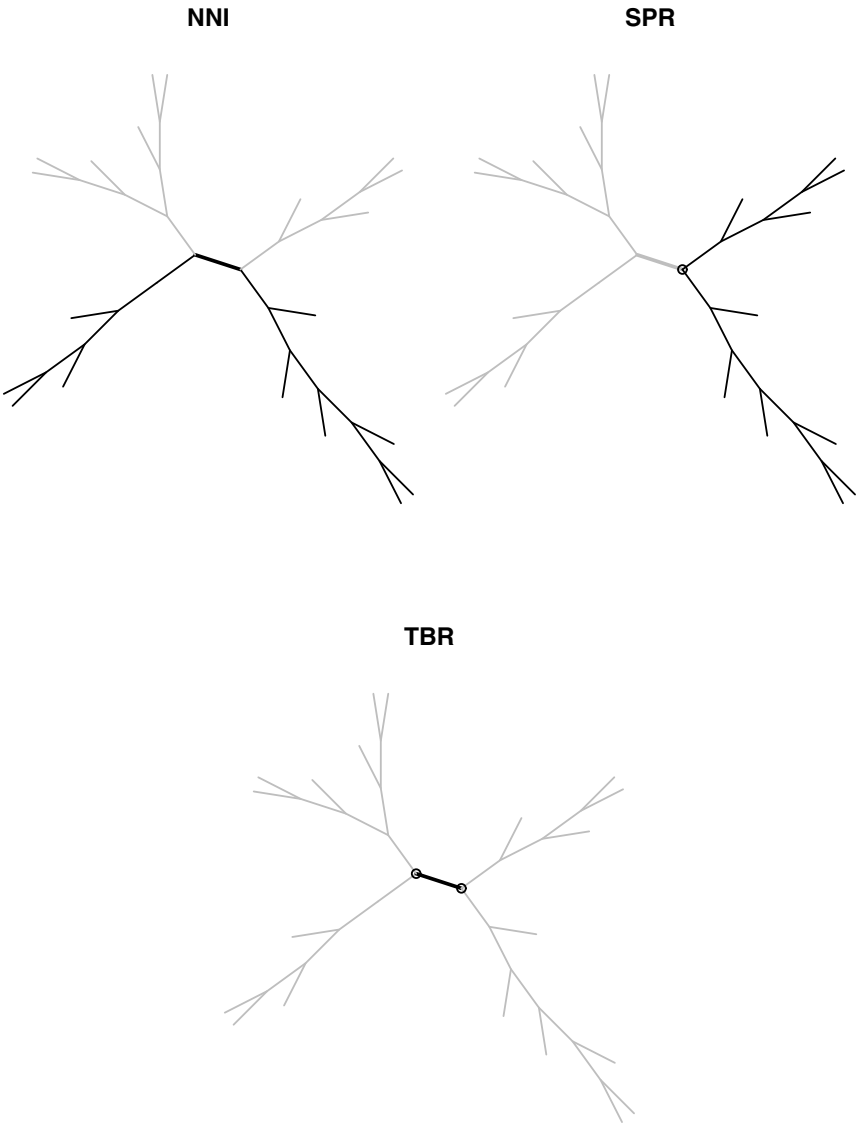

Fig. 5.2. The three common tree rearrangement operations consider an internal branch (in thick line). NNI: the two grey subtrees are exchanged; there are two possible NNIs for each internal branch. SPR: the grey subtree is moved to an internal branch of the black subtree; the circled node is deleted and a new one is created where the grey subtree is regrafted. TBR: both grey subtrees are moved in the same way than in SPR; the two circled nodes are deleted and a new one is created on each subtree

We have to insure that the distances are ordered in the same way in `d` and in `dt.nj`. The latter is a matrix—not an object of class `"dist"`—which helps to reorder its rows and columns:

```
dmat <- as.matrix(d)
nms <- rownames(dmat)
dt.nj <- dt.nj[nms, nms]
dt.nj <- as.dist(dt.nj)
```

We have converted `dt.nj` as a `"dist"` object to have each distance only once and remove the diagonal of zeros. There are several ways to plot these two sets of distances, for instance:

```
plot(dt.nj - d, ylab = "distance residuals")
abline(h = 0, lty = 3)
```

to display something similar to a plot of residuals in a regression analysis. This is done using four tree estimation methods in Fig. 5.3. UPGMA shows a clear larger dispersion of these 'residuals' compared to the three other methods. Interestingly, BIONJ shows a more compact dispersion around the $y = 0$ line of most points, though the extreme points are similar to NJ and FastME.

Note that regression residuals are assumed to be independent; however, we cannot make the same assumption for the points in Fig. 5.3 because they are calculated using pairwise distances.

It is simple from the above code lines to build functions to ease this kind of goodness-of-fit diagnostics, or even to propose other kinds of plots. The reader is invited to explore these ideas in the Exercices at the end of this chapter.

5.2 Maximum Likelihood Methods

Maximum likelihood is the cornerstone of modern statistics [61, 67]. Applying this estimation method to phylogenetic problems is simple in principle. Suppose we have data on a trait from three species, one mammal, one bird, and one fish. With a statistical model of the evolution of this trait and the phylogeny ((mammal,bird),fish), it is possible to compute the probabilities of the trait values on the tree. There are two big difficulties though. First, choosing a model of evolution seems to have a particular impact on the result of phylogeny estimation which raises the question of what is a "good" model of trait evolution. Second, it is rare that we know the tree of the species or sequences under study, and since this is the question of interest to most researchers this has attracted considerable attention.

Before diving in these two issues, the first section gives the bases of substitution models as applied to molecular sequences which are the most widely used data for phylogeny estimation.

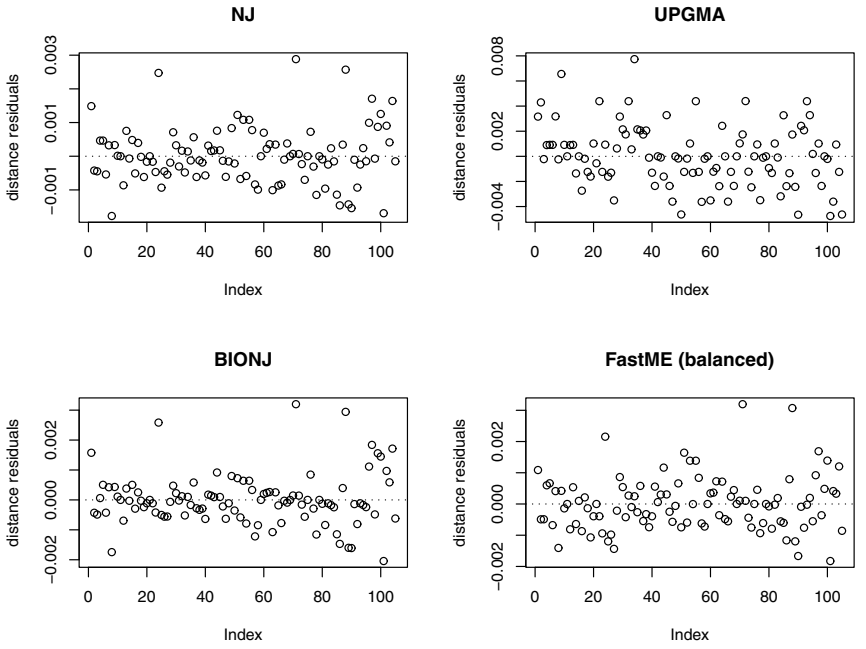

Fig. 5.3. Plot of the difference between the distances from the estimated tree and original distances (distance residuals) for four tree estimation methods

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, but the recent rise of large-scale molecular databases has led to this approach being applied essentially to nucleotide (DNA) and amino acid (protein) sequences.

A substitution model is a formulation of the instantaneous rate(s) 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:

$$Q = \begin{bmatrix} -0.1 & 0.1 \\ 0.1 & -0.1 \end{bmatrix} . \tag{5.2}$$

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:

$$P = \mathrm{e}^{tQ} \ . \tag{5.3}$$

The element p_{ij} from the ith row and jth column of P is the probability of being in state j after time t given 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:

$$\mathrm{e}^{tQ} = I + tQ + \frac{(tQ)^2}{2!} + \frac{(tQ)^3}{3!} + \cdots \tag{5.4}$$

$$= I + \sum_{i=1}^{\infty} \frac{(tQ)^i}{i!} \ . \tag{5.5}$$

In practice, an approximation is done. `ape` has the function `matexpo`:

```
> 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
> matexpo(Q) # t = 1
           [,1]       [,2]
[1,] 0.90936538 0.09063462
[2,] 0.09063462 0.90936538
> matexpo(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 could be asymmetric if Q were.

With longer time t, the probabilities in P tend to be equal. Indeed if $t = 1000$, all probabilities would be equal to 0.5 in the above example. This is an important property of Markovian models: the final state is independent of the initial state and its probabilities are equal to the equilibrium frequencies of the states. Here we assumed that these frequencies are balanced, but it does not need to be always true: in that case, the transition rates must be multiplied by these frequencies. For instance, if the equilibrium frequencies of A and B are 0.95 and 0.05, Q and P become:

```
> Q <- matrix(c(-0.005, 0.095, 0.005, -0.095), 2)
> Q
       [,1]   [,2]
[1,] -0.005  0.005
[2,]  0.095 -0.095
> matexpo(Q) # t = 1
           [,1]        [,2]
[1,] 0.99524187 0.004758129
[2,] 0.09040445 0.909595547
> matexpo(Q*1000) # t = 1000
     [,1] [,2]
[1,] 0.95 0.05
[2,] 0.95 0.05
```

When fitting a substitution model to some data, its parameter(s) will usually be unknown. For the hypothetical two-state character we write:

$$Q = \begin{bmatrix} \cdot & \alpha \\ \alpha & \cdot \end{bmatrix}, \tag{5.6}$$

where α is the parameter and the dots on the diagonal indicate that these values are set so that the rows sum to zero. There may be other parameters, for instance, if the equilibrium frequencies are to be estimated as well, there would be one additional parameter in Q:

$$Q = \begin{bmatrix} \cdot & (1 - f_A)\alpha \\ f_A\alpha & \cdot \end{bmatrix}.$$

This is generalized to DNA sequences (by assuming that Q is 4×4), or to protein sequences (20×20). 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 various packages.

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 [79, p. 156]. Substitution models are used to calculate distances between pairs of sequences (p. 128) in which case the parameters may not be estimated [317]. When these models are used in phylogeny estimation, their parameters are estimated from the data (see details below). We focus on the structure of the models and give pointers to the literature for the formulae (e.g., [317] for a thorough treatment).

Jukes and Cantor 1969 (JC69)

This is the simplest model of DNA substitution [155]. 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:

$$\begin{array}{c} \\ \mathrm{A} \\ \mathrm{G} \\ \mathrm{C} \\ \mathrm{T} \end{array} \begin{array}{c} \begin{array}{cccc} \mathrm{A} & \mathrm{G} & \mathrm{C} & \mathrm{T} \end{array} \\ \begin{bmatrix} . & \alpha & \alpha & \alpha \\ \alpha & . & \alpha & \alpha \\ \alpha & \alpha & . & \alpha \\ \alpha & \alpha & \alpha & . \end{bmatrix} \end{array} .$$

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 [79]):

$$p_{ab}(t) = (1 - \mathrm{e}^{-4\alpha t})/4 \qquad a \neq b \,, \tag{5.7}$$

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 ($\hat{t}$).

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 [161] 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.

Felsenstein 1981 (F81)

Felsenstein [74] 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.

Kimura 1981 (K81)

Kimura [162] 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.

$$\begin{bmatrix} . & \alpha & \beta & \gamma \\ \alpha & . & \gamma & \beta \\ \beta & \gamma & . & \alpha \\ \gamma & \beta & \alpha & . \end{bmatrix}.$$

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 [83] gave formulae for the probability matrix and the distance.

Hasegawa, Kishino, and Yano 1985 (HKY85)

This model is very close in essence to the previous one but its parameterization is different [129]:

$$\begin{bmatrix} . & \alpha\pi_G & \beta\pi_C & \beta\pi_T \\ \alpha\pi_A & . & \beta\pi_C & \beta\pi_T \\ \beta\pi_A & \beta\pi_G & . & \alpha\pi_T \\ \beta\pi_A & \beta\pi_G & \alpha\pi_C & . \end{bmatrix}.$$

Due to some mathematical properties of this rate matrix, it is not possible to derive analytical formulae of the transition probabilities, and so for the distance as well [314].

Tamura 1992 (T92)

The model developed by Tamura [296] 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 [296] gave formulae for the distance, and Galtier and Gouy [94] gave formulae for the transition probabilities.

Tamura and Nei 1993 (TN93)

Tamura and Nei [297] 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 $\alpha_R/\pi_R = \alpha_Y/\pi_Y$ results in the HKY85 model [79].

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 [176]. 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. Nonetheless, it is possible to calculate a distance with [317]:

$$\hat{t} = -\text{trace}\{\hat{\Pi} \ln(\hat{\Pi}^{-1}\hat{F})\}, \tag{5.8}$$

where $\hat{\Pi}$ is a diagonal matrix with the observed base frequencies, $\hat{F}$ is a 4×4 matrix with the observed relative frequencies of pairs of nucleotides, and 'ln' is the matrix logarithm calculated as $\ln(X) = VUV^{-1}$ with V the matrix of

eigenvalues of X and U a diagonal matrix with the logarithm of the eigenvalues of X. This is not implemented in a phylogenetic package, but it can computed using standard R commands and this is the subject of an exercise at the end of this chapter.

It appears from the structure of the GTR model in the above matrix that a number of intermediate models, albeit still time-reversible, can be defined by constraining the six substitution rates $(\alpha, \dots)$ and / or the four base frequencies $(\pi_A, \dots)$. phangorn implements twenty three of these models in its function `modelTest`: these include six of the models described above (F84 and T92 being not included in phangorn's scheme; Table 5.4, p. 152).

Barry and Hartigan 1987, LogDet, and Paralinear

More complex models than the GTR can be built by removing the constraint of time-reversibility: the most general model would thus have twelve substitution rates—plus eventually the three frequency parameters. The models though are difficult to analyze with real data, and their general usefulness in data anlysis is not clear [79, 317].[3]

Barry and Hartigan developed a formula to compute the distance from a general model [21]. They claim this distance is valid even when the model is non-stationary (i.e., the rates are not assumed to be constant through time). The distance matrix is not symmetric, for instance examing the six first columns and rows with the woodmouse data:

```
> round(dist.dna(woodmouse, "BH87")[1:6, 1:6], 4)
         No305  No304  No306 No0906S No0908S No0909S
No305   0.0000 0.0174 0.0165  0.0230  0.0182  0.0179
No304   0.0136 0.0000 0.0041  0.0169  0.0119  0.0155
No306   0.0125 0.0039 0.0000  0.0127  0.0077  0.0113
No0906S 0.0148 0.0125 0.0085  0.0000  0.0101  0.0137
No0908S 0.0158 0.0132 0.0092  0.0158  0.0000  0.0146
No0909S 0.0160 0.0175 0.0135  0.0201  0.0152  0.0000
```

The level of asymmetry is visible to the third digit. This clearly leads to the virtual impossibility to use the Barry–Hartigan distance in distance-based methods. Lake [174] and Lockhart et al. [186] derived variants of the Barry–Hartigan distance that are symmetric: the LogDet and paralinear distances, respectively. These three distances are available in `dist.dna`.

Galtier and Gouy 1995

Galtier and Gouy [93] developed a nonequilibrium model where the G + C content is allowed to change through time. Sequences are assumed to evolve

[3] Rodríguez et al. [269] showed that models that are more complex than GTR do not have an estimable distance with pairwise data.

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`.

Amino Acid Models

Substitution models applied to amino acid sequences use a radically different approach than the one used for nucleodite data: they are based on empirical rate matrices and amino acid frequencies derived from databases of protein sequences so that the rate matrix Q is given, and no parameter has to be estimated. Various authors established different versions of Q; nine of them are available in phangorn for phylogeny estimation (Table 5.3). Additionally, the function `dist.ml` computes distances for aligned amino acid sequences under these models.

Table 5.3. Models of amino acid sequence evolution available in phangorn

Model	Description	Ref.
Dayhoff	General model developed in the late 1970's	[50]
JTT	Improved version of the above	[153]
WAG	General improved model with a maximum likelihood approach	[311]
LG	Improved version of the above	[178]
cpREV	Model for amino acid sequences from chloroplast genes	[4]
mtmam	Model for mitochondrial proteins of mammals	[318]
mtArt	Model for proteins of arthropods	[1]
MtZoa	General model for mitochondrial proteins of animals	[271]
mtREV24	General model developed in the late 1990's	[3]

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 [74] 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

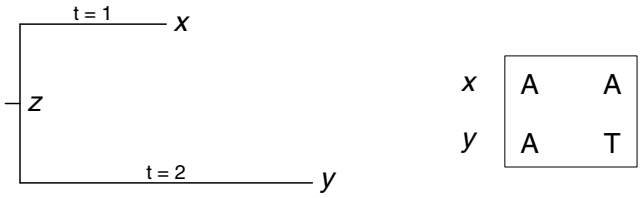

Fig. 5.4. A tree with tips x and y (left) and a matrix with two nucleotides (right)

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:

$$L_{az} = \left(\sum_{b=1}^{M} p_{ab}(t_{xz})L_{bx}\right)\left(\sum_{b=1}^{M} p_{ab}(t_{yz})L_{by}\right) . \tag{5.9}$$

If x is a tip, then $L_{bx} = 1$ if state b is observed, 0 otherwise. This equation looks complicated but its logic is apparent with a very simple example. Figure 5.4 shows a tree (or a part of a bigger tree) with two tips, x and y, where two nucleotides have been observed: the first site is the same for both tips (A) while the second site is polymorphic. To make things simple, we shall use the Jukes–Cantor model with $\alpha = 0.05$. So we only have to use (5.7) to build the probability matrices; for the branch between x and z with $t_{xz} = 1$:

```
> pab <- (1 - exp(-4 * 0.05))/4
> Px <- matrix(pab, 4, 4)
> diag(Px) <- 1 - 3 * pab
> Px
          [,1]       [,2]       [,3]       [,4]
[1,] 0.86404806 0.04531731 0.04531731 0.04531731
[2,] 0.04531731 0.86404806 0.04531731 0.04531731
[3,] 0.04531731 0.04531731 0.86404806 0.04531731
[4,] 0.04531731 0.04531731 0.04531731 0.86404806
```

The calculations are the same for the branch between y and z except that $t_{yz} = 2$: this creates the matrix `Py` (not shown). Consider the first site: the datum is the same for x and y, so we create two identical vectors with the appropriate 1 and 0's:

```
> Lx <- Ly <- c(1, 0, 0, 0)
```

We can compute (5.9) now. Below are two ways to do this: the first command, using a `for` loop, has the advantage of displaying clearly its relationship with the mathematical equation, but the second one, using R recycling rule, is more efficient from all points of view:

```
> # for (i in 1:4) print(sum(Px[i, ]*Lx) * sum(Py[i, ]*Ly))
> colSums(Px * Lx) * colSums(Py * Ly)
[1] 0.650403570 0.003735052 0.003735052 0.003735052
```

Not surprisingly, the likelihood of z having A on this first site is the highest, but the likelihoods of the other states are not zero.

Now consider the second site. The datum is the same for x, so we change only `Ly`. The probability matrices are unchanged (the branch lengths and α are the same), so calculation proceeds exactly as before:

```
> Ly <- c(0, 0, 0, 1)
> colSums(Px * Lx) * colSums(Py * Ly)
[1] 0.071214832 0.003735052 0.003735052 0.034112155
```

Further opportunities to discuss these calculations are provided in the Exercises.

Once this computation has been applied to all nodes of the tree, the likelihood of the character for the tree is obtained by:

$$L = \sum_{a=1}^{M} \pi_a L_{ar} \ ,$$

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 [74]. The likelihood of the full data set is:

$$L = \prod_{i=1}^{N} \sum_{a=1}^{M} \pi_a L_{air} \ ,$$

where N is the number of characters. Taking the logarithm of this expression leads to:

$$\ln L = \sum_{i=1}^{N} \ln \left(\sum_{a=1}^{M} \pi_a L_{air} \right) \ . \tag{5.10}$$

A further layer of complexity is added by considering heterogeneity among sites. Two types of heterogeneity are commonly considered: partitions and mixtures [247, 316]. With partitions, the different sites 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 (5.9) 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 (see the section on partitioned models below).

phangorn provides a wide array of tools to estimate phylogenies by maximum likelihood. Its main function is `pml`:

```
pml(tree, data, bf = NULL, Q = NULL, inv = 0, k = 1,
    shape = 1, rate = 1, model = "", ...)
```

where `tree` is an object of class `"phylo"` and `data` is an object of class `"phyDat"`. Recall that this data class may store several kinds of discrete data: nucleotides, amino acids, or user-defined. By default the frequencies of the states are taken as equal unless `bf` is used; it is the same for the rate matrix unless `Q` is used. The three next arguments parameterize inter-site variation in substitution rate as commonly used in phylogenetic analyses [315]: `inv` is the proportion of invariant sites, `k` is the number of categories with different substitution rates, and `shape` is the parameter of the Γ distribution. These two forms of inter-site variation are mixtures because we do not know the category each site belongs to. When the data are amino acids, `model` specifies the substitution model. Finally, '`...`' is used to pass some arguments to other functions.

`pml` calculates the likelihood of the data given the tree and the model specified. Coming back to our two-tip, two-nucleotide data above, we can compute its likelihood as we did by hand:

```
> tr <- read.tree(text = "(y:2,x:1);")
> x <- matrix(c("a", "a", "a", "t"), 2, 2)
> rownames(x) <- c("x", "y")
> x <- as.phyDat(x)
> pml(tr, x, rate = .05/.25)

 loglikelihood: -5.288237

unconstrained loglikelihood: -1.386294

Rate matrix:
  a c g t
a 0 1 1 1
c 1 0 1 1
g 1 1 0 1
t 1 1 1 0

Base frequencies:
```

```
0.25 0.25 0.25 0.25
```

If we use (5.10) above we find -5.368 which is quite close but not equal. This is because `pml` uses standard formulae for all models and parameterizations. The rate parameters in Q are multiplied by the base frequencies before computing the probability matrix: this is why we used `rate = .05/.25` (i.e., α/π).

The "unconstrained loglikelihood" is a measure of the maximally achievable value of the log-likelihood—though it is often smaller than the log-likelihood of the data as shown below—and is independent of the tree.

`pml` does not estimate the parameters in the sense that it does not find their values that maximize the likelihood function. This is done with `optim.pml`:

```
optim.pml(object, optNni=FALSE, optBf=FALSE, optQ=FALSE,
   optInv=FALSE, optGamma=FALSE, optEdge=TRUE, optRate=FALSE,
   control=pml.control(eps=1e-08, maxit=10, trace=1),
   model=NULL, subs=NULL, ...)
```

`object` is an output from `pml`, and the arguments of the form `opt*` specify which parameters to estimate. Very logically, we cannot estimate the parameters in our two-tip example, so we take the woodmouse data and its NJ tree:

```
> tw <- nj(dist.dna(woodmouse))
> x <- as.phyDat(woodmouse)
> o1 <- pml(tw, x)
Warning message:
In log(siteLik) : NaNs produced
> o2 <- optim.pml(o1)
optimize edge weights:  -1867.307 --> -1857.168
optimize edge weights:  -1857.168 --> -1857.168
optimize edge weights:  -1857.168 --> -1857.168
> o2

 loglikelihood: -1857.168

unconstrained loglikelihood: -1866.991

Rate matrix:
  a c g t
a 0 1 1 1
c 1 0 1 1
g 1 1 0 1
t 1 1 1 0

Base frequencies:
0.25 0.25 0.25 0.25
```

(The warning message issued after the first call of `pml` is caused by the fact that `tw` has one branch length negative; this problem is solved once we estimated them by maximum likelihood.) We used the default options of `optim.pml`, so only the branch lengths have been estimated. Thus we effectively used the JC69 model. The log-likelihood has been increased from −1867 to −1857. *We have changed the parameter values in order to maximize the likelihood.* This is the normal estimation procedure and there are no reason to compare the two values here. Now let us try to estimate the substitution rate together with the branch lengths:

```
> ctr <- pml.control(trace = 0)
> optim.pml(o1, optRate = TRUE, control = ctr)

 loglikelihood: -1857.168
....

Warning message:
you can't optimise edges and rates at the same time,
 only edges are optimised
```

The warning issued by `phangorn` is clear: the substitution rate α cannot be identified separately from the branch lengths. Of course, we could estimate the rate if we assume the branch lengths to be known:

```
> o3 <- optim.pml(o1, optRate=TRUE, optEdge=FALSE, control=ctr)
> o3$rate
[1] 1.105272
```

Thus using the branch lengths as estimated by NJ, the maximum likelihood estimate is $\hat{\alpha} = 1.105$. However, the branch lengths estimated by NJ have also time and substitution rate confounded; such estimates of rates are more useful when branch lengths are in absolute time units.

`optim.pml` allows us to define the model by specifying which parameters are to be estimated. To see this, let us fit the GTR model:

```
> o4 <- optim.pml(o1, optBf=TRUE, optQ=TRUE, control=ctr)
> o4

 loglikelihood: -1756.274

unconstrained loglikelihood: -1866.991

Rate matrix:
             a         c         g            t
a 0.000000e+00  1.455591 22.192215 5.133905e-05
c 1.455591e+00  0.000000  2.332634 1.964957e+01
g 2.219222e+01  2.332634  0.000000 1.000000e+00
```

Table 5.4. Nucleotide substition models in phangorn in order of decreasing complexity. The second column shows how rates of the GTR model are reduced to simplify the models

Model	Substitution rates	Base frequencies
GTR	$\alpha, \beta, \gamma, \delta, \epsilon, \zeta$	Unbalanced
SYM	"	Balanced
TVM	$\alpha = \zeta$	Unbalanced
TVMe	"	Balanced
TIM3	$\beta = \delta, \gamma = \epsilon$	Unbalanced
TIM3e	"	Balanced
TIM2	$\beta = \gamma, \delta = \epsilon$	Unbalanced
TIM2e	"	Balanced
TIM1	$\beta = \epsilon, \gamma = \delta$	Unbalanced
TIM1e	"	Balanced
TPM3u	$\alpha = \zeta, \beta = \delta, \gamma = \epsilon$	Unbalanced
TPM3	"	Balanced
TPM2u	$\alpha = \zeta, \beta = \gamma, \delta = \epsilon$	Unbalanced
TPM2	"	Balanced
TPM1u	$\alpha = \zeta, \beta = \epsilon, \gamma = \delta$	Unbalanced
K81 (TPM1)	"	Balanced
TrN	$\beta = \gamma = \delta = \epsilon$	Unbalanced
TrNe	"	Balanced
HKY	$\alpha = \zeta$, $\beta = \gamma = \delta = \epsilon$	Unbalanced
K80	"	Balanced
F81	$\alpha = \beta = \gamma = \delta = \epsilon = \zeta$	Unbalanced
JC	"	Balanced

```
t 5.133905e-05 19.649572  1.000000 0.000000e+00

Base frequencies:
0.3042873 0.2662815 0.1288796 0.3005517
```

What we have done now is very different from above: *We have added parameters in the model: this will necessarily increase the likelihood.* This increase is because we gave more freedom to the model to fit the data by adding parameters. A statistical test may tell us whether this increase is significant or not. Instead of rushing into the tests, let us have a look at the rate matrix. The diagonal is set to 0, but this is arbitrary. Like for the JC69 model, rates cannot be identified separately from the edge lenths, however, if there are several rates their relative values may be estimated. So here the rate of the substituion G $\leftrightarrow$ T is fixed and, e.g., the rate of A $\leftrightarrow$ G is estimated to be 22 times higher.

It is critical to examine such outputs before doing the tests, because they give more than useful information on the processes. Now, we may wonder if the increase in log-likelihood from -1857 to -1756 is statistically significant.

We know that twice this difference in log-likelihood follows a χ^2 with degrees of freedom given by the number of additional parameters (this is a likelihood ratio test, LRT). We have added eight parameters (five rates and three base frequencies), so the LRT is $\chi^2_8 = 202$. The P-value may be calculated with `1 - pchisq(202, 8)`. Getting these differences correctly may be sometimes tedious, and the generic function `anova` is helpful here:

```
> anova(o2, o4)
Likelihood Ratio Test Table
  Log lik. Df Df change Diff log lik. Pr(>|Chi|)
1  -1857.2 27
2  -1756.3 35         8        100.89  < 2.2e-16
```

It is left to the user to insure that the correct models are passed to `anova`. For instance, it would not be meaningful to do `anova(o1, o4)` because the branch lengths were not estimated by maximum likelihood in `o1`.

Now we incorporate inter-site variation with the usual Γ distribution: we have to redefine a new model with `pml` and estimate α:

```
> o5 <- optim.pml(pml(tw, x, k = 4), optBf=TRUE, optQ=TRUE,
+                 optGamma = TRUE, control = ctr)
> o5

 loglikelihood: -1743.367

unconstrained loglikelihood: -1866.991
Discrete gamma model
Number of rate categories: 4
Shape parameter: 0.004467235

Rate matrix:
             a         c         g            t
a 0.000000e+00  1.476878 23.294362 6.595230e-05
c 1.476878e+00  0.000000  2.566102 2.010049e+01
g 2.329436e+01  2.566102  0.000000 1.000000e+00
t 6.595230e-05 20.100486  1.000000 0.000000e+00

Base frequencies:
0.3054005 0.2646932 0.1279518 0.3019545
```

The fact that substitution rates vary substantially within coding sequences has been widely documented in the literature, so we anticipate the estimate of the shape parameter $\hat{\alpha}_\Gamma = 0.0045$ (the subscript Γ is to avoid confusion with the substitution rate α) to be statistically significant. Note that if $\alpha_\Gamma \to \infty$, then there is no intersite variation in substitution rate.

```
> anova(o4, o5)
```

```
Likelihood Ratio Test Table
  Log lik. Df Df change Diff log lik. Pr(>|Chi|)
1  -1756.3 35
2  -1743.4 38         3        25.820  1.040e-05
```

There are many other models that we may fit to these data. The choice of the fitted models must be driven by biological questions—as far as the evolutionary biologist is concerned. Figure 5.14 and Table 5.5 give general guidelines on how to conduct phylogeny estimation.

Partitioned Models

`pmlPart` is the function in `phangorn` to fit partitioned models. It is used with `pmlPart(`*`global ~ local`*`, object, ...)` with *`global`* being the components of the model that are common across partitions, *`local`* are the components specific to each partition, and `object` is an object of class `"pml"`. For instance, `pmlPart(edge ~ shape, obj)` will estimate branch lengths common to all partitions and an α_Γ specific to each one. Only the parameters whose components are given in this formula are estimated by `pmlPart`. Before calling the function, we have to specify whose partition each site belongs to. The procedure is slightly tedious, so we detail it here. For the woodmouse data it makes sense to define three partitions with the three codon positions. We recall that the 965 sites define 65 patterns across the 15 individuals:

```
> x
15 sequences with 965 character and 65 different site patterns
The states are a c g t
```

It may be useful to remind that the site patterns correspond to the unique columns of the original alignment, and two sites with the same pattern have exactly the same contribution to the likelihood function, so we have to compute (5.10) for $N = 65$ sites instead of 965. The attribute `"index"` keeps track of the pattern each site belongs to:

```
> str(attr(x, "index"))
 num [1:965] 1 2 2 3 4 5 5 5 5 5 ...
```

So the second and third columns in `woodmouse` have the same pattern (i.e., they are identical) which is the second one in `x`, while the first, fourth, and fifth columns define different patterns, etc. We have now to make a contingency table dispatching the 65 site patterns in the three partitions defined by the three codon positions. This can be done with `xtabs` as illustrated in the help page `?pmlPart`, or more directly with `table`:

```
> w <- table(attr(x, "index"), rep(1:3, length.out = 965))
> dim(w)
[1] 65  3
```

```
> w[11:15, ]

      1   2   3
  11  0   0   1
  12  0   0   1
  13 82  60 120
  14 76 125  59
  15 70  73  82
```

This shows that, for instance, the eleventh and twelveth patterns were observed only once on a third position, whereas the thirteenth pattern was observed 82, 60, and 120 times on each position, and so on for the 65 rows of `w`. This table is used as weights in the call to `pmlPart`:

```
> o6 <- pmlPart(~ rate, o2, weight = w, control = ctr)
> o6

loglikelihood: -1827.652

loglikelihood of partitions:
  -530.835 -490.0756 -806.7415
AIC:  3713.304  BIC:  3854.596

Proportion of invariant sites: 0 0 0

Rates:
0.4397931 0.2189536 2.345432
....
```

The improvement in log-likelihood is nearly 30 units for two additional parameters, so this is surely very significant:

```
> pchisq(2*(1857.168 - 1827.652), 2, lower.tail = FALSE)
[1] 1.518323e-13
```

The estimated rates agree with the expectation that the second codon positions are those evolving at the slowest rate while the third ones are the fastest ones.

We have shown in the previous section that the GTR + Γ model fits better to these data than the simple JC69, so we now try this model with different rates and shape parameters (denoted α_Γ) in the three partitions:

```
> o7 <- pmlPart(~ rate + shape, o5, weight = w, control = ctr)
> o7

loglikelihood: -1714.530
```

```
loglikelihood of partitions:
  -529.0336 -473.0926 -712.4034
AIC:  3521.059  BIC:  3745.177

Proportion of invariant sites: 0 0 0

Rates:
0.3793673 0.1884981 2.436596

Shape parameter:
0.005775529 99.99474 0.2169682
....
```

The decrease in log-likelihood is again close to 30, but this time we added four parameters:

```
> pchisq(1743.367 - 1714.530, 4, lower.tail = FALSE)
[1] 8.436594e-06
```

This is again very significant, and suggests that substitution rate is more heterogeneous in the first codon position than in the third one (recall that the smaller the value of α_Γ, the more heterogeneous the substitution rate among sites). However, a more parsimonious model shows that this is not significant with the present data:

```
> o8 <- pmlPart(~ rate, o5, weight = w, control = ctr)
> o8

loglikelihood: -1715.329

loglikelihood of partitions:
  -529.0337 -473.2011 -713.0942
AIC:  3510.658  BIC:  3705.543

Proportion of invariant sites: 0 0 0

Rates:
0.3910165 0.1919223 2.421476
....
```

The increase in log-likelihood of `o7` compared to `o8` is less than one while, to be significant with two degrees of freedom, it needs to be at least three.

Mixture Models

We have seen that including intersite variation in substitution rate with a Γ distribution is a form of mixture. `pmlMix` is a function to model higher order

mixtures. Its syntax is close to the one of `pmlPart`, the differences are the option `m` specifying the number of categories (2 by default), and `omega` is a vector giving their frequencies (f_k in eq. 5.11); by default they are equal and used as starting values for the estimation procedure. Let us take the fitted object `o2` which is a simple JC69 model. We now include a mixture on its rate with four categories:

```
> o9 <- pmlMix(~ rate, o2, m = 4, control = ctr)
> o9

loglikelihood: -1845.219

unconstrained loglikelihood: -1866.991
AIC:  3750.438  BIC:  3815.670


posterior: 0.7313842 0.004928028 0.2476871 0.01600061

Rates:
3.523458e-07 0.3680481 4.030027 1.448669e-05
....
```

The substitution rate varies considerably along the sequence (remind this is cytochrome *b*) and the most frequent category is the one with the lowest rate.

Logically we expect similar results with a traditional Γ distribution. We check it by printing the log-likelihood:

```
> logLik(optim.pml(update(o2, k = 4), optGamma = TRUE,
+   control = ctr))
'log Lik.' -1845.001 (df=30)
```

The values are close but not equal because `o6` did not use a Γ distribution.

It is interesting to compare this result with the one from the partitioned model (`o8`). The likelihoods cannot be compared directly with a LRT because the models have different structures and no relationship of nestedness. However, the AIC values can be compared and show that the partitioned model fits much better. This shows that if the structure in the data can be identified and taken into account explicitly, this leads to much more powerful analyses.

5.2.3 Finding the Maximum Likelihood Tree

The previous section details how to find the parameter values, including branch lengths, that maximize the likelihood function for a given tree topology. Finding the topology that maximizes the likelihood function is a much more difficult problem. Many solutions have been proposed in the literature and not a single one seems superior, though some are more useful than others.

phangorn implements an approach similar to the one developed by Guindon and Gascuel [115] whose principle is to consider only NNIs (a tree rearrangement seen above about FastME) that improves the likelihood. Guindon and Gascuel's idea is to avoid recalculating the likelihood of the whole tree by realizing that the likelihood values at a node, as detailed above, may be computed in two directions in an unrooted tree. After performing an NNI operation, some node likelihoods in one direction will be unchanged, and others in the other direction as well: this limits considerably the calculations to obtain the complete likelihood of the new topology. Besides, there is only one possible NNI for each internal branch, so scanning the tree for all potential improvements is straightforward. These cycles of NNI moves are alternated with cycles of parameter optimizations. The whole procedure is iterated until the tree likelihood is stable. The option `optNni = TRUE` in `optim.pml` will perform this likelihood topology estimation.

phangorn has a significant improvement over PhyML in that partitions can be modeled with `pmlPart` and it is possible to estimate a different topology for each partition with `pmlPart(~ nni, ....)`, meaning that NNI search will be performed independently in each partition. This is what is often called "mixture of trees" in the literature.

Because phylogeny estimation by maximum likelihood requires intensive computations, computer programs to perform this task are the results of a lot of work. phangorn represents a significant achievement with this respect, but it makes sense to write simple interfaces between R and other applications. There already exist several of them in various packages and two of them are relevant here: `phymltest` in ape (detailed in the next section) and `raxml` in phyloch, an interface to RAxML [287].[4]

RAxML was particularly designed for the analysis of large data sets (> 1000 sequences) though it can analyse small data sets as well. It uses a tree rearrangement named lazy subtree rearrangement (LSR) where a subtree is pruned and grafted on a neighboring branch: among all the new topologies explored, only the 20 best trees are fully optimized [288]. This procedure is repeated until no better topology is found. RAxML uses an initial parsimony tree. Another particularity of this program is the use of a GTR–CAT model in place of the usual GTR + Γ: the likelihood are first calculated for each individual site which is then attributed to a category with respect to its individual likelihood contribution. This avoids recalculating the contribution of each site to each category of the Γ distribution. However, the likelihood values calculated with the GTR–CAT model are not comparable with those from traditional models, so RAxML allows one to use the GTR + Γ model on the final topology in order to estimate α_Γ.

The interface from R is:

```
raxml(seq, runs = 10, partition = NULL, outgroup = NULL,
      backbone = NULL, path = "/Applications/RAxML-7.0.4",
```

[4] http://icwww.epfl.ch/~stamatak/index-Dateien/Page443.htm

```
        optimize = FALSE, clear = TRUE, file = "fromR")
```

The data `seq` must be of class `"DNAbin"`. The option `partition` allows to define a partitioned model. The call of `raxml` creates a subdirectory in the directory defined by the argument `path` where the outputs of RAxML are stored and returns a tree of class `"phylo"`:

```
> MLtree <- raxml(woodmouse, path = "~/RAxML-7.0.4/")
....
Overall Time for 10 Inferences 6.447887
Average Time per Inference 0.644789
Average Likelihood   : -1743.700212

Best Likelihood in run number 8: likelihood -1743.528033
....
```

The returned log-likelihood (and not likelihood as printed by RAxML) is very close to the one obtained with `optim.pml` (see `o5`, p. 153).

5.2.4 DNA Mining with PhyML and `modelTest`

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 [34, 35, 48, 204]. The importance of model selection in a likelihood framework has been made repeatedly in the phylogenetic literature [206, 246, 244]. Posada and Crandall [245] 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.

In order to provide a similar functionality, but with a free phylogeny estimation program, `ape` has the function `phymltest` which uses PhyML developed by Guindon and Gascuel [115, 139].[5] Another difference is that `phymltest` lets PhyML search for the best tree using NNI moves 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 = NULL, append = TRUE)
```

where `seqfile` is the name of the file with the sequences (given as a character). The other arguments have default values: `itree` is the initial tree (by default, a BIONJ tree made by PhyML), and `execname` is the name of the PhyML

[5] http://www.atgc-montpellier.fr/phyml/

executable which depends on the operating system (phyml_3.0.1_linux32, phyml_3.0.1_macintel, or phyml_3.0.1_win32.exe). Under Unix-style systems, it is convenient to create a symbolic link named 'phyml', for instance in your home directory, which can be subsequently modified when a new version of PhyML is issued; thus it is not needed to change the R scripts which have `execname = "~/phyml"`.

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:

```
phymltest("/home/paradis/data/seq.txt",
          execname = "phyml", path2exec = "/usr/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 5.8 for an example).

phangorn has a similar function, `modelTest`, that fits a series of models with its own functions. The set of default models is slightly different: JC69, F81, K80, HKY85, SYM, GTR, and includes intersite variation with four categories and invariants. These may be controlled with the options `model`, `G`, `I` (both logical), and `k` for the number of categories if `G = TRUE`. `modelTest` returns a data frame with the values of log-likelihood, AIC and BIC (Bayesian Information Criterion). For a simple example with the woodmouse data:

```
> modelTest(x, G = FALSE, I = FALSE)
  Model df    logLik      AIC      BIC
1    JC 27 -1857.165 3768.330 3899.878
2   F81 30 -1810.958 3681.916 3828.080
3   K80 28 -1806.798 3669.596 3806.016
```

```
4   HKY 31 -1759.459 3580.918 3731.954
5   SYM 32 -1803.493 3670.986 3826.894
6   GTR 35 -1756.274 3582.548 3753.073
```

The AIC and BIC values agree that the HKY85 model is the best one for these data, closely followed by GTR, while the four other models do not fit appropriately.

5.3 Bayesian Methods

The methods we have used so far consider that data are produced by a random process whose unknown parameters are constant; the estimates of these parameters are random variables. They are called frequentist methods. Bayesian methods do not consider parameters as constants but as probabilistic distributions. This requires to define these distributions before looking at the data: the priors.[6] Bayesian methods calculate the posterior distributions of the parameters by combining the priors with the likelihood of the data using conditional probabilities with Bayes's theorem (or Bayes rule). Such a calculation involves integrating over the priors of all parameters in the model: this integral has thus as many dimensions as there are parameters. Calculating an integral in a single dimension may be difficult, so calculating an integral in many dimensions is an arduous task. A large number of computational methods have been invented to do these calculations, the most well-known being the Markov chains Monte Carlo (MCMC).

It would take many pages to describe Bayesian phylogenetic methods and the controversy about their use (which is little compared to the same debate among statisticians), but the above paragraph summarizes the fundamental points to keep in mind when using a Bayesian method in phylogenetics, and actually in any field. For a gentle introduction to Bayesian statistics see Albert's *Bayesian Computation With R* [7].

Consider a very simple Bayesian estimation of a phylogeny. We assume a JC69 model, so there is only one substitution parameter (α). Priors must therefore be specified for the tree topology, the branch lengths, and α. We choose the followings: all topologies are equiprobable, branch lengths from a uniform distribution on [0, 0.1], and α from a uniform distribution on [0, 1]. Integrating over all the parameters is not possible with standard mathematics. A solution is to sample this multidimensional space randomly in order to have a general image of its density. The sampling procedure is repeated a large number of times (say 10^6) storing each value α and the log-likelihood (the data `x` is of class `"phyDat"`, and the number of sequences is `n`). We focus here on the estimation of α:

[6] Bayesian priors may be interpreted very differently depending on the meaning attached to them.

```
alpha <- lnL <- numeric(1e6)
for (i in 1:1e6) {
    tr <- rtree(n, rooted = FALSE, tip.label = names(x),
                br = runif, min = 0, max = 0.1)
    alpha[i] <- runif(1)
    lnL[i] <- pml(tr, x, rate = alpha[i])$logLik
}
```

We compute the posterior distribution of α for the woodmouse data with a histogram; we take care to set the x-axis on the original scale of the prior (Fig. 5.5):

```
o <- hist(alpha*lnL/sum(alpha*lnL), 50, xaxt = "n",
          main = "", xlab = expression(alpha))
axis(1, at = seq(min(o$breaks), max(o$breaks), length.out=5),
     labels = seq(min(alpha), max(alpha), length.out = 5))
```

The posterior is highest at $\alpha \approx 0.05$. It can be checked that this distribution is heavily influenced by the priors by modifying the above code.

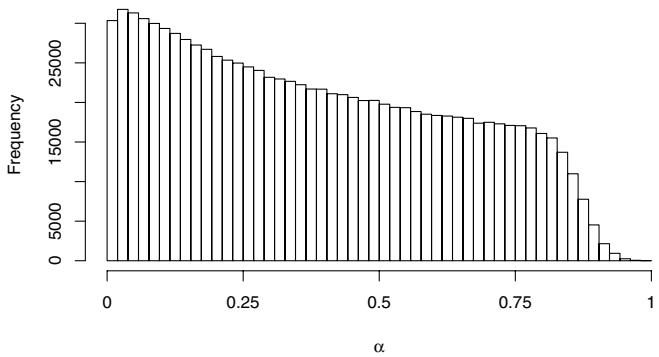

Fig. 5.5. Posterior distribution of α from 10^6 Monte Carlo generations

What we have done is a simple Monte Carlo integration assuming that the algorithm sufficiently explored the parameter space to give a picture of the posterior distribution. In most cases Monte Carlo integration is not efficient because the space is very large and its random exploration spends a lot of time in parts of low likelihood value. A significant improvement is due to Hastings [130] who proposed to avoid exploring the low-likelihood portions of the parameter space with the following rule:

- If the likelihood of the new proposal is higher than the previous one, then accept it.
- If the likelihood of the new proposal is lower, then accept it with probability equal to the ratio of the two likelihoods. If it is rejected, sample the priors for another proposal.

If we try to implement a Metropolis–Hastings version of our algorithm above, the chain would be quickly stuck because the complete random sampling procedure is highly likely to generate data far from the current tree and so will be rejected. To avoid this we sample trees in a correlated way: topologies are modified with a NNI move using the function `rNNI` from phangorn, and branch lengths are modified by adding to each of them a small noise. Still we sample from the same priors by constraining the branch lengths to be between 0 and 0.1. For the sake of simplicity, we continue to sample values of α in a non-correlated way. The code is longer this time because we must initialize the chain with a random tree, and we implement the Hastings ratio. As before, we focus on the estimation of α.

```
alpha <- lnL <- numeric(1e3)
tr <- rtree(n, rooted = FALSE, tip.label = names(x),
            br = runif, min = 0, max = 0.1)
alpha[1] <- runif(1)
lnL[1] <- pml(tr, x, rate = alpha[1])$logLik
i <- 2
while (i <= 1e3) {
    tr2 <- rNNI(tr)
    tr2$edge.length <- tr2$edge.length
          + rnorm(2*n - 3, sd = 0.1)
    tr2$edge.length[tr2$edge.length < 0] <- 0
    tr2$edge.length[tr2$edge.length > 0.1] <- 0.1
    alpha[i] <- runif(1)
    lnL[i] <- pml(tr2, x, rate = alpha[i])$logLik
    accept <- if (lnL[i] >= lnL[i - 1]) TRUE
    else as.logical(rbinom(1, size = 1,
         prob = exp(lnL[i] - lnL[i - 1])))
    if (accept) {
        tr <- tr2
        i <- i + 1
    }
}
```

We run this until we have 1000 proposal trees accepted. Figure 5.6 shows the evolution of the log-likelihood along the chain:

```
plot(lnL, type="l", xlab="Generation", ylab="Log-likelihood")
```

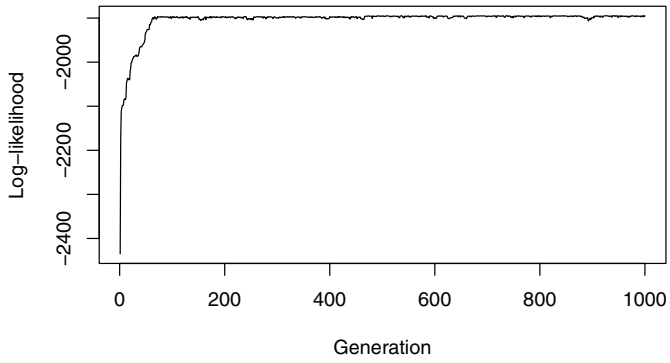

Fig. 5.6. Change in log-likelihood along an MCMC

The same plot for the simple Metropolis algorithm, not shown here, would have displayed wide variation in the log-likelihood of the sampled trees. Figure 5.7 displays the estimated posteriors for α with all samples (left) and after removing the first one hundred ones (right); the code is the same than above. This time the estimates are much more narrower and concentrated around $\alpha \approx 0.066$. In publications, these distribution estimates are plotted as a smoothed curve (see `?density` on how to smooth easily a histogram, or the package KernSmooth for more sophisticated methods).

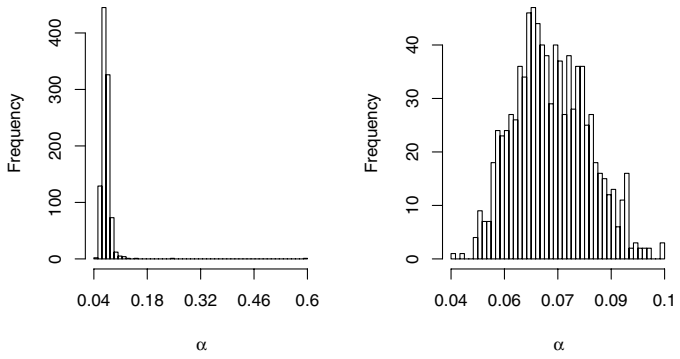

Fig. 5.7. Posterior distribution of α with all samples (left) and after removing the first one hundred ones (right)

This very simple example gives a brief view of the complexity of Bayesian estimation in a situation with many parameters which is the case in phylogenetics. Checking the impact of the priors, the convergence of the chains, the impact of the sampler used should all be done routinely, but this is a lot work. Often (too often in my opinion), a Bayesian analysis is run more to contemplate the final tree rather than to give insights into the biological processes that shaped the data. I recommend to spend a comparable effort into graphical exploratory analyses, examination of the alignment(s) and of the distribution of pairwise distances, quantification of inter-site variation in substitution rate, and assessment of support and conflict values from bootstrap trees. This list is not limitative and Fig. 5.14 gives an overview of this process.

The function `mrbayes` in phyloch provides an interface to MrBayes [142]:

```
mrbayes(x, file, nst = 6, rates = "invgamma", ngammacat = 4,
  nruns = 2, ngen = 1e+06, printfreq = 100, samplefreq = 10,
  nchains = 4, savebrlens = "yes", temp = 0.2, burnin = 10,
  contype = "allcompat", path="/Applications/mrbayes-3.1.2/",
  run = TRUE)
```

`mrbayes.mixed` has exactly the same options: it considers morphological data.

5.4 Other Methods

5.4.1 Parsimony

The parsimony principle is extremely simple: assume the least number of evolutionary changes through time.[7] For instance, let us go back to the data on Fig. 5.4 and take the first site: it is the same for x and y. So the parsimony principle leads us to say that z had A as well on this site. This reasoning involves no parameter and no branch lengths. We have seen in Section 5.2.2

[7] The principle of parsimony as it is applied to phylogenetics must be distinguished from the principle of the same name in statistics. The latter says that non-significant effects should not be included in a model. For instance, the linear model $y = \beta_1 x_1$ is preferred to $y = \beta_1 x_1 + \beta_2 x_2$ if we find that β_2 is not significantly different from zero. There are two fundamental differences between these two principles of parsimony. First, statistical parsimony is data-driven: the choice of the model depends on the data at hand. Phylogenetic parsimony is an a priori concept which does not require any data. Second, statistical parsimony is more closely related to the principle of not invoking unnecessary causes known as Occam's razor. A single cause, high mutation rate, may result in many changes in a molecular sequence. Parsimony in process does not imply parsimony in pattern. On the other hand, parsimony analysis of complex characters (eye, lung, ...) is meaningful because such characters evolve slowly and, certainly, each with different causes, but this logic applies poorly to DNA sequences.

that the likelihood approach does not ignore the fact that z may not have A (z is not observed) but this requires to assume an evolutionary model.

Let us consider the second site now (recall that it is A in x and T in y). The likelihood calculations followed logically, but what about applying parsimony in this case? Clearly, we have to assume that one change occurred. But on which branch? Because branch lengths do not exist in parsimony, this is a problem.

Suppose that we observe the same second site on many other species (x and y are still sister-species). If A is observed in these species, then we will conclude that a change occurred on the branch $z \rightarrow y$. If T is observed instead, then we will conclude that a change occurred on the branch $z \rightarrow x$. The likelihood calculations for z will remain the same in both cases.

This illustrates a property of parsimony. When polymorphism is low, parsimony calculations are relatively easy, but when this is not the case, they are problematic.

phangorn has the function `parsimony` that computes the parsimony score of a tree (or a list of trees) and some data of class `"phyDat"`. Two methods are available through the option `method`: `"sankoff"` (the default) and `"fitch"`.

```
> parsimony(tw, x)
[1] 68
```

`optim.parsimony` searches for the maximum parsimony tree by performing NNIs on the topology; the branch lengths are in number of inferred changes:

```
> pars.tr <- optim.parsimony(tr, x)
optimize topology:  68 --> 68
Final p-score 68 after  0 nni operations
> pars.tr$edge.length
 [1] 1 1 1 0 1 3 1 6 9 5 3 2 2 3 3 0 4 5 1 2 5 2 1 1 1 2 3
```

Thus the MP tree has the same topology than the NJ tree. We remember that the ML tree inferred on page 153 has a slightly different topology than the NJ one:

```
> parsimony(o5$tree, x)
[1] 68
```

So the ML tree has the same parsimony score.

The functions `CI` and `RI` calculate the consistency and retention indices:

```
> CI(tw, x)
[1] 3.191176
> RI(tw, x)
[1] -0.4056604
```

5.4.2 Hadamard Conjugation

Methods of phylogenetic inference based on the Hadamard conjugation have the advantage of permitting direct calculations of the quantities of interest, instead of the recursive computations needed for likelihood or distance-based methods. Their weaknesses however prevent their general use. Only simple substitution models can be handled: the most complex model that can be analyzed is K81. Furthermore, though the calculations are direct they cannot handle a large number of observations (see below). This section will outline the principle of these methods and illustrate the functions available in phangorn.

The principle of Hadamard methods is to calculate the frequency distribution of the site patterns among the observations. To illustrate this, we take four sequences from the woodmouse data that we transform in two-state characters (purine and pyrimidine, R and Y) with the function `acgt2ry`:

```
> X <- acgt2ry(as.phyDat(woodmouse[12:15, ]))
> X
4 sequences with 915 character and 5 different site patterns
The states are r y
```

To see what are the five site patterns, we print the structure of these data:

```
> str(X)
List of 4
 $ No1114S: int [1:5] 1 2 1 2 2
 $ No1202S: int [1:5] 1 2 1 1 2
 $ No1206S: int [1:5] 1 2 1 1 1
 $ No1208S: int [1:5] 1 2 2 1 2
....
 - attr(*, "weight")= int [1:5] 393 518 1 2 1
....
```

The first observed pattern is R in all four sequences and it was observed 393 times; the second pattern is Y everywhere and it was observed 518 times, and so on.

Hendy and Penny [133] developed a series of equations that allow calculation of the distribution of the site patterns for a given tree assuming a model of equal rate of change R $\leftrightarrow$ Y. By reversing these equations and using the observed site pattern frequencies, it is possible to estimate the branch lengths of the tree. Different trees can be compared with a least squares criterion.

The function `h2st` in phangorn computes the Hadamard conjugation under this model. It is limited to analyze no more than 23 sequences. The calculations are based on identifying the site patterns that define a split: the sequences with R all on one side, and those with Y on the other. With two states, there are 2^{n-1} possible splits, here $n = 4$, so $2^{4-1} = 8$ (taken from the internal code of `h2st`):

```
> sv
[1] 911   2   0   0   1   0   0   1
```

These are the numbers of sites defining each split. So there are 911 sites definig the first split (393 + 518, as seen in `"weight"`), and so on. For each of these splits, one internal branch is defined, and its length is computed with the fast Walsh–Hadamard transform:

```
> cbind(sv, qv)
      sv            qv
[1,] 911  6.814539e+00
[2,]   2  2.195398e-03
[3,]   0  5.290676e-09
[4,]   0 -1.204954e-06
[5,]   1  1.097702e-03
[6,]   0 -2.409891e-06
[7,]   0 -2.409891e-06
[8,]   1  1.097702e-03
```

Logically, the splits defined by site patterns not observed in the data have very small or negative internal branch lenths. `h2st` drops the first split (with all observations) and those whose branch length is below a value given by its option `eps` (0.001 by default):

```
> h2st(X)
[[1]]
[1] 1

[[2]]
[1] 3

[[3]]
[1] 1 2 3

attr(,"weights")
[1] 0.002195398 0.001097702 0.001097702
attr(,"labels")
[1] "No1114S" "No1202S" "No1206S" "No1208S"
attr(,"class")
[1] "splits"
```

`h4st` implements the Hadamard conjugation with the K81 model. It is limited to $n \leq 11$. The calculations are more complicated and we take only three sequences to shorten the output:

```
> h4st(as.phyDat(woodmouse[13:15, ]))
$q
```

```
     transversion  transition.1  transition.2
[1,]  0.003125601 -1.077571e-06  3.019600e-08
[2,]  0.005269232  1.060610e-03 -5.553509e-06
[3,]  0.010586204  1.038186e-03  2.110004e-03

$qv
            [,1]          [,2]          [,3]          [,4]
[1,] 6.848987958  3.019600e-08 -5.553509e-06  2.110004e-03
[2,] 0.003125601 -1.077571e-06 -1.120762e-05 -1.206513e-05
[3,] 0.005269232 -2.226369e-06  1.060610e-03 -1.436271e-05
[4,] 0.010586204  2.527083e-08 -3.255929e-06  1.038186e-03

$sv
     [,1] [,2] [,3] [,4]
[1,]  943    0    0    2
[2,]    3    0    0    0
[3,]    5    0    1    0
[4,]   10    0    0    1

$n
[1] 965

$names
[1] "No1202S" "No1206S" "No1208S"
```

5.4.3 Species Trees *Versus* Gene Trees

Molecular phylogenies allow us to trace back the history of genetic lineages, but these may not coincide completely with the history of species because, as geneticists have demonstrated some time ago, different genes may have different histories [163, 165]. The problem is particularly acute for recently diverged species: Kimura and Ohta's classical result showed that it takes some time for two populations to diverge by genetic drift, so that some shared molecular polymorphism can be the result of recent inherante from their ancestors [164]. This process is called *incomplete lineage sorting* [193].

A classical text-book image shows connected tubes (the species tree) with lines inside (the gene trees) that diverge before or after the tubes diverged themselves. The distribution of the gene trees depends on the coalescent and has attracted a lot of attention in recent years [e.g., 52, 243]. Several methods developed to estimate a species tree from a set of gene trees share a common framework:

1. For each gene tree, compute the cophenetic distance matrix; this gives for each pair of taxa a series of distances derived from each gene.
2. Summarize this series of distances resulting in a new distance matrix for all taxa.

3. Compute a hierarchical clustering from this matrix using a single-linkage method.

This framework is implemented in the function `speciesTree`:

```
speciesTree(x, FUN = min)
```

`x` is a list of (gene) trees, and `FUN` is the function used in step 2 above. The default calculates the maximum tree of Liu et al. [185]. If `FUN = sum`, then the species tree is calculated with the shallowest divergence tree of Maddison and Knowles [193]. The trees in `x` must be ultrametric and with branch lengths on the same temporal scale. Edwards et al. [65] showed using simulations that the gene-tree approach performs much better than the concatenation of sequences (supermatrix) approach, particularly even if most gene trees are discordant with the species tree. However, the gene-tree approach seems well suited when there are many genes and few species.

We take a simple hypothetical example with three species and two gene trees that have the same topology (though the gene trees may have different topologies):

```
> t1 <- read.tree(text = "(c:2,(a:1,b:1):1);")
> t2 <- read.tree(text = "(c:4,(a:2,b:2):2);")
> tmax <- speciesTree(list(t1, t2))
> all.equal(tmax, t1)
[1] TRUE
```

So here the maximum tree is the shortest of the two gene trees. This seems logical in the sense that oldest gene splits predate the species divergence. The shallowest divergence tree is different (Fig. 5.8):

```
> tsha <- speciesTree(list(t1, t2), sum)
> kronoviz(list(t1, t2, tmax, tsha), type = "c")
```

5.5 Bootstrap Methods and Distances Between Trees

The use of the bootstrap has enjoyed great success in phylogenetic analyses [75]. 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 the expected distribution of this parameter does not hold (because, e.g., of a small sample size, or non-independence of observations). Then we could resample the data at hand many times, mimicking the process of sampling the real population. The variation in the estimated parameter from the "bootstrap" samples is a measure of the confidence level in this estimate [66].

The idea is simple, intuitive, and elegant, but, in some situations, requires intensive computations [69]. The application of the bootstrap in phylogeny

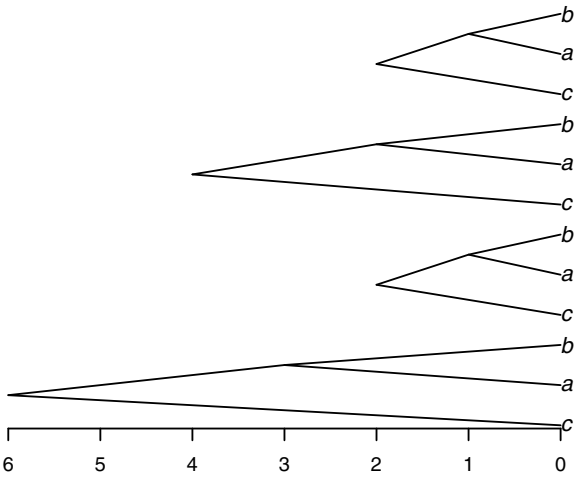

Fig. 5.8. Two gene trees (`t1` and `t2` in the text) on top, the maximum tree, and the shallowest divergence tree on bottom

estimation is almost as simple: estimate a tree with a given method, resample the original data (the matrix taxa × 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 [68] showed that this was due to confusion in the interpretation of the original bootstrap method by Felsenstein [75]. Efron et al. also proposed another way to compute the bootstrap values for hypothesis testing rather than assessing confidence levels [68]. Shimodaira and Hasegawa [283] developed a similar test to compare different topologies with a bootstrap resampling and a likelihood ratio test (see the function `SH.test` in `phangorn`).

In this section, we examine the different ways of resampling phylogenetic data, comparing (possibly a large number of) phylogenetic trees, and computing bootstrap values.

5.5.1 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 × 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. 17). 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 [68, 168]. 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. A general solution is to sample the indices of the vector instead of the vector itself. Let us consider the case of sampling blocks of three nucleotides in a vector `x`. First, build a vector with the indices 3, 6, ...:

```
> 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"
> block <- 3
> i <- seq(block, length(x), block)
> i
[1]  3  6  9 12
```

Then, sample this vector `i` as before:

```
> iboot <- sample(i, replace = TRUE)
> iboot
[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`. Binding these three vectors in a matrix converted then in a vector is a simple way to proceed:

```
> boot.ind <- as.vector(rbind(iboot - 2, iboot - 1, iboot))
> 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.

For data of class `"phyDat"`, the above scheme cannot be used because the characters are compressed: those with the same pattern among taxa are not repeated. phangorn has two functions to perform a bootstrap on this data class:

```
bootstrap.pml(x, bs = 100, trees = TRUE, ...)
bootstrap.phyDat(x, FUN, bs = 100, ...)
```

The first function does the bootstrap with maximum likelihood estimation, while the second has the argument `FUN` similar to `boot.phylo` (see below). In both cases, a list of `bs` trees is returned.

When doing a bootstrap, it is not always trivial what needs to be resampled: the rows or the columns of a data matrix? What we resample depends on the assumptions of the model. In a phylogeny estimation, the rows of the data matrix (the sequences, the species, or the populations) are not assumed to be independent because of their phylogenetic relationships. The characters (the columns of the matrix) are assumed to evolve independently of each other, so these are the elements to be resampled. In case we do not want to assume independent evolution of the characters, resampling of independent blocks of characters may be done.

5.2.2 Bipartitions and Computing Bootstrap Values

Once bootstrap samples and trees have been obtained, it is necessary to summarize the information from them. ape and phangorn provide several functions for this task depending on the approach taken.

A bipartition is similar to a split: it 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 function, the results are indexed according to the node numbers.

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, trees = FALSE,
           quiet = FALSE, rooted = FALSE)
```

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) bionj(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);
`trees`: a logical value indicating whether to return the `B` bootstrap trees;
`quiet`: a logical value indicating whether to display a progress bar;
`rooted`: a logical value indicating whether to treat the trees as rooted or not. If `FALSE`, the splits are counted; if `TRUE`, the clades are counted.

`boot.phylo` returns exactly the same vector as `prop.clades`. By default, the bootstrap trees 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. [68]. Setting `trees = TRUE` allows to return the bootstrap trees: `boot.phylo` then returns a list with the bootstrap values and the trees.

Why would we want to examine bootstrap trees when we have the bootstrap values right at the nodes of the estimated tree? If these bootstrap values are high, then this is fine, but if they are low this may indicate either a lack of information in the data or a systematic bias in the estimation. A lack of information will be revealed by the fact that bipartition frequencies are more or less random among the bootstrap trees. On the other hand, if a bipartition appears at a high frequency, which would not be visible in the bootstrap values because it is absent in the estimated tree, this may be evidence for a bias, maybe due to a few sites that are resampled rarely during the bootstrap. In this case, using a different approach, for instance with partitions or mixtures, may be useful.

A way to analyze a list of trees and the potential conflict among them is provided by the Lento plot [182]. The function `lento` in `phangorn` analyzes a list of trees provided as an object of class `"splits"`. Two quantities are calculated for each split:

- 'support' which is equal to the observed frequency of the split (as given by the attribute `"weights"` in the class `"splits"`, or the attribute `"number"` in the class `"prop.part"`);
- 'conflict' which is the sum of the support values of the splits that are not compatible with the considered one.

Two splits are not compatible if they cannot be observed in the same binary tree (e.g., AB|CD and A|BCD are compatible, while AB|CD and AC|BD are not). For illustration, we do a bootstrap analysis on the woodmouse data and store the bootstraped trees:

```
> f <- function(x) nj(dist.dna(x))
> tw <- f(woodmouse)
> o <- boot.phylo(tw, woodmouse, f, trees = TRUE)
  |================================================| 100%
```

```
> o
$BP
 [1] 100  24  54  58  61  50  78  65  87  89  95  99  59

$trees
100 phylogenetic trees
```

We extract the splits and their frequencies; we can do this operation with `prop.part` but we prefer `as.splits` so the result can be input directly into `lento`:

```
> s <- as.splits(o$trees)
> length(s)
[1] 90
```

So there are 108 different splits observed among the 100 bootstrap trees. We can now do the Lento plot (Fig. 5.9):

```
> lento(s, trivial = TRUE)
```

The splits are ordered from left to right with decreasing support value indicated as positive. The conflict values are indicated as negative. The structure of each split is displayed with plot symbols. The first sixteen splits are the trivial ones: the fifteen defined by the terminal branches and the one with all tips. They cannot conflict with any other, and so are obviously observed in all trees—they are not plotted by default. It is interesting to note in the present example that, apart the trivial splits, only one has a non-zero conflict value (No1208S, No1007S, No0909S), which may result from the low divergence of these sequences.

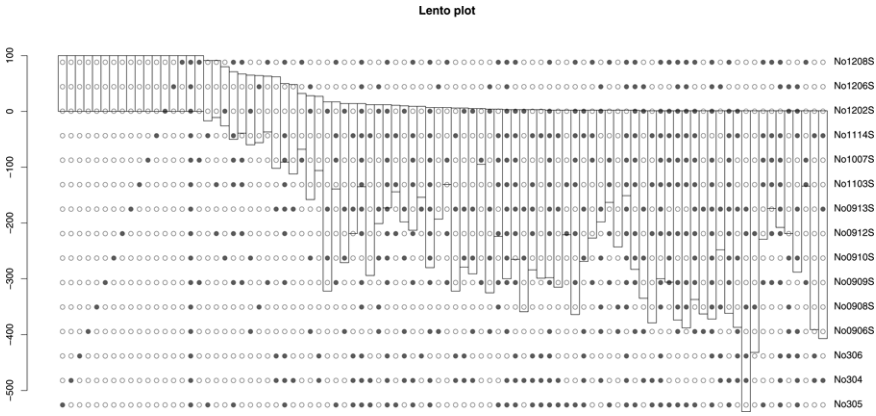

Fig. 5.9. Lento plot from 100 bootstrap trees of the woodmouse data

White et al. [312] proposed an alternative way to partition the phylogenetic signal present in a sequence alignment into three components—internal, terminal, and residual—summing to one and so can be represented as frequencies on a triangle. This has the advantage of displaying the phylogenetic information with a single point, thus several data sets (or subsets) can be represented simultaneously.

Finally, the bootstrap seems ideally suited to examine bias in the estimation of other evolutionary parameters (substitution rates, shape parameter of inter-site variation, branch lengths, ...) though this does not seem to have attracted some attention (see Freedman [91] for the use of the bootstrap to assess estimator bias).

5.5.3 Distances Between Trees

The idea of distances between trees is related to the bootstrap because this requires summarizing and quantifying the variation in topology among different trees. Several ways to compute these distances have been proposed in the literature [79, Chap. 30]. Two of them are available in the function `dist.topo`.

Penny and Hendy [241] proposed measuring the distance between two trees as twice the number of internal branches that differ in their splits. Rzhestky and Nei [272] 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
```

Kuhner and Felsenstein [170] proposed to calculate the square root of the sum of the squared differences of the internal branch lengths defining similar splits in both trees. They called this distance as the branch length score; it is somewhat similar to the previous distance but taking branch lengths into account.

```
> tr <- rtree(10)
> tb <- rtree(10)
> dist.topo(tr, tb, "score")
[1] 2.270937
```

The function `treedist` in phangorn offers an alternative implementation by computing four distances for a pair of trees:

```
> treedist(tr, tb)
     symmetric.difference   branch.score.difference
                14.000000                  2.587576
```

```
path.difference quadratic.path.difference
      16.248077                 10.082005
```

The "symmetric difference" is the same than Penny and Hendy's distance in `dist.topo`. The "path difference" is the difference in path lengths, counted as the numbers of branches, between the pairs of tips [290], while the "quadratic path difference" is the same but using branch lengths.

Billera, Holmes, and Vogtmann [23] developed an 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 (Fig. 5.10). Thus it is possible to define a distance between two rooted trees across these interconnected cubes which is called a geodesic distance. The package distory implements this distance as well as other tools. The function `dist.multiPhylo` calculates the geodesic given a list of trees, which could be an object of class `"multiPhylo"`, and returns an object of class `"dist"`. For instance, computing the distance between the two binary trees on Fig. 5.10:

```
> library(distory)
> ta <- read.tree(text = "((a:1,b:1):1,c:1);")
> tc <- read.tree(text = "((a:1,c:1):1,b:1);")
> dist.multiPhylo(c(ta, tc))
  1
2 2
```

For rooted trees `phylo.diff` is a graphical tool to identify internal branches that are unique between two trees (see also `compare.phylo`, p. 54).

The fact that the geodesic distance has a physical interpretation leads to the possibility of using it in some applications related to the idea of a tree space. distory implements some of them. For instance, the help page `?dist.multiPhylo` gives an example of the use of the geodesic distance together with multidimensional scaling (also known as principal coordinates analysis) to represent geometrically a set of bootstrap trees.

Another application is provided by the function `mcmc.target.seq` which finds, using MCMC, a bootstrap sample of a `"DNAbin"` object that is closest to a target tree in terms of geodesic distance [68]. The distance at each step is output so that the convergence of Markov chain can be assessed.

5.5.4 Consensus Trees and Networks

Consensus trees are an interesting way to summarize a set of trees: if they are dichotomous, the clades not observed in all (strict consensus) or the majority (majority-rule consensus) will be collapsed as multichotomies.

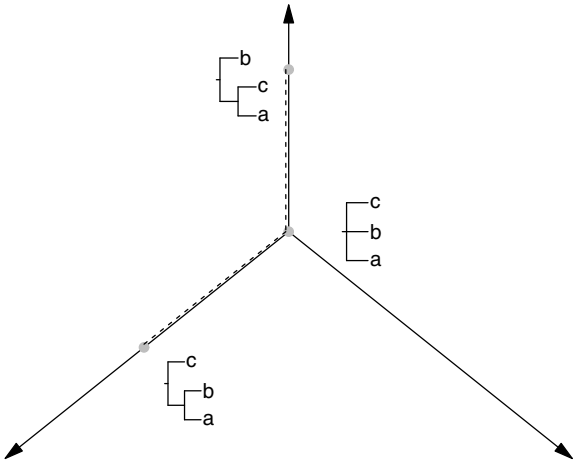

Fig. 5.10. The tree space for $n = 3$. It is made of three 1-d spaces interconnected at their origin—and not the three dimensions of a 3-d space. The grey dots indicate the positions of the three trees in this space. The dashed line shows the geodesic distance between the two binary trees. In addition each tree is characterized by an n-d Euclidean space representing the terminal branches

The function `consensus` in `ape` 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 number between 0.5 and 1, of the proportion of the splits 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 [79].

This parameter l cannot be smaller than 0.5 because the splits not observed in at least 50% of the trees are incompatible with some observed in the majority of the trees and so cannot be represented together in the same tree. A way to solve this problem is to display alternative branchings with reticulations. Consider the unrooted trees in the middle of Fig. 5.11: their internal branches define splits, namely 12|34, 13|24, and 14|23, which clearly cannot be observed in the same tree and thus are all incompatible. The network on the top left represents simultaneously the two first trees with a rectangle symbolizing the two internal branches. In order to represent all three splits, we need a cube as shown by the network on the right of Fig. 5.11.

The remarkable thing about consensus networks is that once boxes and rectangles have been identified like in Fig. 5.11, they can be assembled using the geometry of some structures known as squaregraphs [14]. Figure 5.12 shows an example of a network made of two rectangles displaying three splits. An internal branch in an unrooted tree is represented as two or more parallel

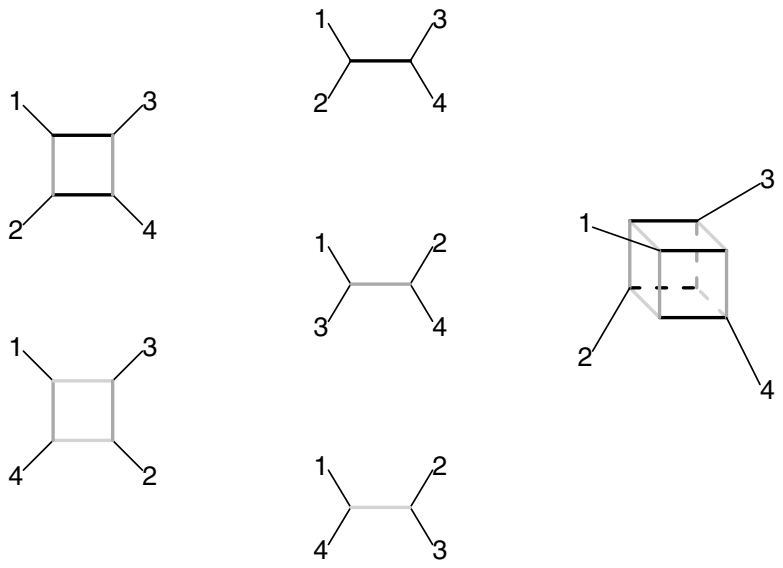

Fig. 5.11. The three possible unrooted topologies with $n = 4$ (middle), two out of the three possible consensus networks involving two topologies (left), and the consensus network representing the three topologies (right). The internal branches and their representations in the networks are shown with the same shades of grey

edges in a consensus network. This idea was originally introduced by Bandelt [15, 16, 17].

The function `consensusNet` in `phangorn` computes the consensus network for a list of trees given as first argument or, equivalently, a list of splits. The second argument, `prob`, gives the minimum frequency for a split to be uncluded in the network (0.33 by default). We come back to the bootstrap trees from the woodmouse data:

```
> cnt <- consensusNet(s) # same than consensusNet(o$trees)
> class(cnt)
[1] "networx" "phylo"
> cnt

Phylogenetic tree with 15 tips and 17 internal nodes.

Tip labels:
No305, No304, No306, No0906S, No0908S, No0909S, ...

Unrooted; includes branch lengths.
```

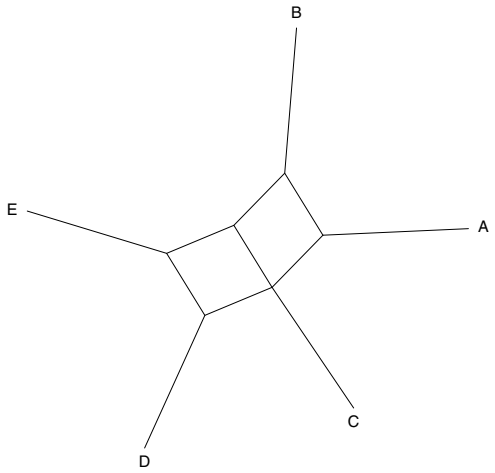

Fig. 5.12. A consensus network showing the splits AB|CDE, ABC|DE, and ACD|BE which is incompatible with the two previous ones

We note that the number of nodes is greater than the number of tips—in a tree without reticulation the former is always smaller than the latter. The network is then plotted with the appropriate method (Fig. 5.13):

```
plot(cnt, "2", tip.color = "black", edge.width = 1, font = 1)
```

The edge lengths are proportional to the frequencies of the splits, so a more "open" box indicates, in the present case, a stronger bootstrap support. This simple example clearly shows the advantage of the consensus network over a tree with bootstrap values. Here, the alternative positions of No1206S and No0908S cannot be represented together on the same tree, and similarly for No1007S, No1208S, and No0909S. Setting `prob = 0.1` gives logically a more complex network which is best visualized in 3-D with rgl (see Exercices).

The 3-D plot of consensus networks was originally developed by Schliep for phangorn [278]; the 2-D version used here is derived from this one (Klaus Schliep, personal communication). Other details on plotting networks can be found on page 117.

5.6 Molecular Dating

Branch lengths and substitution rates are confounded in phylogenetic estimation, 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. This is the approach used in molecular dating methods. These techniques have two goals: transform a non-ultrametric tree into an

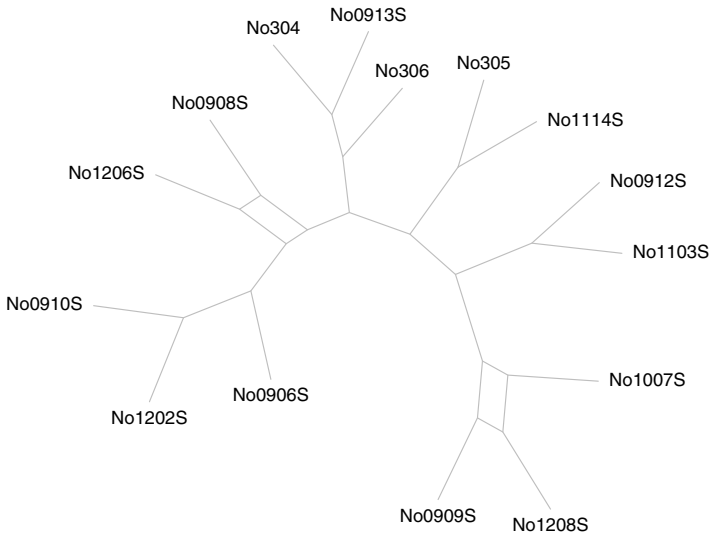

Fig. 5.13. Consensus network made from 100 bootstrap trees of the woodmouse data

ultrametric one, and estimate the absolute dates of the nodes of a tree. The latter requires at least one "calibration point", that is a node of the phylogeny which date is known. Each of these dates may be known exactly or within an interval defined by a lower (younger) and upper (older) bounds.

5.6.1 Molecular Clock

If substitution rate is constant, then molecular sequences are said to evolve according to the molecular clock: molecular divergence is related to time. Note that even if the substitution rate is constant, we do not expect a strictly linear relationship between molecular distance and time because of the stochastic nature of molecular evolution [163].

There are various methods in the literature to estimate a dated tree under a molecular clock model [79]. Britton et al. [32] proposed the mean path lenghts method where the date of a node is estimated with the mean of the distances from this node to all tips descending from it. This is implemented in the function `chronoMPL` in `ape` which takes as argument a phylogeny whose branch lengths are the mean numbers of substitutions, so branch lengths in numbers of susbtitutions per site must be multiplied by the sequence length before analysis. Under the molecular clock assumption, the two subtrees originating from a node are expected to accumulate similar molecular divergence: this leads to a test, computed by default by `chronoMPL`, for each node of the phylogeny. The results are returned as attributes of the dated tree (and thus extracted with `attr`; see `?chronoMPL` for an example).

The present method must be seen as mainly exploratory. Often, a plot of the original rooted phylogeny is very informative on variation in substitution rate.

5.6.2 Penalized Likelihood

Sanderson [275] proposed a method called nonparametric rate smoothing (NPRS) which assumes that each branch of the tree has its own rate, but these rates change smoothly between connected branches. 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:

$$\Phi = \sum |\hat{r}_k - \hat{r}_j|^p \ , \tag{5.12}$$

where $\hat{r}$ is the estimated substitution rate, k and j are two nodes of the same branch, and p is an exponent (usually 2). In order to make a trade-off between clock-like and non-clock models, Sanderson [276] proposed to modify his method by using a semiparametric approach based on a penalized likelihood. The latter (denoted Ψ) is made of the likelihood L of the "saturated" model (the one that assumes one rate for each branch of the tree) minus a roughness penalty which is similar to (5.12) multiplied by a smoothing parameter λ:

$$\Psi = \ln L - \lambda\Phi \ , \tag{5.13}$$

$$L = \prod r^x \frac{\mathrm{e}^{-r}}{x!} \ . \tag{5.14}$$

with x being the number of substitutions observed on a branch. The product in 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 clock-like model with the same rate for all branches. In order to choose an optimal value for λ, Sanderson [276] suggested a cross-validation technique where each terminal branch is removed from the data and then its length is predicted from the reduced data set. 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 [276].

The penalized likelihood method is implemented in the function `chronopl`; its interface is:

```
chronopl(phy, lambda, age.min = 1, age.max = NULL,
         node = "root", S = 1, tol = 1e-8, CV = FALSE,
         eval.max = 500, iter.max = 500, ...)
```

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, `age.min` and `age.max` are numeric vectors giving the lower and upper bounds of the dates that are known, `node` is the number of the nodes whose dates are known (the last three arguments must be of the same length), `S` is the number of sites used as unit for the original branch lengths (usually this must be left as default), `CV` is a logical specifying whether to do the cross-validation, and `tol`, `eval.max` and `iter.max` are usual parameters to control the estimation process. 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, for instance, for $\lambda = 0.1, 1, 10, \ldots, 10^6$:

```
l <- 10^(-1:6)
cv <- sapply(l, function(x) sum(attr(chronopl(phy,
                    lambda = x, CV = TRUE), "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, CV = TRUE)
plot(attr(chr, "D2"), type = "l")
```

5.6.3 Bayesian Dating Methods

A Bayesian approach to molecular dating requires to define priors for the substitution rates and for the dates of the tree. There are several ways to do the latter, the most common one being to assume that the times between branching events follow a specific distribution. Thorne et al. [300] proposed to use a Dirichlet distribution (a Dirichlet process generates multidimensional variables that sum to one). Other methods are based on a birth-death process [317]. Defining a prior for the substitution rate is somewhat easier and a lognormal distribution is usually assumed [see 300, for details].

As mentioned above, implementing Bayesian methods requires a lot of programming and it makes sense to interface existing programs with R. That is what the package LAGOPUS does with its function `multidivtime` which calls Thorne's programs 'estbranches' and 'multidivtime' which themselves

call Yang's program 'baseml'. These programs must therefore be installed on the computer and accessible from R's working directory. The most efficient way to achieve this is to create symbolic links from the executable binaries to the working directory. Running multidivtime is quite tedious and LAGOPUS makes a very good job to have the results returned into R.

The input data is an object of class `"mdt.in"` built with the function of the same name: it includes all the sets of sequences, associated trees, and a data frame defining the time calibration points. The list `multicntrl.dat` controls the MCMC run by multidivtime and must be loaded into R:

```
> library(LAGOPUS)
> data(multicntrl.dat)
> multicntrl.dat
$numsamps
[1] 10000

$sampfreq
[1] 100

$burnin
[1] 1e+05

$rttm
[1] 90

$rttmsd
[1] 90

$rtrate
[1] "median"

$rtratesd
[1] "median"

$brownmean
[1] 0.4

$brownsd
[1] 0.4

$minab
[1] 1

$newk
[1] 1
```

```
$othk
[1] 0.5

$thek
[1] 0.5

$bigtime
[1] 200

$tipsnotcoll
[1] 0

$nodata
[1] 0

$commonbrown
[1] 0
```

Once the data have been prepared with `mdt.in` and the above list has been loaded, `multidivtime` may be called in R; the inteface is:

```
multidivtime(x, file = NULL, start = "baseml", part = 1,
             runs = 1, path = NULL, transfer.files = TRUE,
             LogLCheck = 100, plot = TRUE)
```

The object returned is of class `"mdt"` and is a list with three components: `TREE` (the estimated chronogram), `TABLE` (the estimated dates with confidence intervals), and `CONSTRAINTS` (a recap of the time constraints used as calibration points). There is also a nice `plot` method which draws the credibility intervals computed from the posterior density of the dates over the estimated chronogram; this has many options.

Running `multidivtime` from LAGOPUS takes a few minutes with the default controls. So this package not only eases the use of an existing program, but also makes possible to repeat the analyses since Bayesian computations are relatively fast in the present case.

5.7 Summary and Recommendations

Figure 5.14 summarizes a workflow of phylogeny estimation derived from this chapter. This shows how the techniques exposed in the previous sections can be used for phylogenetic inference. This considers DNA sequences as the raw data: they are not the only kind of data that can be analyzed for estimating phylogenies, but they have appeared more and more clearly over the last few years as the most appropriate for this task [31, 64]. Table 5.5 gives for each

step, the points which are particularly important to examine, though they are not limitative.

There has been an increasing trend in the literature over the past few years to perform estimation of phylogenies as a black-box computation where data are input and a tree is output. The availability of high performance hardware and software, not surprisingly, has reinforced this practice. I defend here a different approach. Phylogeny estimation should be done like other statistical inference procedures. Thus, some attention must be paid to heterogeneity and variability in the processes under investigation. This obviously requires a combination of exploratory, graphical, and modeling tools. The description of such tools in this chapter, and these final recommendations, do not mean to be exhaustive in this respect. Nevertheless, I hope they will contribute to a more critical approach to phylogeny estimation.

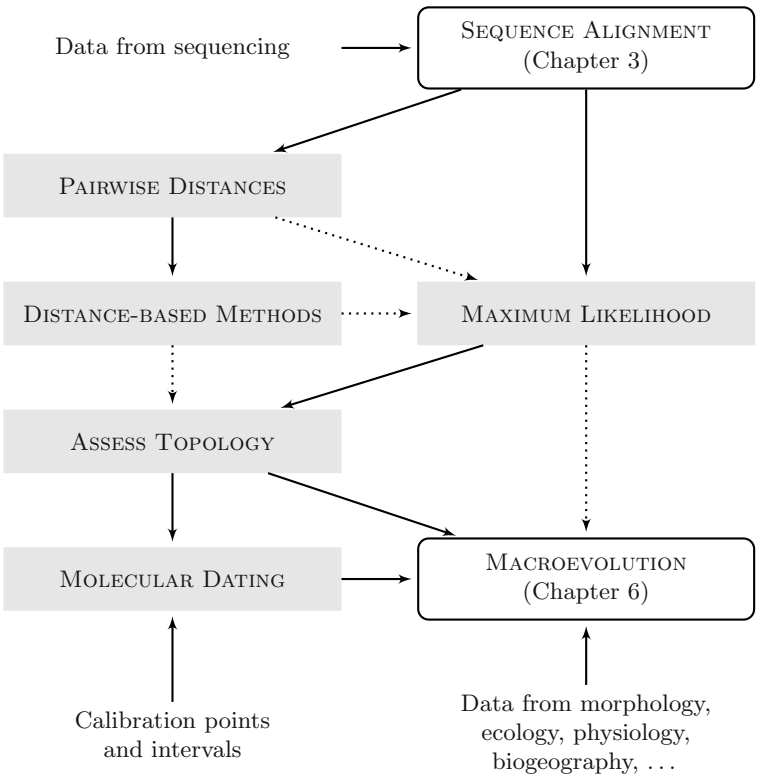

Fig. 5.14. Flowchart of phylogeny estimation. Straight arrows indicate direct flow of data; dotted arrows show when some results from the origin would impact the process in the destination

Table 5.5. Some points to be considered in phylogeny estimation

	Points to be examined	Tools
Pairwise Distances	Low level of variability	Histogram and summary statistics
	Multiple substitutions	Plot JC69 against uncorrected
	Ts/Tv bias (etc.)	Plot K80 against JC69, F81 against F84 (etc.)
	Heterogeneity along the sequences	Same than above with data subsets (e.g., codon positions)
	Heterogeneity among genes	Same than above with different plots on the same figure
	Saturation	Histogram and saturation plots
Dis. Met.	Robustness to the algorithm	Use different methods (NJ, FastME, BIONJ, ...)
	Impact of heterogeneity within / among sequences	Repeat analyses for different data subsets
Max. Likelih.	Test hypotheses on variation in substitution rate(s)	Fit alternative models
	– along sequences	With(out) Γ variation
	– among genes	With(out) partitions
	– parameters	Alternative models (GTR, ...)
	– base frequencies	Id.
	Alternative topologies	Shimodaira–Hasegawa test
Assess Top.	Confidence in the inferred topology	Bootstrap, MCMC
	Bias and conflicting signal	Lento plot, consensus network
		Geodesic distance with multidimensional scaling
	Incomplete lineage sorting	Species tree estimation from gene trees
Mol. Dating	Heterogeneity in substitution rate among branches	Plot rooted trees with scale
	Assess unclock-like of the substitution rate	Find the best value of λ with PL
	Impact of individual sequences	Examine individual contributions to CV with D_i^2 with PL

5.8 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.

5.8.1 *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", p = TRUE)
syl.TN93 <- dist.dna(sylvia.seq.ali, model = "TN93", p = TRUE)
syl.GG95 <- dist.dna(sylvia.seq.ali, model = "GG95", p = 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   1.000    1.000    0.928
syl.F84    1.000   1.000    1.000    0.927
syl.TN93   1.000   1.000    1.000    0.925
syl.GG95   0.928   0.927    0.925    1.000
```

This shows that the GG95 distances differ substantially from the others. 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 JC69 model and the raw distance (i.e., proportion of different sites):

```
syl.JC69 <- dist.dna(sylvia.seq.ali, model = "JC69", p=TRUE)
syl.raw <- dist.dna(sylvia.seq.ali, model = "raw", p=TRUE)
```

We then plot these two distances expecting the raw distances to be smaller because they do not consider multiple substitutions; we also plot the Jukes–Cantor distance *versus* the Kimura one to show the potential influence of the transition/transversion ratio (Fig. 5.15):

```
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)
```

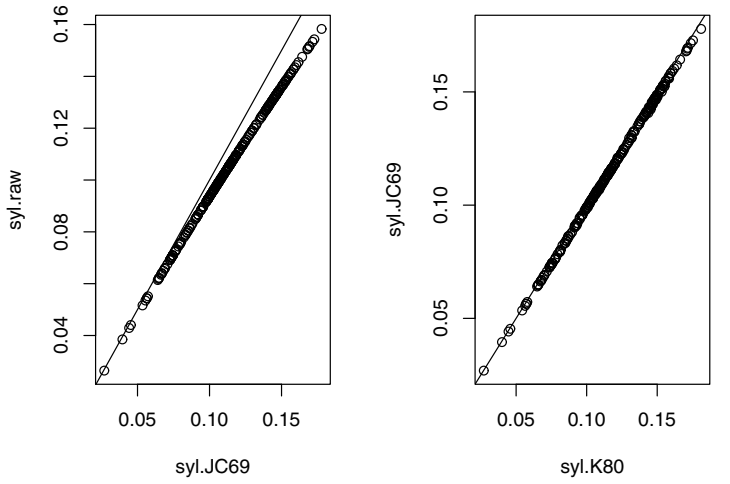

Fig. 5.15. Saturation plots for the cytochrome *b* sequences of 25 species of *Sylvia* showing the effects of multiple substitutions (left) and of the transition/transversion ratio (right)

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.

This analysis, though informative, is not what is usually called "saturation plots" in the literature. The latter is a plot, eventually for each codon position, of the numbers of transitions and transversions on the x-axis against the K80 distance on the y-axis. These graphs are not shown here but the code to do them is as follows:

```
layout(matrix(1:3, 1))
for (i in 1:3) {
    s <- logical(3); s[i] <- TRUE
    x <- sylvia.seq.ali[, s]
    d <- dist.dna(x, p = TRUE)
    ts <- dist.dna(x, "Ts", p = TRUE)
    tv <- dist.dna(x, "Tv", p = TRUE)
    plot(ts, d, xlab = "Number of Ts or Tv", col = "blue",
         ylab = "K80 distance", xlim = range(c(ts, tv)),
         main = paste("Position", i))
    points(tv, d, col = "red")
}
```

A special attention must be paid to the scales of the x- and y-axes when interpreting the three plots created by these commands.

A complementary graphical analysis can be done by plotting a histogram of the pairwise distances for each codon position (as suggested in Table 5.5). In order to plot the three histograms on the same scales, we use the function `histogram` in lattice instead of the usual `hist`. To this end, we prepare the data by computing the distances for each position, and concatenating them in a single vector. We then create a factor variable so that `histogram` can attribute each value of the concatenated vector to its codon position which is used as a conditional variable in the call of this function (Fig. 5.16):

```
y <- numeric()
for (i in 1:3) {
    s <- logical(3); s[i] <- TRUE
    y <- c(y, dist.dna(sylvia.seq.ali[, s], p = TRUE))
}
g <- gl(3, length(y) / 3)
library(lattice)
histogram(~ y | g, breaks = 20)
```

The figure is very revealing: not only the mean pairwise distance is affected by the codon position but also its variance. This will have a significant impact on the choice of model for phylogeny estimation.

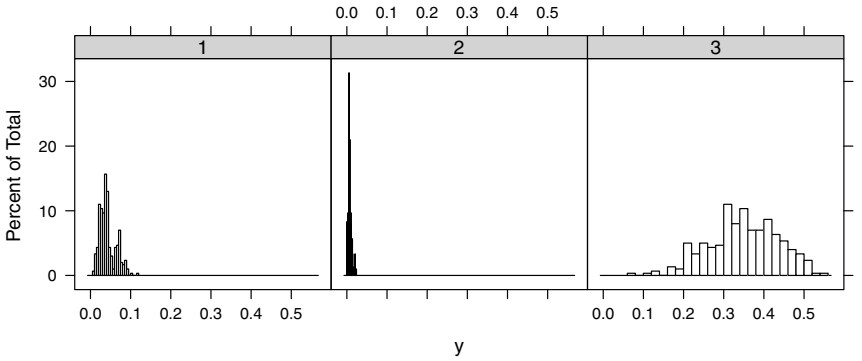

Fig. 5.16. Pairwise histograms for the cytochrome b sequences of 25 species of *Sylvia* for the three codon positions

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 the K80 and GG95 distance matrices:

```
nj.sylvia.K80 <- nj(syl.K80)
```

```
nj.sylvia.GG95 <- nj(syl.GG95)
```

To see if the estimated topologies are similar, we compute the topological distance between them:

```
> dist.topo(nj.sylvia.K80, nj.sylvia.GG95)
[1] 12
```

We now do a bootstrap analysis like the one reported by Böhning-Gaese et al. [27] using `boot.phylo`. There are several ways to conduct this analysis. We choose to treat rooted trees, and so first identify the outgroup species, *Chamaea fasciata*:

```
> grep("Chamaea", taxa.sylvia, value = TRUE)
          AJ534526
"Chamaea_fasciata"
```

We then build a function that includes the `root` function in order to estimate the NJ tree including the rooting operation:

```
f <- function(xx) root(nj(dist.dna(xx, p=TRUE)), "AJ534526")
tr <- f(sylvia.seq.ali)
## same than: tr <- root(nj.sylvia.K80, "AJ534526")
```

We may now call `boot.phylo` specifying the option `rooted = TRUE` and using 200 bootstrap replicates as in [27]:

```
> nj.boot.sylvia <- boot.phylo(tr, sylvia.seq.ali, f, 200,
+                              rooted = TRUE)
> nj.boot.sylvia
 [1] 200 186  78 143 146 103 107 200 187  94 192 197  33 154
[15]  82  86 170 195  86 197 134 193  72
```

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.codon <- boot.phylo(tr, sylvia.seq.ali, f, 200, 3,
+                             rooted = TRUE)
> nj.boot.codon
 [1] 200 193  72 139 134  99  99 198 180  87 192 194  38 163
[15]  75  87 164 194  97 196 140 196  94
```

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 and substitute the accession numbers (which were used as tip labels) with the species names:

```
nj.est <- tr
nj.est$tip.label <- taxa.sylvia[tr$tip.label]
```

The tree is then plotted with `plot`, the bootstrap values are added with `nodelabels`, and we draw a scale bar (Fig. 5.17):

```
plot(nj.est, no.margin = TRUE)
nodelabels(round(nj.boot.sylvia / 200, 2), bg = "white")
add.scale.bar(length = 0.01)
```

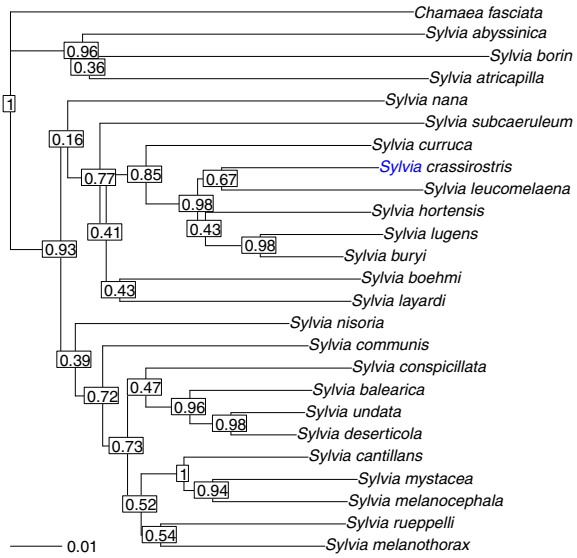

Fig. 5.17. Phylogenetic relationships among 25 species of the genus *Sylvia* based on cytochrome *b* sequences analyzed with neighbor-joining and Kimura's two-parameter distance

The bootstrap values shown in Fig. 5.17 are very close to those obtained by Böhning-Gaese et al. [27]. We finally save the final tree in a file using the Newick format:

```
write.tree(nj.est, "sylvia_nj_k80.tre")
```

We now move on to maximum likelihood by first performing an analysis with `phymltest`. PhyML has been installed and set as described on page 159. The command for the present analysis is thus simply (after writing the sequences in a file):

```
write.dna(sylvia.seq.ali, "sylvia.txt")
phyml.sylvia <- phymltest("sylvia.txt", execname = "~/phyml")
```

This takes a few minutes to run on a modern laptop. Displaying the results shows the log-likelihood and AIC values for each model:

```
> phyml.sylvia
          nb.free.para    loglik      AIC
JC69                 1 -9022.097 18046.19
JC69+I               2 -8307.898 16619.80
JC69+G               2 -8227.001 16458.00
JC69+I+G             3 -8223.574 16453.15
K80                  2 -8508.629 17021.26
K80+I                3 -7766.568 15539.14
K80+G                3 -7680.375 15366.75
K80+I+G              4 -7656.458 15320.92
F81                  4 -8941.140 17890.28
F81+I                5 -8157.509 16325.02
F81+G                5 -8072.674 16155.35
F81+I+G              6 -8055.982 16123.96
F84                  5 -8430.607 16871.21
F84+I                6 -7575.661 15163.32
F84+G                6 -7489.272 14990.54
F84+I+G              7 -7467.842 14949.68
HKY85                5 -8471.269 16952.54
HKY85+I              6 -7600.758 15213.52
HKY85+G              6 -7505.759 15023.52
HKY85+I+G            7 -7482.774 14979.55
TN93                 6 -8413.108 16838.22
TN93+I               7 -7569.273 15152.55
TN93+G               7 -7483.681 14981.36
TN93+I+G             8 -7461.591 14939.18
GTR                  9 -8350.477 16718.95
GTR+I               10 -7548.011 15116.02
GTR+G               10 -7467.290 14954.58
GTR+I+G             11 -7444.848 14911.70
```

The function `summary` computes all possible paired likelihood ratio tests (211 tests):

```
> summary(phyml.sylvia)
       model1    model2       chi2 df  P.val
1        JC69    JC69+I 1428.39644  1 0.0000
2        JC69    JC69+G 1590.19116  1 0.0000
3        JC69  JC69+I+G 1597.04450  2 0.0000
4        JC69       K80 1026.93474  1 0.0000
5        JC69     K80+I 2511.05698  2 0.0000
....
```

We can plot these results to have a more synthetic view (Fig. 5.18):

```
plot(phyml.sylvia, col = "black")
```

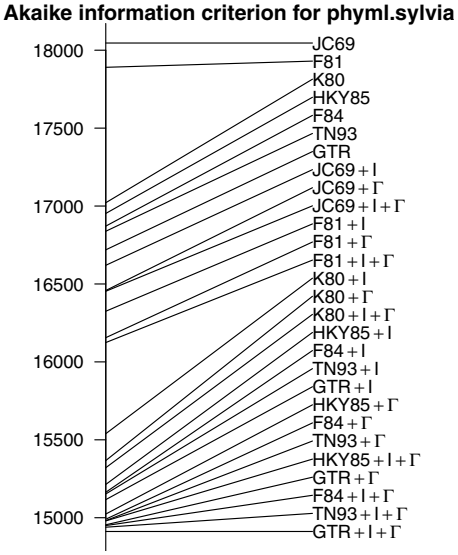

Fig. 5.18. Results of the analysis of 25 species of the genus *Sylvia* based on cytochrome *b* sequences with `phymltest`

The most complex model GTR + I + Γ is the one that best explains the data in terms of AIC. An interesting pattern from Fig. 5.18 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 $X >>> X + I >> X + \Gamma > X + I + \Gamma$ (X being a substitution model).

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("sylvia.txt_phyml_tree.txt")
> TR
28 phylogenetic trees
```

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, and root the tree with *Chamaea fasciata* as outgroup (Fig. 5.19):

```
mltr.sylvia <- TR[[28]]
mltr.sylvia$tip.label <- taxa.sylvia[mltr.sylvia$tip.label]
mltr.sylvia <- root(mltr.sylvia, "Chamaea_fasciata")
plot(mltr.sylvia, no.margin = TRUE)
add.scale.bar(length = 0.01)
```

Naturally, the next step of our analyses should be to fit partitioned models because of the heterogeneity we have characterized in relation to codon position. However, the practical details of this approach have been exposed with the woodmouse data earlier in this chapter (p. 154), and we shall keep the present analysis shorter than it should be.

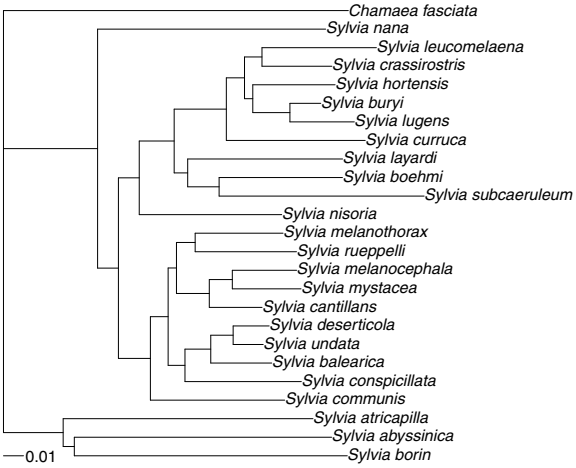

Fig. 5.19. Maximum likelihood phylogeny estimate of 25 species of the genus *Sylvia* based on cytochrome *b* sequences with the GTR + I + Γ model

From this ML estimate of the phylogeny of *Sylvia*, we can now estimate a chronogram with the penalized likelihood method [276]. The fossil record of these birds is extremely sparse and it is difficult to obtain a calibration point from it. Blondel, Catzeflis and Perret [26] found, using a molecular clock, a time of origin of the common ancestor of *Sylvia* between 12.1 and 13 million years ago which, unsurprisingly, agrees with the date estimated by Sibley and Ahlquist [284]. For the present analysis, we use an interval 12-16 Ma for the most recent common ancestor of *Sylvia*. We perform the penalized likelihood analysis, with cross-validation, with values of the smoothing parameter (λ) $10^{-4}, 10^{-3}, \ldots, 10^{3}, 10^{4}$. Before proceeding, we drop the outgroup from the estimated maximum likelihood tree:

```
tr.ml <- drop.tip(mltr.sylvia, "Chamaea_fasciata")
res <- vector("list", 9)
```

```
for (L in -4:4)
    res[[L + 5]] <- chronopl(tr.ml, 10^L, 12, 16, CV = TRUE)
```

We now plot the sum of the cross-validation values (5.15) against the value of λ (Fig. 5.20):

```
Lambda <- 10^(-4:4)
CV <- sapply(res, function(x) sum(attr(x, "D2")))
plot(Lambda, CV / 1e5, log = "x")
```

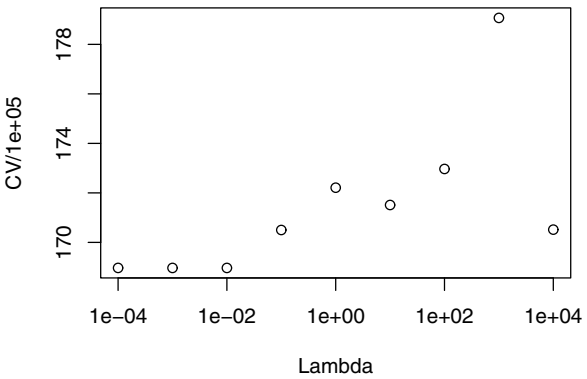

Fig. 5.20. Plot of the cross-validation criterion (CV) with respect to the value of the smoothing parameter λ

The lowest values show the lowest discrepancy between the predicted and the observed values. Therefore, we select the chronogram estimated with $\lambda = 10^{-3}$. We remind that a small value of λ implies heterogeneous substitution rates among brances (because we have set an absolute time frame, branch lengths and substitution rates can be disentangled). The attribute `"rates"` stores the estimated absolute rates of substitution:

```
> sylvia.chrono <- res[[2]]
> rts <- attr(sylvia.chrono, "rates")
> summary(rts)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
0.002187 0.013950 0.025660 0.025350 0.034200 0.059810
```

These values are ordered along the edges of the tree, so they can be used as argument to some options of `plot.phylo`, such as `edge.color` or `edge.width`.

We now plot the estimated chronogram with the edge thickness proportional to these rates, and draw the time-axis with `axisPhylo` (Fig. 5.21). The scaling factor of 100 is found after several attempts in order to get the best visual effect:

```
par(mar = c(2, 0, 0, 0))
plot(sylvia.chrono, edge.width = 100*rts, label.offset = .15)
axisPhylo()
```

We finally save this chronogram for further analysis:

```
write.tree(sylvia.chrono, "sylvia.chrono.tre")
```

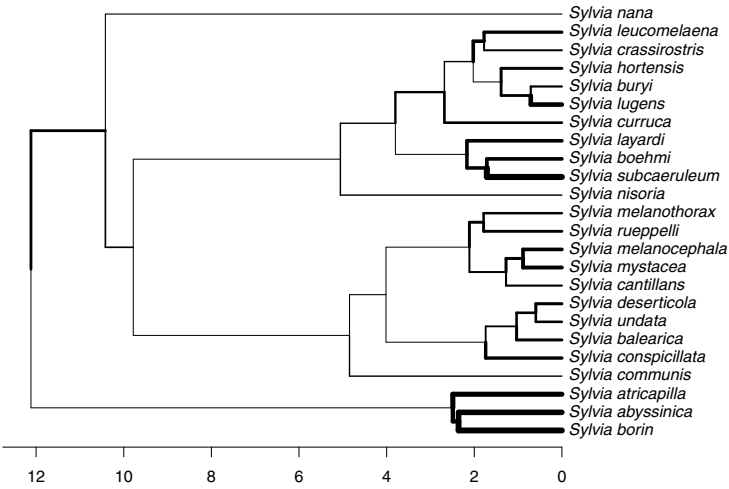

Fig. 5.21. Chronogram of the *Sylvia* warblers based on the penalized likelihood method. The branch thicknesses are proportional to the absolute rates of molecular evolution

5.8.2 Butterfly DNA Barcodes

We have 466 aligned sequences of COI: we limit ourselves here to simple analyses. Hebert et al. [131] 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, p = 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
```

We may plot an histogram of the 108,345 distances (Fig. 5.22):

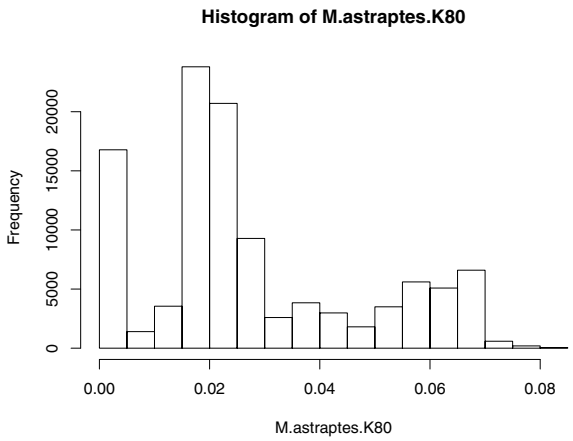

Fig. 5.22. Distribution of pairwise distances among 466 specimens *Astraptes fulgerator* based on cytochrome oxydase I sequences analyzed with Kimura's two-parameter distance

```
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 <- vector(mode = "list", length = 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])`.

5.9 Exercises

1. (a) Show that ultrametric distances are also Euclidean.
   (b) Simulate some data with `rTraitCont` and show that distances among these variables may be Euclidean. Compare with data generated with `rnorm`.
   (c) Show that DNA distances cannot be Euclidean. Find a distance method with continuous variables that shows the same property. See the formula in `?dist` and compare with what you know on how DNA distances are calculated.

2. Compare the seven methods available in `upgma`. You will draw a single figure with the seven UPGMA trees and the necessary annotations. You will take a data set of your choice.

3. Longer distances inferred from molecular sequences tend to have higher variances than the shorter ones. Derive a weighted version of the least squares formula by modifying equation 5.1 (p. 136) in order to give less importance to longer distances. Find a diagnostic plot that will confirm the rationale of this weighting scheme. (Hint: you may type `example(lm)` in R to find some inspiration.)

4. 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?

5. 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?

6. Write R code to calculate the distance between two aligned nucleotide sequences with the GTR model using equation 5.8 (p. 144). (Hints: you will need the functions `base.freq`, `Ftab`, `eigen`, and `solve`. The 'trace' function is the sum of the diagonal elements of a matrix.)

7. (a) Give the R code to calculate `Py` in Section 5.2.2. Compare with `Px`. How this will affect subsequent calculations? Eventually try different values of α.
   (b) Why the likelihood values do not sum to one?
   (c) For site 2, why the likelihood for A is twice bigger than for T?
   (d) How many parameters are involved in these calculations?
   (e) Write an R function to calculate the likelihood of the (full) data; the arguments of this function will be these parameters.

8. Sketch a function doing Bayesian estimation of phylogeny. The code should include comments explaining the rationale of the choices.

9. 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).

10. Generate 100 bootstrap trees from the woodmouse data (see p.176). Compute and plot the consensus network displaying the splits with frequency 0.1 or more. Compare with Fig. 5.13 and explain the differences. Repeat with 1000 bootstrap trees.

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, the third one, methods to analyze diversification above the species level, and the fourth one treats issues recently developed in the use of phylogenies in ecology and biogeography.

6.1 Phylogenetic Comparative Methods

Comparing observations made on different species is an intuitive and appealing approach that certainly dates back to antiquity [127]. 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. Therefore, 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 [76, 267].

E. Paradis, *Analysis of Phylogenetics and Evolution with R*, Use R,
DOI 10.1007/978-1-4614-1743-9_6, © Springer Science+Business Media, LLC 2012

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 [127] for a review). With the growing availability of complete phylogenies with estimated branch lengths, it is possible to go further.

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 [128]. 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 [122].

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).

Phylogenetic comparative methods are now part of the standard background of all evolutionists. As in any statistical method, an improper use of these methods might lead to spurious or aberrant results. We shall see a few examples below; for a more complete treatment of potential problems, see Freckleton [90].

6.1.1 Phylogenetically Independent Contrasts

Felsenstein [76] 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 normal distribution

[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.

with mean zero, and 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:

$$C_{ij} = \frac{x_i - x_j}{\sqrt{d_{ij}}} \,, \tag{6.1}$$

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.

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 (i.e., their covariances are null, 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 [190] 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
        6         7         8         9
3.3583189 1.1929263 1.5847416 0.7459333
> pic.longevity
        6         7         8         9
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")
```

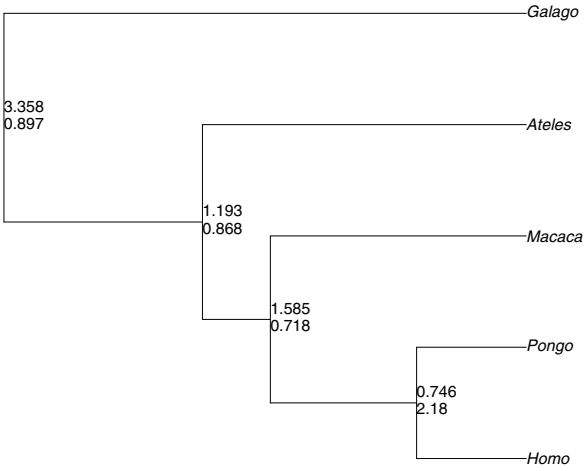

Fig. 6.1. A tree of five primate genera showing phylogenetically independent contrasts of ln(body mass) and ln(longevity), above and below, respectively

A plot of the two sets of PICs shows no clear relationship between them (Fig. 6.2):

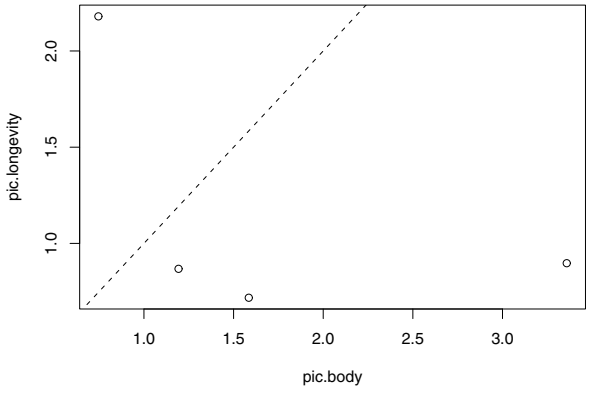

Fig. 6.2. Plot of the four pairs of contrasts from Fig. 6.1; the dashed line is $x = y$

```
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)

Coefficients:
(Intercept)         pic.body
     1.6957          -0.3081
```

Because PICs have expected mean zero, Garland et al. [97] recommended that such linear regressions 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:

```
> 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 [97]. However, I argue that many biologically realistic—and interesting—situations are likely to be ignored by a regression trough the origin, such as nonlinear relationships [252], heterogeneity in rate of evolution, or presence of a trend as suggested with our primate data. In all cases, it is wise to plot the PICs.

Legendre and Desdevises [179] showed that in the case of the regression through the origin the standard test for the significance of the slope based on the F-distribution is actually slightly biased and proposed instead a permutation procedure for this test. This is implemented in the function `lmorigin` which is used in the same way as `lm` with the addition of a few options that control the permutation procedure:

```
> lmorigin(pic.longevity ~ pic.body, nperm = 1e4)
Regression through the origin
Permutation method = raw data
Computation time = 26.087000  sec

Regression through the origin
```

```
Call:
lmorigin(formula = pic.longevity ~ pic.body, nperm = 10000)

Coefficients and parametric test results

          Coefficient Std_error t-value Pr(>|t|)
pic.body      0.43193   0.28649  1.5077   0.2288

Two-tailed tests of regression coefficients

          Coefficient p-param p-perm
pic.body      0.43193  0.2288 0.2350

One-tailed tests of regression coefficients:
test in the direction of the sign of the coefficient

          Coefficient p-param p-perm
pic.body      0.43193  0.1144 0.1155

Residual standard error: 1.137666 on 3 degrees of freedom
Multiple R-square: 0.4310711   Adjusted R-square: 0.2414281

F-statistic: 2.273067 on 1 and 3 DF:
   parametric p-value   : 0.228751
   permutational p-value: 0.2349765
after 10000 permutations of raw data
```

Generally, like with our primate data, the difference will be small, the P-value of the permutation test being slightly larger than the one of the standard F-test.

Purvis and Garland [248] introduced a modification of Felsenstein's [76] method in order to take multichotomies into account. This is not implemented in the function `pic`, but this may be done by combining this function with others such as `multi2di` (Section 3.4.4). There are alternative approaches, such as generalized least squares, to cope with multichotomies with continuous traits (Section 6.1.5).

Contrasts can be calculated for discrete variables. The underlying model would be the threshold model described by Felsenstein [81] where a binary trait is determined by a continuous parameter, called *liability*, and it can be assumed that this parameter evolves under Brownian motion. It follows that the contrasts can be interpreted as changes in liability along the phylogeny. The same approach can be used if there are more than two states with the slight complication that liability will have more than one parameter. This implies that more than one contrast must be computed. This can be done using the binary coding used to enter factors in regression models (the func-

tion `model.matrix` may be useful here). The number of contrasts will be the number of states minus one.

6.1.2 Phylogenetic Autocorrelation

Because species are not independent through their phylogenetic relationships, the latter may be used to quantify the association between the variables observed on the species. Gittleman and Kot [109] introduced a method based on an autocorrelation approach. This uses Moran's autocorrelation index I [208]:

$$I = \frac{n}{S_0} \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij}(x_i - \bar{x})(x_j - \bar{x})}{\sum_{i=1}^{n} (x_i - \bar{x})^2} , \tag{6.2}$$

$$S_0 = \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} , \tag{6.3}$$

where w_{ij} is a weight quantifying the 'phylogenetic proximity' 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 [109] proposed that in the absence of an accurate phylogeny, the weights can be derived from the taxonomy. They considered two ways to calculate them:

- With phylogenetic distances among species, e.g., $w_{ij} = 1/d_{ij}$, where d_{ij} are distances measured on a tree, and $w_{ii} = 0$.
- With taxonomic levels where $w_{ij} = 1$ if species i and j belong to the same group, 0 otherwise.

There are quite a lot of possibilities to set the weights in the first situation. For instance, Gittleman and Kot also proposed:

$$w_{ij} = \begin{cases} 1/d_{ij}^{\alpha} & d_{ij} \leq c \\ 0 & d_{ij} > c, \end{cases}$$

where c is a cut-off phylogenetic distance above which the species are considered to have evolved completely independently, and α is a coefficient (see [109] for details). By analogy to the use of a spatial correlogram where coefficients are calculated assuming different sizes of the "neighborhood" and then plotted to visualize the spatial extent of autocorrelation, they proposed to calculate I at different taxonomic levels.

In the absence of phylogenetic autocorrelation, the mean expected value of I and its variance are known [109]. It is thus possible to test the null hypothesis of the absence of dependence among observations.

Gittleman and Kot's [109] 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` (the user must be sure that the vector and the matrix are ordered in the same way):

```
> w <- 1/cophenetic(tree.primates)
> diag(w) <- 0 # OR: w[w == Inf] <- 0
> Moran.I(body, w)
$observed
[1] -0.0731218

$expected
[1] -0.25

$sd
[1] 0.08910814

$p.value
[1] 0.04714628
```

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`).

The diagonal elements of the matrix returned by `cophenetic` are all equal to zero (the distance of a tip to itself), so calculating $1/0$ returns ∞ and we set the diagonal of `w` to zero. The alternative command is to be preferred if there are tips seperated by zero-length branches so their cophenetic distance is zero as well.

This result suggests a slighly significant positive phylogenetic correlation, in agreement with the positive values observed for the contrasts in the previous section. 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 [208]).

`ade4` has the function `gearymoran` which 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:

```
> library(ade4)
> gearymoran(w, data.frame(body, longevity))
```

```
class: krandtest
Monte-Carlo tests

Test number:   2
Permutation number:   999
       Test         Obs   Std.Obs   Alter Pvalue
1      body -0.06256789 2.1376964 greater  0.001
2 longevity -0.22990437 0.3574691 greater  0.363

other elements: NULL
```

The result for body mass is close to the one with `Moran.I`. This latter function gives with longevity:

```
> Moran.I(longevity, cophenetic(tree.primates))
$observed
[1] -0.1837739

$expected
[1] -0.25

$sd
[1] 0.0911455

$p.value
[1] 0.4674727
```

For this variable, the computed coefficients are close between both functions, but the P-values are somehow different although both not significant. An exercise at the end of this chapter proposes to explore how these results are influenced by the choice of the weights.

adephylo has another implementation with its function `abouheif.moran`: it performs Abouheif's [2] test which is in fact a Moran test with a special weight matrix as shown by Pavoine et al. [240]. This function may be used like the previous one with one or several traits and a weight matrix:

```
> library(adephylo)
> abouheif.moran(cbind(body, longevity), w)
....
Permutation number:   999
       Test         Obs   Std.Obs   Alter Pvalue
1      body  0.03108993 1.8209696 greater  0.012
2 longevity -0.16812515 0.5720004 greater  0.252
```

Another way to proceed is to build a `"phylo4d"` object, and then use the option `method` whose default is `"oriAbouheif"` (other possibilities are `"patristic"`, `"nNodes"`, `"Abouheif"`, or `"sumDD"`):

```
> X <- phylo4d(tree.primates, data.frame(body, longevity))
> abouheif.moran(X)
....
       Test       Obs   Std.Obs   Alter Pvalue
1      body 0.4458995 2.2458001 greater  0.017
2 longevity 0.1848654 0.9640888 greater  0.182
> abouheif.moran(X, method = "sumDD")
....
       Test        Obs  Std.Obs   Alter Pvalue
1      body -0.0449781 2.169082 greater  0.021
2 longevity -0.1112259 1.409788 greater  0.106
```

Gittleman and Kot [109] 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.

To illustrate the use of taxonomic levels, we take the data compiled by Gittleman [108] 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 [109]) 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)
> frm <- SW ~ Order/SuperFamily/Family/Genus
> correl.carn <- correlogram.formula(frm, data = carnivora)
> correl.carn
           obs     p.values      labels
1  0.432479537 8.806040e-05       Genus
2  0.572366226 0.000000e+00      Family
3 -0.119934300 2.135604e-10 SuperFamily
4 -0.004402383 6.776239e-01       Order
```

The returned object is of class `"correlogram"`; there is a `plot` method for this class (Fig. 6.3):

```
plot(correl.carn, col = c("white", "black"))
```

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.

[2] This is the usual notation to specify nested effects in R's formulae; see `vignette("MoranI")` for details.

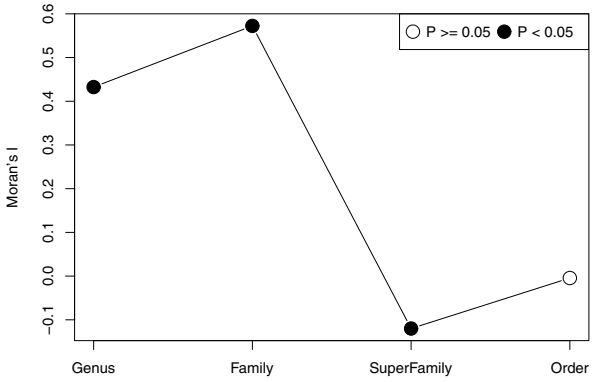

Fig. 6.3. Phylogenetic correlogram of body mass among 112 species of carnivores

Cheverud, Dow, and Leutenegger [42] proposed a method based on autoregression of the trait, but it seems to be inappropriate for comparative analyses [79, p. 442]. This method is implemented in the function `compar.cheverud`, but it is not computationally stable. Furthermore, we will see that generalized least squares provides a nice alternative for the analysis of a single trait (Section 6.1.5).

6.1.3 Orthonormal 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 [57] 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. recommended first running a phylogenetic autocorrelation analysis (Section 6.1.2) 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. Sakamoto, Lloyd, and Benton used this method, together with others described later in this chapter, to assess the quantity of phylogenetic and ecological constraints in the evolution of biting performance among felid species [274].

Ollier, Couteron, and Chessel [222] 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 orthonor-

mal transform on this matrix leading to a matrix whose columns are linear combinations of the columns of the original matrix.

The function `treePart` in package `adephylo` calculates this orthonormal basis. By default, this returns the dummy matrix coding the tree topology. As a simple example:

```
> tr <- rtree(3)
> treePart(tr)
   X5
t3  1
t1  1
t2  0
```

Each column represents a node of the tree, and the rows are the tips. The value is 1 if the tip is a descendant of the node, 0 otherwise (the first column with only 1's is omitted). The option `"orthobasis"` returns the QR decomposition of this dummy matrix:

```
> treePart(tr, "orthobasis")
          V 1           V 2
t3 -0.7071068 -1.224745e+00
t1 -0.7071068  1.224745e+00
t2  1.4142136 -1.155374e-16
```

For a tree with n tips, the returned matrix has n rows and $n-1$ columns. The latter have mean zero, variance one, and are uncorrelated. The successive columns of this orthonormal basis quantifies the phylogenetic dependence among tips in order of decreasing strength [222].

The function `orthobasis.phylo` generalizes Ollier et al.'s idea and computes an orthonormal basis for a phylogeny using different measures of phylogenetic structure:

```
> as.matrix(orthobasis.phylo(tr))
         ME 1        ME 2
t3 -0.7410234  1.20452662
t1 -0.6726390 -1.24400836
t2  1.4136623  0.03948174
```

This matrix is a linear transformation of the cophenetic distance matrix that satifies the three conditions mentioned above (independence, mean zero, and variance one). By default, the cophenetic (patristic) distances are used to calculate this basis. This may be modified either with the option `prox` to pass a user-defined matrix of phylogenetic proximities, or with `method` which may take the same values than in the function `abouheif.moran` seen above except that the default is different.

A phylogenetic analysis of variance can be performed with the first two columns of an orthonormal basis [57]. Considering again the primate data:

```
> B <- as.matrix(orthobasis.phylo(tree.primates))
> X <- B[, 1:2]
> X
             ME 1       ME 2
Homo    1.1696539  0.3172214
Pongo   1.1696539  0.3172214
Macaca -0.3194465 -1.0382542
Ateles -0.7630971 -1.1470267
Galago -1.2567642  1.5508381
```

We may now perform a standard linear regression with `lm`. We are essentially interested in the global F-test of the model which is performed with `anova`:

```
> anova(lm(body ~ X))
Analysis of Variance Table

Response: body
          Df Sum Sq Mean Sq F value   Pr(>F)
X          2 18.910  9.4548  112.57 0.008805
Residuals  2  0.168  0.0840
```

We thus conclude with a significant phylogenetic inertia for body mass. The same analysis with longevity gives:

```
> anova(lm(longevity ~ X))
Analysis of Variance Table

Response: longevity
          Df Sum Sq Mean Sq F value Pr(>F)
X          2 2.2304  1.1152  2.1829 0.3142
Residuals  2 1.0218  0.5109
```

The test is in agreement with the results from the autocorrelation analysis.

Ollier et al. developed a tool called 'orthogram' that quantifies the variance in a continuous trait explained by the phylogeny. They compute a series of coefficients as $r = \frac{1}{n} B^T x_0$ where B is an orthonormal basis and x_0 is the centered and standardized trait x (`scale(x)` in R). The sum of the squared coefficients is equal to one: $\sum_{i=1}^{n-1} r_i^2 = 1$. This leads to two graphical tools: the variance decomposition given by the values r_i^2, and the cumulative variance decomposition given by their cumulative sums. These are computed and plotted by the function `orthogram` in `adephylo`. This function also performs four tests of phylogenetic dependence proposed by Ollier et al. [222] (detailed in `?orthogram`). The significance of these tests is calculated by randomization. The orthogram with body size for the primate shows no significant phylogenetic dependence (Fig. 6.4):

```
> orthogram(body, tree.primates)
```

```
class: krandtest
Monte-Carlo tests
Call: orthogram(x = body, tre = tree.primates)

Test number:   4
Permutation number:   999
   Test       Obs    Std.Obs   Alter Pvalue
1 R2Max 0.8475057  0.8184698 greater  0.344
2 SkR2k 1.2415354 -1.5846800 greater  0.979
3  Dmax 0.5975057  1.5069381 greater  0.211
4   SCE 0.5904470  1.5608456 greater  0.065
```

Only SCE, which measures the local variation in the r_i^2 values, is close to significance. However, the cumulative variance suggests a phylogenetic dependence (as could be shown with simulated data). The same analysis with longevity shows clearly an absence of phylogenetic dependence (Fig. 6.4):

```
> orthogram(longevity, tree.primates)
class: krandtest
Monte-Carlo tests
Call: orthogram(x = longevity, tre = tree.primates)

Test number:   4
Permutation number:   999
   Test        Obs    Std.Obs   Alter Pvalue
1 R2Max 0.40372034 -1.6978893 greater  0.992
2 SkR2k 2.30231793 -0.2756344 greater  0.607
3  Dmax 0.15372034 -0.3770156 greater  0.630
4   SCE 0.03702042 -1.2127811 greater  0.956
```

By default, `orthogram` uses an orthonormal basis computed from the topology of the tree computed by `treePart` and thus ignores branch lengths. This may be modified by using either the option `orthobas` to pass a user-defined basis, or `prox` to pass a matrix of phylogenetic proximities as returned by `proxTips`.

Desdevises et al. [54] proposed a method close to Diniz-Filho et al.'s [57]: instead of selecting the eigenvectors according to a broken-stick model, they suggested selecting all statistically significant eigenvectors in the regression.

Giannini [105] proposed a method with a matrix coding the tree structure similar to the one used by Ollier et al. [222]: 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.

6.1.4 Multivariate Methods

Jombart et al. [152] developed the phylogenetic principal component analysis (pPCA) which, by contrast to the methods described above, is really mul-

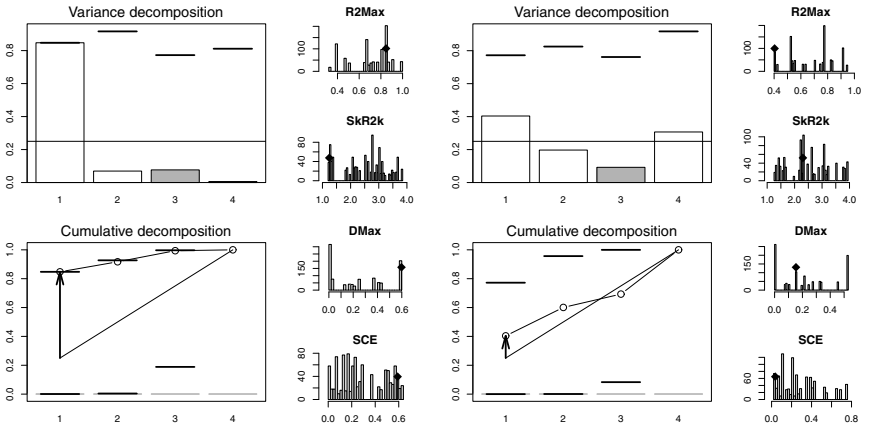

Fig. 6.4. Orthograms of the phylogenetic variance decomposition of body size (left) and longevity (right) for the primate data

tivariate in considering a matrix of data X. The principle is to perform a principal component analysis (PCA) using a weight matrix W as defined in Section 6.1.2. Specifically, a decomposition of $X^T W X$ is done, which is close to a PCA done by decomposition of $X^T V X$ where V is a variance-covariance matrix. Jombart et al. select the largest and smallest principal components of the pPCA. The largest components summarize the general covariance structure in the data, and are called "global axes": the variables contributing to these axes are likely to evolve in the same direction throughout the phylogeny. The smallest (negative) components summarize negative covariances in the data: the variables contributing to these "local axes" are likely to evolve in opposite direction for closely related species.

The function `ppca` in `adephylo` performs the pPCA. Its use is quite simple requiring to prepare the data as a `"phylo4d"` object. We illustrate its use with a random tree (`tr`) and two sets of five variables: one normally distributed (`X`) and one simulated along the tree with a Brownian motion model (`X2`). The commands are:

```
tr <- rtree(30)
X <- matrix(rnorm(150), 30, 5)
rownames(X) <- tr$tip.label
X2 <- replicate(5, rTraitCont(tr))
dat <- phylo4d(tr, X)
dat2 <- phylo4d(tr, X2)
res <- ppca(dat)
res2 <- ppca(dat2)
```

The results of the pPCA may be printed directly or with the `summary` method. However, the `plot` method is particularly informative (Figs. 6.5 and 6.6):

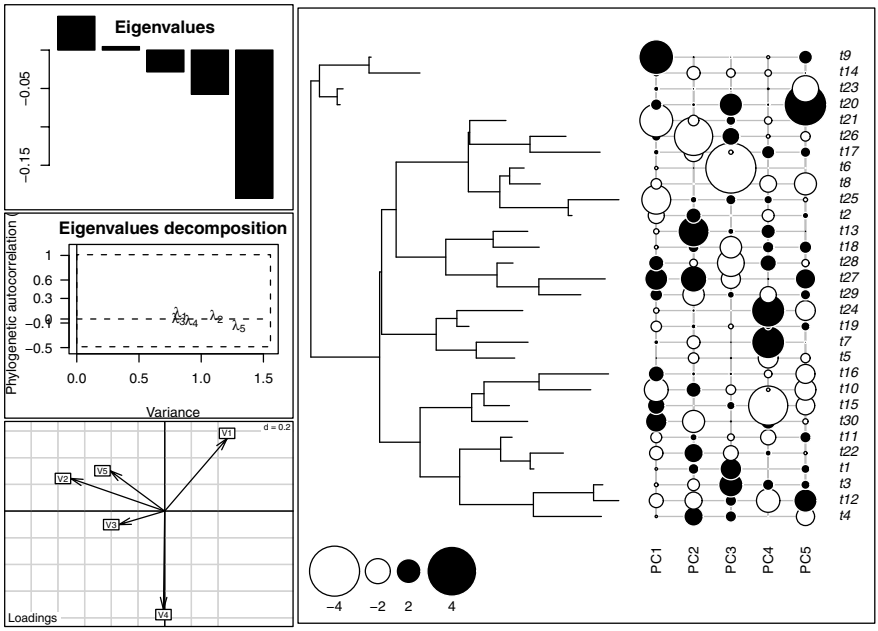

Fig. 6.5. pPCA of five random variables

```
plot(res)
plot(res2)
```

The contrast in the distribution of the eigenvalues is especially striking and shows that all eigenvalues tend to be positive when traits evolve along the phylogeny, even if they evolved independently.

Revell [266] independently developed a similar method: it uses a phylogenetic variance-covariance matrix (see Section 6.1.5) instead of a weight matrix, and the variables are centered with their respective phylogenetic means (p. 225) instead of the usual means in Jombart et al.'s pPCA. Revell's method was proposed in the context of correcting for body size prior to a phylogenetic regression as we see in the next section. Schmitz and Motani [210, 280], on the other hand, adapted a multivariate discriminant approach into a phylogenetic framework to devise a method called phylogenetic flexible discriminant analysis (pFDA). All these authors provide R code with their publications.

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.

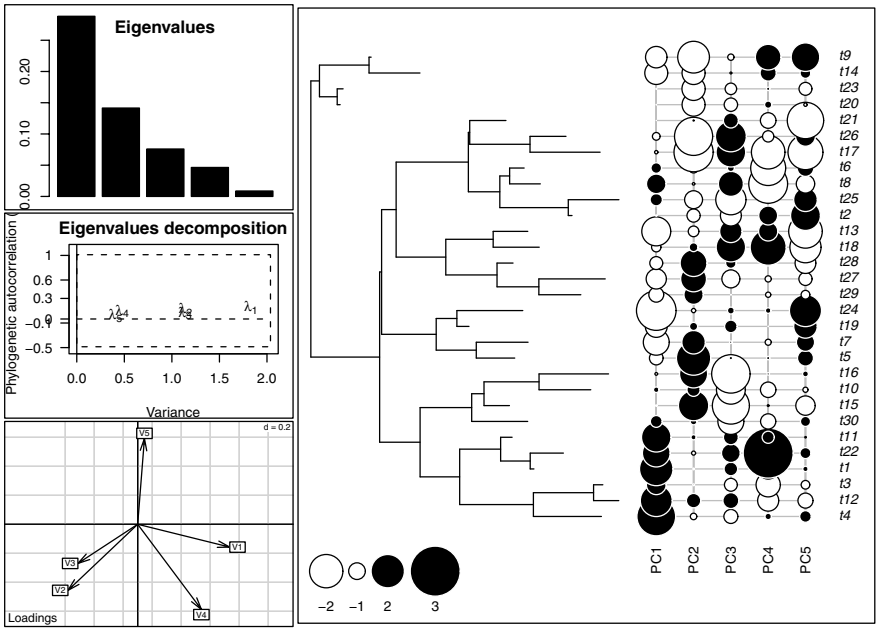

Fig. 6.6. pPCA of five random variables simulated with a Brownian motion model along the phylogeny

For a trait evolving along a phylogeny following a given model, it is possible to derive the theoretical variance-covariance matrix among species. This matrix quantifies how much species resemble each others (covariance), and how much they diverged from their common ancestor (variance). Using such a matrix in a GLS model fitting approach is therefore logical.

Models of Trait Evolution and Covariance Structures

The use of GLS in comparative methods came as a way to generalize the contrasts approach. Grafen [113] 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:

$$v_{ij} = \sigma^2 d_a \ , \tag{6.4}$$

where d_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 [199] suggested the Ornstein–Uhlenbeck model where the covariance between two species is given by:

$$v_{ij} = \frac{\sigma^2}{2\alpha} \exp(-\alpha d_{ij}) \,, \tag{6.5}$$

where σ^2 is similar to the variance of the Brownian process, α specifies the strength of the evolutionary contraint, and d_{ij} is the distance between both species. We shall see the Ornstein–Uhlenbeck model again in Section 6.1.8.

Two other modifications of the Brownian model have been proposed. Pagel [224] modifies the variance-covariance matrix by multiplying the off-diagonal elements (i.e., the covariances) with the parameter λ. Blomberg, Garland, and Ives [24] proposed the ACDC (accelerated/decelerated) model where traits evolve under a Brownian motion model which rates accelerates ($g < 1$) or decelerates ($g > 1$) through time. These two models reduce to the Brownian motion model if $\lambda = 1$ or $g = 1$, respectively.

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"`: 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 models sketched above are specified with the following functions:

```
corGrafen(value, phy, fixed = FALSE)
corBrownian(value = 1, phy)
corMartins(value, phy, fixed = FALSE)
corPagel(value, phy, fixed = FALSE)
corBlomberg(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). This last option is not available in `corBrownian` because the variance of the Brownian process is confounded with the residual variance of the regression model (see below). These functions return an object of class with three elements:

1. The name of the called function;
2. `"corPhyl"`;
3. `"corStruct"`.

The last one is important because it allows us to fit these models with `gls` or `gnls` for linear or nonlinear models.

Model Fitting by Residual Maximum Likelihood

The combination of the phylogenetic correlation structures with the model fitting functions of nlme gives an extremely flexible and powerful framework for the analysis of trait evolution.

For instance, coming back to the primate data, we first create a correlation structure assuming a Brownian motion model:

```
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`:[3]

```
DF.prim <- data.frame(body, longevity)
```

We can now fit the model:

```
library(nlme)
m1 <- gls(longevity ~ body, DF.prim, correlation = bm.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

 Correlation:
     (Intr)
body -0.437
```

[3] When this data frame is created, the names of the vectors are used as rownames (see p. 19); the latter are then matched with the tip labels of the tree, even if they are not in the same order.

```
Standardized residuals:
      Homo      Pongo     Macaca     Ateles     Galago
 0.4187373 -0.6395037 -0.1376075 -0.4269456  0.3844060
attr(,"std")
[1] 1.137666 1.137666 1.137666 1.137666 1.137666
attr(,"label")
[1] "Standardized residuals"

Residual standard error: 1.137666
Degrees of freedom: 5 total; 3 residual
```

Note that no parameter is estimated in the present correlation structure, hence the output `numeric(0)`. The item labelled "Correlation:" gives the correlation among the estimated parameters: it is computed from the variance-covariance matrix extracted from the fitted object (`m1$varBeta` or `vcov(m1)`). The "Standardized residuals" are the raw residuals divided by the residual standard error (the raw residuals can be output with `residuals(m1)`).

The estimated slope is equal to the slope of the regression through the origin with the contrasts (p. 207). This is not a coincidence: both analyses are identical and have the same number of degrees of freedom (five data points minus three estimated parameters with the GLS; four minus two with the contrasts).

The default fitting method of `gls` is residual maximum likelihood (REML). This differs from maximum likelihood (used if `method = "ML"`) in the way the variance parameters are estimated. With ML, the likelihood function is optimized over all parameters (regression coefficients and variance parameters) simultaneously. This leads to a negative bias in the variance parameters. REML takes into account the fact that some parameters need to be estimated to calculate the variance parameters. The same model fitted by either REML or ML will give exactly the same regression coefficient estimates and their associated standard errors.[4] On the other hand, the estimate of the residual standard error will always be different and smaller with ML than with REML. It is recommended to always use the default REML.

If one predictor is discrete (i.e., a factor), the coefficients are interpreted in the same way than for an analysis of variance. It could be shown that including a factor as a predictor in a GLS regression gives the same results than computing the corresponding PICs as described on page 209 and entering them as predictors in a standard linear regression. This further emphasizes the similarity between PICs and GLS.

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)
```

[4] The standard errors of the coefficients will be different for models including random effects; see `?anova.lme`.

```
> m2 <- gls(longevity ~ body, DF.prim, correlation = ou.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.

Variance Heterogeneity

In the present analysis, the standard deviation (`attr(,"std")`) is the same for all residuals and equal to the residual standard error. This is consistent with the assumption of homogeneous variance among the five species resulting from the ultrametric tree. It is perfectly reasonable to assume a non-ultrametric tree resulting in phenotypic rates of evolution proportional to the branch length units instead of time. In that case the expected variance of the species, as given by the distances from the root to the tips, will differ among species. Because `gls` works with a correlation matrix, heterogeneous variances must be modeled with the option `weights`. As an example, we replace the branch lengths of the tree with random numbers uniformly distributed between 0 and 1. We have then to recalculate the correlation structure; the diagonal of the corresponding variance-covariance matrix contains the expected variances:

```
> tr <- compute.brlen(tree.primates, runif)
> couni <- corBrownian(phy = tr)
> v <- diag(vcv(couni))
> vf <- varFixed(~ v)
```

The function `varFixed` returns an object of class `c("varFixed", "varFunc")`. The class `"varFunc"` in `nlme` is a general description of variance functions including cases where variances are modeled with respect to other covariates (see `?varClasses`). We now fit the model with the modified correlation structure and the variance function:

```
> summary(gls(longevity ~ body, DF.prim, couni, weights = vf))
Generalized least squares fit by REML
  Model: longevity ~ body
  Data: DF.prim
       AIC      BIC    logLik
  17.69167 14.98751 -5.845836
....

Standardized residuals:
      Homo      Pongo     Macaca     Ateles     Galago
 0.1406854 -0.5622591 -0.1986280 -0.5611175  0.5321529
attr(,"std")
    Homo    Pongo   Macaca   Ateles   Galago
1.545858 1.689781 1.438930 1.050152 1.121496
attr(,"label")
[1] "Standardized residuals"

Residual standard error: 1.247344
Degrees of freedom: 5 total; 3 residual
```

The standard deviation associated to each observation is now heterogeneous and given by the product of the expected variance and the residual standard error:

```
> sqrt(v) * 1.247344
    Homo    Pongo   Macaca   Ateles   Galago
1.545859 1.689781 1.438931 1.050152 1.121496
```

Coefficient of Determination

Researchers often want to compute the coefficient of determination of their phylogenetic regressions. Before exposing how this can be calculated, it is useful to remind that such coefficients, usually denoted as R^2, though very popular in the scientific literature, must be interpreted with care. Particularly, they depend directly on the range of the predictors so they have little predictive value—only the parameter values allow us to make predictions. Several definitions of the coefficient of determination are available and we whall consider three of them here.

The standard definition $1-\hat{\sigma}^2_{\mathrm{M}}/\hat{\sigma}^2_0$ can be used with GLS provided that the estimates of the residual variance of the model, $\hat{\sigma}^2_{\mathrm{M}}$, and of the null (intercept-only) model, $\hat{\sigma}^2_0$, are done with the same correlation structure. This way, the

value R^2 is the quantity of variation explained by the linear model taking the correlation among observations into account (otherwise it would be inflated).

A general definition of R^2 is based on the scaled likelihood-ratio [196]:

$$R^2_{\mathrm{LR}} = 1 - \exp\left[\frac{2}{n}(\ln L_{\mathrm{M}} - \ln L_0)\right] ,$$

where L is the maximum likelihood and the subscripts are as before. This has the advantage of being applicable to a wide range of models. Nagelkerke [212] suggested to adjust this coefficient by the maximum achievable value given by $1 - \exp(\frac{2}{n} \ln L_0)$ so that R^2_{LR} varies between 0 and 1.

The third definition is used by Lavin et al. [177] and taken from the econometric literature:

$$1 - \frac{(e^T V^{-1} e)}{(y - \bar{y}_\phi)^T V^{-1} (y - \bar{y}_\phi)} ,$$

where e is the vector of (raw) residuals from the GLS, V is the variance-covariance matrix, and $\bar{y}_\phi$ is the estimated phylogenetic mean of y:

$$\bar{y}_\phi = (\mathbf{1}^T V^{-1} \mathbf{1})^{-1} (\mathbf{1}^T V^{-1} y) . \tag{6.6}$$

These three definitions will give different, but usually close, values. They will not be correct if the wrong correlation structure is used: thus they must not be used for model selection.

The calculations in R are relatively simple. We first need to fit the null GLS model and then calculate the first and second versions of R^2:

```
> m1.0 <- gls(longevity ~ 1, DF.prim, correlation = bm.prim)
> 1 - (m1$sigma/m1.0$sigma)^2
[1] 0.2414281
> 1 - exp(-2*(m1$logLik - m1.0$logLik)/5)
[1] 0.2506952
```

The code for the "econometric" version is longer and gives a different result:

```
> iV <- solve(vcv(bm.prim))
> one <- rep(1, 5)
> e <- residuals(m1)
> y <- longevity
> m <- solve(t(one) %*% iV %*% one) %*% (t(one) %*% iV %*% y)
> 1 - (t(e) %*% iV %*% e)/(t(y - m) %*% iV %*% (y - m))
          [,1]
[1,] 0.4310711
```

As mentioned above, ignoring the correlation structure gives an inflated value of R^2:

```
> summary(lm(longevity ~ body))$r.squared
[1] 0.6882539
```

It could be that the first two versions give negative values in which case they should be considered as effectively equal to zero:

```
> tr <- rcoal(100)
> x <- rTraitCont(tr)
> y <- rTraitCont(tr)
> DF <- data.frame(x, y)
> cb <- corBrownian(phy = tr)
> m <- gls(y ~ x, DF, correlation = cb)
> m0 <- gls(y ~ 1, DF, correlation = cb)
> 1 - (m$sigma/m0$sigma)^2
[1] -0.00875768
> 1 - exp(-2*(m$logLik - m0$logLik)/n)
[1] -0.02647871
```

It is remarkable that with these two independent variables, but with phylogenetic correlation among observations, the ordinary R^2 value is very high:

```
> summary(lm(y ~ x))$r.squared
[1] 0.5318128
```

(Repeated simulations would show that the ordinary R^2 is highly variable whereas its GLS counterparts are stable around zero. This points out again on the limited usefulness of these coefficients: we already have all the needed information in the model structure and its estimated parameters.)

Nonlinear Models

The possibility to fit nonlinear models with `gnls` opens a range of possibilities that has been certainly underexploited in the evolutionary literature. The use of this function is similar to `gls` except that the model is specified as a nonlinear formula and starting values for the parameter estimation procedure must be given. (It is generally recommended to repeat the model fitting with different starting values to check the stability of the estimates).

Among the many possible models for the primate data, we will consider longevity as an exponential function of body size:

$$\text{longevity}_i = \exp(\beta\ \text{body}_i) + \alpha + \epsilon_i$$

This insures that longevity will always be positive (though it was originally log-transformed). The relationship will be increasing if $\beta > 0$ or decreasing if β is negative. If $\beta = 0$, longevity will not depend on body and will be equal to $1 + \alpha$. We first create the model formula (`mod`) and the list of starting values for the estimation (`init`), then we call `gnls` with the Brownian motion correlation structure:

```
> mod <- longevity ~ exp(beta * body) + alpha
> init <- list(alpha = 1, beta = 1)
> exm1 <- gnls(mod, DF.prim, start=init, correlation=bm.prim)
> summary(exm1)
Generalized nonlinear least squares fit
  Model: longevity ~ exp(beta * body) + alpha
  Data: DF.prim
      AIC      BIC    logLik
  15.4518 14.28011 -4.725899

Correlation Structure: corBrownian
 Formula: ~1
 Parameter estimate(s):
numeric(0)

Coefficients:
          Value Std.Error  t-value p-value
alpha 1.2721242 0.6498292 1.957628  0.1452
beta  0.3031309 0.0730883 4.147461  0.0255

 Correlation:
     alpha
beta -0.466

Standardized residuals:
       Homo       Pongo      Macaca      Ateles      Galago
 0.01419676 -0.98915677  0.04673264 -0.24629510  0.41572935
attr(,"std")
[1] 0.9380269 0.9380269 0.9380269 0.9380269 0.9380269
attr(,"label")
[1] "Standardized residuals"

Residual standard error: 0.9380269
Degrees of freedom: 5 total; 3 residual
```

Unsurprisingly, the AIC value is improved compared to the linear model `m1`. However, `gnls` uses maximum likelihood because REML works properly only for linear models.[5] We must thus refit the linear model by maximum likelihood to make a meaningful comparison:

```
> AIC(update(m1, method = "ML"))
[1] 17.38136
```

[5] https://stat.ethz.ch/pipermail/r-sig-mixed-models/2009q1/002104.html

The difference between ML and REML is slight and confirms the better fit of the exponential model. We now fit the exponential model assuming independence of observations:

```
> exm0 <- update(exm1, correlation = NULL)
> summary(exm0)
Generalized nonlinear least squares fit
  Model: mod
  Data: DF.prim
       AIC      BIC    logLik
  10.27032 9.098629 -2.135158

Coefficients:
          Value Std.Error  t-value p-value
alpha 1.4262893 0.3405199 4.188563  0.0248
beta  0.2535032 0.0485285 5.223802  0.0136
....
```

The decrease in AIC suggests that a significant nonlinear relationship exists between these two traits and is not influenced by the phylogeny (keep in mind that this latter characteristics concerns the relationship between the two variables, not the variables themselves). As a final comparison we compute the AIC for the standard linear regression assuming independence:

```
> AIC(update(m1, correlation = NULL, method = "ML"))
[1] 12.21108
```

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 [234].

GEEs were introduced by Liang and Zeger [183] 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:

$$g(E[y_i]) = x_i^T \beta \,, \tag{6.7}$$

However, the distinction comes from the way the variance–covariance matrix is given:

$$V = \phi A^{1/2} R A^{1/2} \,, \tag{6.8}$$

where A is an $n \times n$ diagonal matrix defined by $\mathrm{diag}\{\mathcal{V}(E[y_i])\}$: 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 $\mathcal{V}$ is the variance function. These two components, ϕ and $\mathcal{V}$, 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 (i.e., with 1's on the diagonal and 0's elsewhere).

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)
Beginning Cgee S-function, @(#) geeformula.q 4.13 98/01/27
running glm to get initial regression estimate
(Intercept)        body
  2.5989768   0.3425349
Call: compar.gee(formula=longevity ~ body, phy=tree.primates)
Number of observations:  5
Model:
                     Link: identity
 Variance to Mean Relation: gaussian

QIC: 9.23932

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` reports the quasilikelihood information criterion (QIC) which is an extension of AIC using quasilikelihood instead of the likelihood function and a penalty term that takes the correlation among observations into account [225]. Like in the case of the AIC with GLS, QIC is most appropriate to select an appropriate correlation structure.

If the argument `phy` is used the correlation structure under a Brownian motion is used. Instead, the argument `corStruct` may be used with a phylo-

genetic correlation structure. For instance, we repeat the above analysis with the correlation structure as estimated in Section 6.1.5:

```
> co <- corMartins(51, tree.primates, fixed = TRUE)
> compar.gee(longevity ~ body, corStruct = co)
Beginning Cgee S-function, @(#) geeformula.q 4.13 98/01/27
running glm to get initial regression estimate
(Intercept)        body
  2.5989768   0.3425349
Call: compar.gee(formula = longevity ~ body, corStruct = co)
Number of observations:  5
Model:
                      Link: identity
 Variance to Mean Relation: gaussian

QIC: 5.013872

Summary of Residuals:
        Min          1Q      Median          3Q         Max
-0.50364294 -0.40331899 -0.04356675  0.20702984  0.74349884


Coefficients:
             Estimate      S.E.        t Pr(T > |t|)
(Intercept) 2.5989768 0.3843447 6.762099  0.05532622
body        0.3425349 0.1330977 2.573561  0.18436023

Estimated Scale Parameter:  0.3379575
"Phylogenetic" df (dfP):  3.32
```

The value of QIC is improved but the relationship is not significant with these correlation structures.

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 [234]. A solution to this problem is to correct the number of degrees of freedom (`dfP`) with:

$$df_P = n \times \frac{\sum_{\text{tree}} \text{branch length}}{\sum_{i=1}^{n} \text{distance from root to tip}_i} , \tag{6.9}$$

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 [95, 96]. Lynch [190] 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:

$$x_i = \mu + a_i + e_i \,, \tag{6.10}$$

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_i's 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, V_a and V_e, namely the covariance explained by the phylogeny and the residual covariance, respectively [190].

Lynch proposed an expectation–maximization (EM [53]) algorithm to fit model (6.10) by maximum likelihood but this is very slow and becomes intractable with large sample sizes. Housworth et al. [140] proposed a reparameterization of (6.10) and a new algorithm to remedy this problem, but this applied only to uni- and bivariate cases.

Fitting model (6.10) 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 [242]. With phylogenetic data, there is only one group, and thus σ_a^2 and σ_e^2 are confounded. Felsenstein [82] used Lynch's model in the case where several measures of the same variable are available in each species (p. 244). Hadfield and Nakagawa [118] proposed an alternative approach based on Bayesian analysis, which can be generalized to other models, using the package MCMCglmm [117].

The function `compar.lynch` uses the EM algorithm proposed by Lynch [190] to fit model (6.10). We illustrate its use with the small primate data set. We first build a correlation matrix in the way seen previously:

```
> G <- vcv(tree.primates, corr = 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 which has two additional parameters compared to the Brownian motion model: the

"optimum" value (denoted as θ), and the strength of character evolution towards θ (α). The quantity of character change over a short time interval dt according to a general OU model is [18, 173]:

$$\mathrm{d}x_t = -\alpha(x_t - \theta)\mathrm{d}t + \mathrm{d}\epsilon_t \,, \tag{6.11}$$

where $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$. If $\alpha = 0$, the OU model reduces to a Brownian motion model. A discrete-time version of (6.11) is:

$$x_{t+1} = x_t - \alpha(x_t - \theta) + \epsilon_t \,. \tag{6.12}$$

It is straightforward to simulate an OU model in R using (6.12). If we set $\alpha = 0$, then we simulate a Brownian motion model with zero as initial value and $\sigma^2 = 1$, on 99 time-steps (d$t = 1$) 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 1:99)
    x[i + 1] <- x[i] - 0.2 * x[i] + 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 1:99)
        x[i + 1] <- x[i] - 0.2 * x[i] + 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 variances at $t = 100$ among the five replicates for each model:

```
> var(X[100 ,])
[1] 75.83865
> var(X2[100 ,])
[1] 0.8434638
```

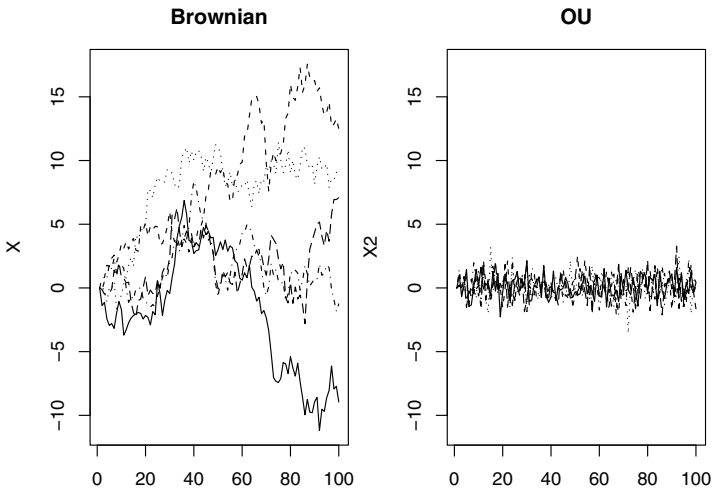

Fig. 6.7. Simulations with five replicates of the Brownian motion (left) and Ornstein–Uhlenbeck models (right)

A plot of the simulated values shows even more clearly the contrast between both models (Fig. 6.7):

```
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 `rTraitCont` gives a way to simulate an OU model along a phylogeny (Section 7.2).

Hansen [121] developed a methodology to estimate θ, α, and σ^2 by maximum likelihood from a phylogeny and trait values observed at the tips. The interesting feature of this is that θ may vary through the phylogeny leading to biologically relevant models which can be assessed by likelihood comparisons. This method is implemented in the function `compar.ou`:

```
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 made similar observations on the instability of the estimates of α [121].

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 = 0.2)
$deviance
[1] 13.88476

$para
        estimate   stderr
sigma2  3.698962 1.654556
theta1 16.720164 2.593057
....
> compar.ou(longevity, tree.primates, alpha = 2)
$deviance
[1] 12.41897

$para
        estimate    stderr
sigma2 0.7480796 0.3346408
theta1 3.6893089 0.3572333
....
```

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.

We illustrate now briefly how to fit a model with variable θ. We may want to fit a model where the optimum longevity is different for the three Old World primates (*Macaca*, *Pongo*, and *Homo*) than for the rest of the tree. We can find easily that this implies a shift in θ at node number 8 (Chapter 4). It is then simply required to pass this number to the argument `node`:

```
> compar.ou(longevity, tree.primates, alpha = 2, node = 8)
$deviance
[1] 9.869067

$para
        estimate    stderr
```

```
sigma2 0.4492266 0.2009895
theta1 4.8491740 0.5280661
theta2 3.0028738 0.3840086
....
```

The deviance has decreased but this is not significant:

```
> 1 - pchisq(12.41897 - 9.869067, 1)
[1] 0.110301
```

6.1.9 Phylogenetic Signal

The concept of phylogenetic signal is at the heart of most phylogenetic methods. From a statistical point of view, a phylogenetic signal is defined by the non null covariances (i.e., non independence) among species. From a biological point of view, phylogenetic signal is a direct consequence of the evolution of traits and its form will depend on the evolutionary mechanisms in action.

Testing for the presence of phylogenetic signal may seem trivial: traits characterize species, and species are related by their phylogenetic relationships, so logically phylogenetic signal is present in species traits (this is a consequence of Darwin's principle of descendance with modification). If the observed variation in a trait is completely determined by the environment, then phylogenetic signal will be zero (unless of course related species live in similar environments) even though the trait may have evolved among species. Similarly, if a trait evolves very slowly, little variation among species will be observed, and phylogenetic signal will be absent. Thus it appears the issue is not whether phylogenetic signal is present or not, but on its correct quantification and how to include it properly in among-trait analyses. An important point though is that phylogenetic signal depends on the context of the analysis so that it may differ if a trait is considered alone or in a regression model as we shall see below.

Section 6.1.3 detailed an approach to quantify phylogenetic signal using orthonormal decomposition. We shall see in the section other approaches to this issue.

Blomberg, Garland and Ives [24] formulated a way to quantify phylogenetic signal in a single trait based on the ratio of residual standard error with and without taking phylogenetic covariances into account. This is different, though with a similar logic, than the R^2 seen on page 224 where residual errors are calculated with and without predictors. In order to avoid confusion, I borrow the notation used in the literature and write Blomberg et al.'s ratio as MSE_0/MSE with MSE_0 being the usual variance of the trait and MSE is the mean squared error taking phylogeny into account.[6] For consistency, MSE_0 is calculated with the phylogenetic (not arithmetic) mean (6.6). For a

[6] MSE, and not MSE_0, is identical to $\hat{\sigma}_0^2$ in the definition of R^2.

given phylogeny and assuming that a trait evolved under a Brownian motion process, this ratio has expectation:

$$\frac{\sum_i v_{ii}^2 - \dfrac{n}{\sum_{i,j} v_{ij}^2}}{n-1} = \frac{\mathrm{tr}(V) - n(1^T V^{-1} 1)^{-1}}{n-1} .$$

This quantity will always be less than one. The statistic K is defined by the ratio of the observed $\mathrm{MSE}_0/\mathrm{MSE}$ on its expected value. If $K > 1$ then the covariance among species is stronger than expected under Browian motion evolution. If $K < 1$, then species tend to be independent with respect to their phylogenetic relationships.

In picante, `Kcalc` calculates K and `phylosignal` tests whether it is significantly different from 0 by randomization of the trait values among species:

```
Kcalc(x, phy, checkdata = TRUE)
phylosignal(x, phy, reps = 999, checkdata = TRUE, ...)
```

`x` must be a vector so only a single trait can be treated. To analyze several traits at once, we can build a list with the individual vectors and use `sapply`:

```
> x <- list(body = body, longevity = longevity)
> sapply(x, phylosignal, tree.primates)
                       body      longevity
K                      1.911770  0.7045068
PIC.variance.obs       5.2564    2.274950
PIC.variance.rnd.mean 13.69819   2.3597
PIC.variance.P         0.0105    0.57
PIC.variance.Z        -1.333335 -0.0780428

> sapply(x, Kcalc, tree.primates)
     body longevity
1.9117696 0.7045068
```

Another way to quantify phylogenetic signal in a single trait is to fit an intercept-only model by GLS: by comparing the fit assuming different correlation structures, it is possible to quantify phylogenetic signal in the trait. How to select an appropriate correlation structure with information criteria has been particularly investigated in the statistical literature [6, 44, 144, 167, 282]. The Akaike information criterion (AIC) is well-suited for this because it allows to compare non-nested models which is the case when comparing different correlation structures.

The procedure in R is relatively simple: fit the same model `body ~ 1` with different phylogenetic correlation structures and compare them with their respective AIC. We first create a function where the argument is the correlation structure passed to `gls`, and then the correlation structures we want to assess (we create only one here as the others have been created above):

```
> f <- function(cs) gls(body ~ 1, DF.prim, correlation = cs)
> pa.prim <- corPagel(1, tree.primates)
```

We now use `*apply` functions to fit the models and extract the AIC values:

```
> o1 <- lapply(list(NULL, bm.prim, pa.prim, ou.prim), f)
> sapply(o1, AIC)
[1] 23.209805 20.273722 20.733318  9.501006
```

The results are consistent with the above one with `phylosignal`. There is a clear phylogenetic signal in body mass: assuming no covariance among species (`NULL`) gave the worst fit. However, the present analysis allows us to go further in showing that an OU-type correlation structure gives a better description of the data than a Brownian motion one. This is consistent with the fact that K was significantly greater than one for this trait.

After editing `f` and replacing `body` for `longevity`, we repeat the analysis:

```
> o2 <- lapply(list(NULL, bm.prim, pa.prim, ou.prim), f)
> sapply(o2, AIC)
[1] 16.13314 16.92377 17.96771 18.08444
```

The results are again consistent with the above ones showing no evidence of phylogenetic signal in longevity. Since we have two lists of model fits, we can extract further results such as the estimated λ:

```
> o1[[3]]$modelStruct # lambda for 'body'
corStruct  parameters:
[1] 1.237311
> o2[[3]]$modelStruct # lambda for 'longevity'
corStruct  parameters:
[1] 0.4304575
```

Let us come back to the regression of longevity on body mass. We recall that we found a slightly smaller AIC value for a Brownian motion correlation structure than an OU-type one (Section 6.1.5). Thus this small data sets tells us an interesting story: whether we treat the variables separately or together we obtained different results relative to the phylogenetic signal. This is not surprising if we look at the models we are fitting here:

$$y_i = \beta x_i + \alpha + \epsilon_i \qquad \epsilon_i \sim \mathcal{N}(0, \sigma^2 C)$$

Here C is the correlation matrix derived from V. The model makes no assumption on the correlation structure among the x_i's, only on the error terms ϵ_i's. So we may find no phylogenetic signal in x alone, but require to include a phylogenetic correlation when fitting the above model.

On the other hand if x shows a phylogenetic signal (as is true for body), y may or may not show one depending on the error term which includes factors not identified by the researcher. This seems to explain the present results:

body mass has a phylogenetic component, and longevity is positively affected by body mass (the coefficient was not significant but recall that $n = 5$) and probably other non-phylogenetic factors. Similar results have been recently reported and widely discussed on mailing lists. Providing care is given in the analyses and interpretation, these will surely reveal interesting evolutionary patterns.

A graphical exploration of phylogenetic signal gives useful information because it can show the pairs of species that are more or less alike compared to what is expected under Brownian motion. Under this model of evolution, we expect a greater variance with greater phylogenetic distance. This suggests a simple graphical tool by plotting the Euclidean distances among species traits against the phylogenetic distances. As example, we consider random data with a normal variable (`x`) and a variable (`y`) simulated along the tree (Fig. 6.8):

```
tr <- rcoal(30)
x <- rnorm(30)
y <- rTraitCont(tr)
dphy <- as.dist(cophenetic(tr))
layout(matrix(1:2, 1))
plot(dphy, dist(x))
plot(dphy, dist(y))
```

The ultrametric nature of the tree is important here. If `rtree` is used in place of `rcoal`, the pattern would not be as clear albeit still present. For both variables, we can check the most appropriate correlation structure as done previously for the primate data:

```
> AIC(gls(x ~ 1))
[1] 93.2606
> AIC(gls(x ~ 1, correlation = corBrownian(1, tr)))
[1] 198.012
> AIC(gls(y ~ 1))
[1] -53.22712
> AIC(gls(y ~ 1, correlation = corBrownian(1, tr)))
[1] -114.6699
```

6.1.10 Intraspecific Variation

Though the present book focuses on issues at the level of species and above, the problem of intraspecific variation has come into comparative methods when addressing questions that consider explictly interindividual differences such as environmental niche breadth (Section 6.4.4). Most methods seen in the above sections can be extended fairly naturally to include intraspecific variation. I shall detail three approaches proposed in the literature.

Cornillon, Pontier and Rochet [46] proposed an approach based on the weight matrix used in autocorrelation models (Section 6.1.2). They build the

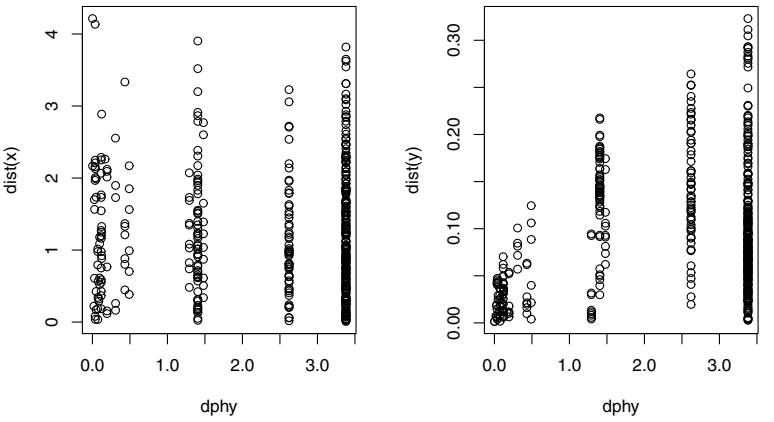

Fig. 6.8. Euclidean distances among observations for a random normal trait (left) and a trait generated by Brownian motion evolution (right) with respect to phylogenetic distances

W matrix with two components, W_b and W_w for the between and within species weights, respectively. These matrices have as many rows as there are populations. The elements of W_b are calculated as described in Section 6.1.2, and those of W_w are set to one if the two populations belong to the same species, zero otherwise [268]. As before, the diagonals are set to zero. Cornillon et al. used these matrices to fit an autoregressive model of the form:

$$Y = (\rho_b W_b + \rho_w W_w)Y \ .$$

As mentionned above, generalized least squares provides a much richer set of tools and models to study this kind of models.

Ives, Midford and Garland [149] used error measurement models to incorporate intraspecific variation in a GLS framework. This kind of models consider an additional error term:

$$y_i = \beta x_i + \epsilon_i + \eta_i \qquad \eta_i \sim \mathcal{N}(0, \sigma_m^2) \ ,$$

where σ_m^2 is assumed to be known (otherwise it would be confounded with the variance σ^2 of the ϵ_i's). In this situation, we do not need individual observations within each species, only a measure of intraspecific variances.

Fitting the model above is relatively easy with the function `varFixed` as explained in Section 6.1.5. Because this function requires a covariate, it is possible to specify a heterogeneous variance with `varFixed(~ vi)` where `vi` is a vector of length equal to the number of observations. Several variance functions can be combined with `varComb`, e.g.:

```
vf <- varComb(varFixed(~ v), varFixed(~ vi))
```

The output is, as above, passed as the `weights` argument to `gls`. As an example, suppose σ^2_m varies from 1 to 5 for the five species of primates:

```
> vi <- 1:5
> vf <- varFixed(~ vi)
> gls(longevity ~ body, correlation = bm.prim, weights = vf)
Generalized least squares fit by REML
  Model: longevity ~ body
  Data: NULL
  Log-restricted-likelihood: -6.175346

Coefficients:
(Intercept)        body
   2.485621    0.623599
....
Degrees of freedom: 5 total; 3 residual
Residual standard error: 0.7680576
```

This obviously decreases the estimated value of σ^2 because a part of the residual variance is interpreted as intraspecific variation.

Ives et al. [149] also developed a phylogenetic estimator for the slope of the reduced major axis (RMA, or type II regression, or geometric mean regression, GMR) in the bivariate case. The RMA is the regression line with the residuals projection orthogonal to the line itself instead of parallel to the y-axis. If the points are assumed to be independent, the slope and the intercept of the RMA are:

$$\hat{b} = \hat{\sigma}_y / \hat{\sigma}_x \qquad \hat{a} = \bar{y} - \hat{b}\bar{x} \,,$$

which are trivial to calculate with R. A phylogenetically corrected version of the slope estimator is:

$$\hat{b}_\phi = \sqrt{\frac{(y - \bar{y}_\phi)^T V^{-1} (y - \bar{y}_\phi)}{(x - \bar{x}_\phi)^T V^{-1} (x - \bar{x}_\phi)}} \,,$$

which is also simple to calculate with R ($\bar{x}_\phi$ and $\bar{y}_\phi$ are the phylogenetic means of x and y). If the variance-covariance matrices differ between x and y, the V's above would be substituted by V_y and V_x, respectively.

The third method is by Felsenstein [82] who extended his contrast method to consider situations when several observations are available for each species. This method computes contrasts with standardized coefficients so that they are an orthonormal transformation (i.e., geometrically independent). So instead of scaling the contrast with its variance as in (6.1), they are multiplied by a function of the coefficients calculated from the number of observations:

$$C_{ij} = \frac{x_i - x_j}{\sqrt{s_i + s_j}} ,$$

If i denotes an observed species, then $s_i = 1/n_i$ with n_i being the number of observations of the trait x on that species. If it is a node, the it is calculated recursively taking branch lengths and the orthogonality constraint into account. Ancestral values of x and their variance are calculated in the same way than for standardized contrasts. If several observations are available for species i $(x_{i1}, x_{i2}, \dots)$, the intraspecific contrasts are computed differently:

$$\begin{aligned}
C_{i1} &= \sqrt{\frac{1}{2}}(x_{i1} - x_{i2}) , \\
C_{i2} &= \sqrt{\frac{2}{3}} \left(x_{i3} - \frac{x_{i1} + x_{i2}}{2} \right) , \\
&\vdots \\
C_{in_i-1} &= \sqrt{\frac{n_i - 1}{n_i}} \left(x_{in_i} - \frac{1}{n_i - 1} \sum_{j=1}^{n_i-1} x_{ij} \right) ,
\end{aligned}$$

where n_i is the number of observations for species i. The normalizing constants are calculated so that the sum of squares of the coefficients in each contrast is one. These calculations assume that individuals within a species are linked by zero-length branches. Felsenstein [82] emphasized that these formulae are arbitrary, though other ways to compute the intraspecific contrasts are equivalent for a given analysis. However, a GLS approach offers a flexible way to model alternative correlation structures within a species (see below).

The function `pic.ortho` computes the orthonormal contrasts with the same arguments than `pic`:

```
> pic.ortho(body, tree.primates)
        6         7         8         9
3.7679120 1.0087441 1.2103664 0.3418296
> pic.ortho(longevity, tree.primates)
        6         7         8         9
1.0064693 0.7338977 0.5480856 0.9989510
```

The values are of course different than those returned by `pic` (p. 204). The option `intra = TRUE` allows one to return the intraspecific contrasts (which are actually not needed to compute the inter-species contrasts). In this case, the data must be arranged in a list where each element is a vector with the individual observations of each species (so that sample size may vary among species). We generate random data for the example below (and print the length of the simulated vectors):

```
> x <- lapply(sample(1:5, 5, TRUE), rnorm)
> sapply(x, length)
[1] 5 2 1 4 4
```

```
> pic.ortho(x, tree.primates, intra = TRUE)
         6          7          8          9
-1.2298571 -1.7143343 -0.7097605  2.4022110
attr(,"intra")
attr(,"intra")$Homo
[1] -0.22660428 -0.03895118 -0.97516058 -0.46674599

attr(,"intra")$Pongo
[1] 0.2000180

attr(,"intra")$Macaca
NULL

attr(,"intra")$Ateles
[1] -0.4508026  0.6215559 -0.7486237

attr(,"intra")$Galago
[1] -0.7789450  0.3658776 -0.1497702
```

The value `NULL` is returned if there is only one observation for that species.

Felsenstein further developed the model proposed by Lynch [190] (see Section 6.1.7) showing that when several traits are analyzed on a tree, the variance-covariance matrix of their orthonormal contrasts can be partitioned into a phylogenetic (A) and a phenotypic (P) components:

$$W \otimes A + I \otimes P \ ,$$

where W is a matrix with the variances of the contrasts. Felsenstein developed an EM algorithm to estimate A and P. This is implemented in Phylip [80] and the function `varCompPhylip` in `ape` calls this program to estimate these parameters in a flexible way:

```
> varCompPhylip(cbind(body, longevity), tree.primates)
$varA
          [,1]     [,2]
[1,] 3.676570 1.097208
[2,] 1.097208 0.328000

$varE
          [,1]     [,2]
[1,] 0.069079 0.152302
[2,] 0.152302 0.344948
```

The flexibility comes from the way the data are specified: they can be a vector (single trait and $n_i = 1$ for all species), a matrix or a data frame (multiple traits, $n_i = 1$), a list of vectors (single trait, $n_i \geq 1$), or a list of matrices (multiple traits, $n_i \geq 1$). The example above shows an analysis with our two

traits and five species. The results are indeed close to those obtained with `compar.lynch` (p. 231). The two examples below are with the list of vectors simulated above and with a single vector:

```
> varCompPhylip(x, tree.primates)
$varA
[1] 0.000193

$varE
[1] 0.720686

> y <- rTraitCont(tree.primates, sigma = 1)
> varCompPhylip(y, tree.primates)
$varA
[1] 0.997501

$varE
[1] 0.000128
```

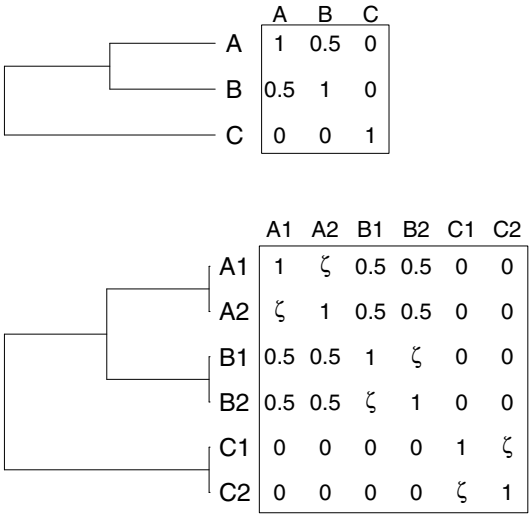

Fig. 6.9. A three-species phylogeny and its derived correlation matrix assuming a Brownian motion model of trait evolution (top), and the expanded phylogeny including intraspecific observations with the correlation matrix (bottom). Here the intraspecific correlation is parameterized with ζ

Because generalized least squares are equivalent to phylogenetically independent contrasts, it is natural to extend the orthonormal contrast approach to GLS using an appropriate intraspecific correlation structure. For instance, Fig. 6.9 shows on top a phylogeny and the corresponding interspecific correlation matrix assuming a Brownian motion model of trait evolution. The tree with six tips on bottom shows the same interspecific relationships but with two observations for each species. The corresponding correlation matrix shows an expanded block structure with in addition a parameter, denoted here as ζ, for the intraspecific correlation.

Once the correlation structure is established and parameterized, it is easy to include it in a GLS model fitting as seen in Section 6.1.5. Giving the tight correspondence between PICs and GLS, we may also expect a close relationship between the GLS approach sketched in the previous paragraph and the orthonormal contrasts method. However, a GLS approach offers a wide range of parameterizations for the correlation structure. For instance, one might want to test the hypothesis that $\zeta \neq 0$, or that ζ is the same for the three species.

The variance can also be modeled with separate variance functions parameterazing the interspecific (e.g., tree height) and intraspecific components, possibly heterogeneous among species. This again offers a wide range of models many of them being biologically interesting.

6.1.11 Phylogenetic Uncertainty

Phylogenies are models of the relationships among species, and like any model, they are subject to uncertainty. It might be necessary to take this phylogenetic uncertainty into account in comparative analyses, at least for the principle. There is no unique formal method for this task, and evolutionists tend to use a single phylogeny for their analyses. This may be a consequence of earlier works on this issue showing that using a wrong tree is better than using no tree at all [187, 197, 198]. Clearly, if there is some covariance among species, assuming a wrong covariance is closer to the truth than assuming none. Thus assessing the impact of uncertainties in phylogeny may seem more as a formal excercise than a really useful analysis.

Various approaches are possible here, and R makes them relatively easy to program with a few commands. They may be classified into two rough categories:

1. Starting from a list of trees taken from a phylogeny estimation (e.g., bootstrap or MCMC), the comparative analysis may be repeated by simply looping over the trees. Depending on the question of interest, the results may be summarized in different ways:
   - Plot or summary statistics of a parameter estimate or P-value. For instance, Duponchelle et al. [58] repeated a GEE-based regression with 100 bootstrap trees, and observed all $P < 0.05$ concluding that phylogenetic uncertainty did not affect their general conclusion.

- If the likelihood of each tree is available, it is possible to average a parameter estimate over all trees by weighting them with their likelihoods, thus giving more weight to the estimates derived from the most likely trees.

2. If no information is available on the phylogeny estimation and some priors on the distribution of the tree or of the among-species covariance matrix can be assumed, a Bayesian approach may be used.

Both categories are not exclusive. For instance, a list of trees may be used to compute a distribution of covariances for each pair of species. These may then be resampled by Monte Carlo and repeat at each step a GLS regression modeling. All these tasks do not require special functions. All that is needed are basic programming skills that can be found in Chapter 2.

As a small illustration, let us consider the Laurasiatherian data in phangorn. We do a simple NJ tree estimation assessing uncertainty with a bootstrap and 100 replications:

```
data(Laurasiatherian)
X <- as.DNAbin(Laurasiatherian)
f <- function(x) nj(dist.dna(x))
tr <- f(X)
o <- boot.phylo(tr, X, f, trees = TRUE)
```

From the 100 trees returned by `boot.phylo`, we want to assess how much the correlation matrix varies among these bootstrap replicates. Using `lapply`, we extract the correlation matrix with `vcv` that we transform directly into an object of class `"dist"`:

```
V <- lapply(o$trees, function(x) as.dist(vcv(x, corr=TRUE)))
```

`V` is thus a list with 100 elements. Fortunately, the 100 trees returned by `boot.phylo` have their tip labels in the same order; otherwise we would have to reorder the rows and columns of the matrices returned by `vcv`. To ease further analyses we transform `V` into a matrix with 100 columns:

```
V <- matrix(unlist(V), ncol = 100)
```

Thus for a given row, the 100 columns of `V` store the "bootstrap" samples of the same phylogenetic correlation element. We can now easily apply some functions on the rows of `V`, for instance, we can compute the means and standard deviations and plot them (Fig. 6.10):

```
plot(apply(V, 1, mean), apply(V, 1, sd))
```

This shows that overall the standard deviations are relatively small (< 0.09) so phylogenetic uncertainty will have little impact on the correlation matrix eventually used in GLS. Besides, the smallest correlations—which are the most important for a regression because they relate independent observations—are

the less variable. To conclude, this suggests that it might not be useful to replicate the regressions with these 100 trees; the analysis using the NJ tree only is likely to be enough.

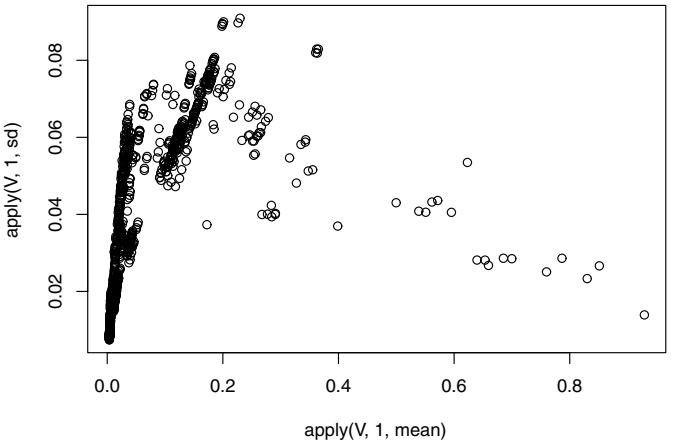

Fig. 6.10. Variation in among species correlations with 100 bootstrap replicates of the Laurasiatherian data. Each point is one of the 1081 elements of the correlation matrix showing its mean (x-axis) and standard-deviation (y-axis) calculated accross the bootstrap samples

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. With the development of alternative methods where ancestral character values are not necessary (distance methods) or their probabilistic distribution is taken into account (likelihood and Bayesian 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 [92]. Some researchers have focused their attention on statistical methods of ancestral character estimation where uncertainty in the estimates is taken into account [221]. 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" seems preferable to "reconstruction".[7] 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. geiger has alternative implementations with the functions `fitDiscrete` and `fitContinuous`. These functions use statistical estimation based on (generalized) least squares or maximum likelihood. ape has the function `MPR` that finds the sets of most parsimonious ancestral states for some discrete characters. Several functions are also available in phangorn to perform ancestral reconstruction for discrete characters under maximum likelihood and various parsimony methods.

6.2.1 Continuous Characters

Four methods can be used for continuous characters in `ace`: least squares (`method = "pic"`), maximum likelihood (`method = "ML"`, the default), residual maximum likelihood (`method = "REML"`), and GLS (`method = "GLS"`). The model of evolution is specified with the option `model`.

The least squares estimator follows from the phylogenetically independent contrasts method [76] (Section 6.1.1). This assumes a Brownian motion model of evolution: this allows us to compute the variance of each ancestral character estimate. This variance depends only on the tree topology and branch lengths because it is computed from its expectation under the model. A 95% confidence interval can be calcuted with the usual formula $\hat{x}_a \pm 1.96\sqrt{\text{var}(\hat{x}_a)}$, with $\hat{x}_a$ being the estimated ancestral value.

The maximum likelihood estimator under a Brownian motion model developed by Schluter et al. [279] uses a likelihood function where the ancestral values are parameters:

$$L(\sigma^2, x_a|\mathcal{T}, x) = \frac{1}{\sigma^n} \exp\left(\frac{1}{2\sigma} \sum_{i \neq j} \frac{(x_i - x_j)^2}{t_{ij}}\right) , \tag{6.13}$$

where σ^2 is the variance of the Brownian motion process, x_a is a vector of ancestral values, $\mathcal{T}$ is the phylogeny, and x is a vector of observed values of the character at the tips of $\mathcal{T}$. Once (6.13) has been maximized, the standard

[7] Strictly speaking, not all methods in this section do "estimation" of ancestral values: for instance, in the case of discrete characters, the likelihood function is maximized with respect to the rate parameters, the ancestral states are inferred afterwards.

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.

Equation (6.13) clearly shows that maximum likelihood estimation is done simultaneously on σ^2 and the ancestral values. This leads, in most cases, to a downward bias in the estimation of σ^2 in the same way than we have seen above about GLS (p. 222). The REML approach used by `ace` proceeds in three steps: first estimate the phylogenetic mean of x; then estimate σ^2 by maximizing a likelihood function under a multivariate normal distribution of the observed trait values; finally estimate the ancestral values using a modified version of (6.13) where σ^2 is fixed.

The GLS estimator of ancestral characters is [199]:

$$\hat{x}_a = V(x_a, x)V(x)^{-1}x \;, \tag{6.14}$$

where $V(x_a, x)$ is the expected covariance matrix between the ancestral values and the recent ones, $V(x)$ is the expected variance-covariance matrix among recent ones, and x is as before. These expected matrices are derived from the phylogenetic tree and the model of evolution assumed for x. Unlike the two other motheds, the model is specified as a correlation structure with the option `corStruct` (`model` is ignored if `method = "GLS"`); the way these structures are built is explained in Section 6.1.5. The variance of the GLS estimator is [199]:

$$\mathrm{var}(\hat{x}_a) = V(x_a) - V(x_a, x)V(x)^{-1}V(x, x_a) \;. \tag{6.15}$$

Let us try these four 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)

    Ancestral Character Estimation

Call: ace(x = body, phy = tree.primates)

    Log-likelihood: -6.714469

$ace
       6        7        8        9
1.183725 2.192018 2.571320 3.503182

$sigma2
[1] 1.9711502 0.6970463

$CI95
        [,1]     [,2]
```

```
6 -0.5058590 2.873308
7  0.9868737 3.397163
8  1.4844056 3.658235
9  2.6858445 4.320519
```

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. The same analysis with REML is:

```
> ace(body, tree.primates, method = "REML")

    Ancestral Character Estimation

Call: ace(x = body, phy = tree.primates, method = "REML")

    Residual log-likelihood: -7.094482

$sigma2
[1] 3.153838 1.99506

$ace
       6        7        8        9
1.183721 2.192012 2.571315 3.503179

$CI95
        [,1]     [,2]
6 -0.9534511 3.320893
7  0.6676123 3.716413
8  1.1964649 3.946165
9  2.4693199 4.537038
```

As expected the estimate of σ^2 is substantially higher. The ancestral values are very close to the previous ones, but the confidence intervals are wider because we assume a higher variance here. We now use the least squares method:

```
> ace(body, tree.primates, method = "pic")

    Ancestral Character Estimation

Call: ace(x = body, phy = tree.primates, method = "pic")
```

```
$ace
       6        7        8        9
1.183725 2.780824 3.200378 3.852630

$CI95
       [,1]     [,2]
6 -1.296931 3.664381
7  0.854866 4.706781
8  1.367000 5.033757
9  2.582428 5.122832
```

The least squares estimates are slightly larger than the maximum likelihood ones, particularly for the oldest nodes. Interestingly, the estimated value at the root is the same than above and equal to the phylogenetic mean. Furthermore, the confidence intervals computed by maximum likelihood or REML are usually narrower than those by least squares (remind that the latter depend only on the tree). Now on for the GLS analysis:

```
> co <- corBrownian(1, tree.primates)
> ace(body, tree.primates, method = "GLS", corStruct = co)

    Ancestral Character Estimation

Call: ace(x = body, phy = tree.primates, method = "GLS",
          corStruct = co)

$ace
       6        7        8        9
1.183725 2.192018 2.571320 3.503182

$CI95
      [,1]     [,2]
6 1.183725 1.183725
7 1.458418 2.925619
8 1.849527 3.293114
9 2.926902 4.079462
```

The ancestral values are identical to the maximum likelihood and REML ones.

`fitContinuous` uses a method similar to REML. For instance, in the case of the Brownian motion model, the function estimates σ^2:

```
> fitContinuous(tree.primates, body)
Fitting  BM model:
$Trait1
$Trait1$lnl
```

```
[1] -9.194399

$Trait1$beta
[1] 3.153841

$Trait1$aic
[1] 22.38880

$Trait1$aicc
[1] 34.3888

$Trait1$k
[1] 2
```

6.2.2 Discrete Characters

Markovian models provide a useful and practical tool for modeling the evolution of discrete characters [223]. We have already seen this framework with the substitution models of DNA and protein 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: $A \to B \to C \to A$.

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. 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 (5.9), 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)
> x.er <- ace(x, tree.primates, type = "discrete")
> x.er

    Ancestral Character Estimation

Call: ace(x = x, phy = tree.primates, type = "discrete")

    Log-likelihood: -1.768921

Rate index matrix:
  1 2
1 . 1
2 1 .

Parameter estimates:
 rate index estimate std-err
          1   0.3776  0.3058

Scaled likelihoods at the root
   (type 'x$lik.anc' to get them for all nodes):
        1         2
0.3047886 0.6952114
```

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:

```
> x.ard <- ace(x, tree.primates, type = "d", model = "ARD")
> x.ard
....

    Log-likelihood: -1.602901

Rate index matrix:
  1 2
1 . 2
2 1 .
```

```
Parameter estimates:
 rate index estimate std-err
          1   0.3060  0.3864
          2   1.0893  1.2244

Scaled likelihoods at the root....
      1       2
0.52689 0.47311
```

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
```

In practice, there is an `anova` method for the class `"ace"` which simplifies the calculation of this test:

```
> anova(x.er, x.ard)
Likelihood Ratio Test Table
  Log lik. Df Df change Resid. Dev Pr(>|Chi|)
1  -1.7689  1
2  -1.6029  2         1    0.33204     0.5645
```

`fitDiscrete` in `geiger` uses the same syntax than `ace` (`ER`, ...) and gives similar results:

```
> fitDiscrete(tree.primates, x)
....
$Trait1
$Trait1$lnl
[1] -2.462069

$Trait1$q
[1] 0.3775513
....
> fitDiscrete(tree.primates, x, "ARD")
....
$Trait1
$Trait1$lnl
[1] -2.296048

$Trait1$q
           [,1]       [,2]
[1,] -1.0892897  1.0892897
```

```
[2,]  0.3059753 -0.3059753
....
```

The log-likelihood values are scaled differently than in `ace` but the likelihood-ratio test is the same:

```
> 2*(-2.296048 - -2.462069)
[1] 0.332042
```

This is a reminder that likelihood values have no absolute meaning.

The package diversitree has also functions to perform ancestral state reconstruction: these are introduced and detailed, together with the general approach of this package on page 272.

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*,[8] 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")
....
    Log-likelihood: -2.772589

Rate index matrix:
  1 2
1 . 1
2 1 .

Parameter estimates:
 rate index estimate  std-err
          1  25.9881 15920.75

Scaled likelihoods at the root....
  1   2
0.5 0.5
> ace(y, tree.primates, type = "discrete", model = "ARD")
....
    Log-likelihood: -1.808865

Rate index matrix:
  1 2
1 . 2
```

[8] 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.

```
2 1 .

Parameter estimates:
 rate index  estimate std-err
          1  3350.683     NaN
          2 13402.728     NaN

Scaled likelihoods at the root....
  1   2
0.5 0.5
Warning message:
In sqrt(diag(solve(h))) : NaNs produced
```

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.

Parsimony reconstruction is based on minimizing the changes along the branches of tree; ancestral states might then be inferred. However there are, in many cases, several solutions to assigning ancestral states that would minimize the length of the tree. The method of most parsimonious reconstructions (MPR) tries to solve this problem by finding for each node the states that would minimize the number of changes along the tree. This method due to Hanazawa et al. [120, 213] is a generalization of the accelerated / decelerated transformation (ACCTRAN / DELTRAN) method by Swofford and Maddison [294]. `MPR` returns the most parsimonious reconstructions of discrete characters. The data must be in a vector of integers (or one that can be coerced as integers), and the tree must be unrooted and fully dichotomous. One of the tip must be defined as outgroup. For instance with our vector `x` defined above we have the obvious result:

```
> MPR(x, unroot(tree.primates), "Galago")
  lower upper
6     2     2
7     2     2
8     2     2
```

So there is only one most parsimonious tree with a transition $1 \rightarrow 2$ between *Galago* and the subtending node.

phangorn has two functions to do ancestral state reconstruction in the context of phylogeny estimation either by maximum likelihood or by parsimony (Chapter 5). The first function is `ancestral.pml` which has two arguments:

```
ancestral.pml(object, type = c("ml", "bayes"))
```

`object` is of class `"pml"` and `type` specifies the type of recontruction. By default, likelihood reconstruction (rescaled to one) is done. The result is an

object of class `"phyDat"` including the original data and the likelihood reconstructions for each node.

The function `ancestral.pars` has three arguments:

```
ancestral.pars(tree, data, type = c("MPR", "ACCTRAN"))
```

`tree` is of class `"phylo"`, `data` is of class `"phyDat"`, and `type` is as before. Like for the previous function, the returned object is an expanded object of class `"phyDat"` with the ancestral reconstructions, possibly with ambiguities.

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 [159]. 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. [216] 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 [230]. 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 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. [217] and Harvey et al. [126]. 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 [70]. 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.

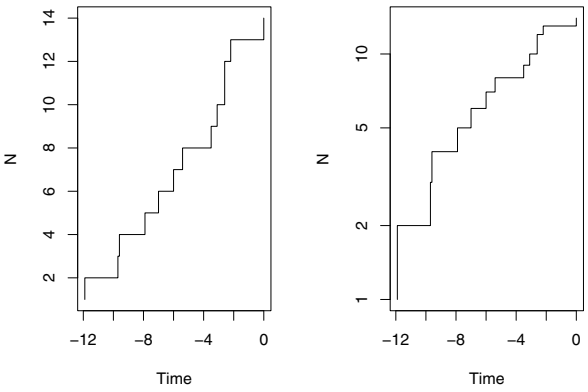

Fig. 6.11. Lineages-through-time plot of the clock tree of Michaux et al. [205] (Fig. 4.20) with a logarithmic scale on the right-hand side

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`, respectively. This function has also a '`...`' argument (see p. 88 for an explanation of this) 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. 4.20. 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. 6.11):

```
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 '`...`' 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. 4.20, 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. 6.12):

```
mltt.plot(trk, trc, dcol = FALSE, dlty = TRUE)
```

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 '`...`'). 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 The Simple Birth–Death and Yule 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 [188], 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 [159, 160].

The estimation of speciation and extinction probabilities when all speciation and extinction events are observed through time is not problematic [157]. Some difficulties arise when only the recent species are observed. Nee et al. [216] 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

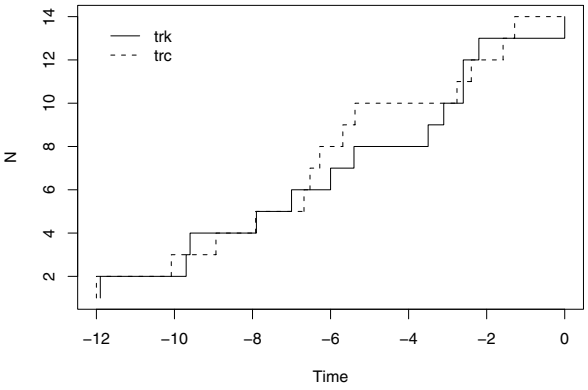

Fig. 6.12. Multiple lineages-through-time plot of the clock tree of Michaux et al. [205] and the tree estimated from the nonparametric rate smoothing method (Fig. 4.20)

of class `"phylo"`. Note that this tree must be dichotomous. If this is not the case, it could be transformed with `multi2di` (Section 3.4.4): 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 maxi-

mum. 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 [235]).

`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])`).

diversitree has an alternative function `make.bd`: this function returns a likelihood function that is subsequently optimized with `find.mle`:

```
> library(diversitree)
> trk.bd <- make.bd(trk)
> trk.bd
Constant rate birth-death likelihood function:
function (pars, ...)
ll.bd(cache, pars, ...)
<environment: 0xa0cca8>
> o <- find.mle(trk.bd)
> o
```

```
$par
      lambda           mu
1.419601e-01 6.512626e-08

$lnLik
[1] -12.71382
....

attr(,"class")
[1] "fit.mle.bd" "fit.mle"
```

diversitree does Bayesian estimation with its function `mcmc` starting from the likelihood function output by `make.bd`. This requires to give the initial values of the parameters, the number of steps of the chain, and the parameter `w` giving the approximate width of the high probability region. For instance, an MCMC with 10,000 steps taking the above maximum likelihood estimates as starting values would be:

```
> outmcmc <- mcmc(trk.bd, o$par, nsteps = 1e4, w=c(0.1, 0.1),
+                 print.every = 0)
> str(outmcmc)
'data.frame': 10000 obs. of  4 variables:
 $ i     : int  1 2 3 4 5 6 7 8 9 10 ...
 $ lambda: num  0.22 0.147 0.326 0.271 0.291 ...
 $ mu    : num  0.1062 0.0352 0.4256 0.2245 0.3231 ...
 $ p     : num  -13.5 -13 -17 -14.4 -15.5 ...
```

The posterior distributions can be displayed with `profiles.plot` (Fig. 6.13):

```
co <- c("darkgrey", "lightgrey")
profiles.plot(outmcmc[c("lambda", "mu")], col.line = co)
legend("topright", c(expression(lambda), expression(mu)),
       pch = 15, col = co, bty = "n")
```

The function `yule` in ape fits a simple Yule model (i.e., assuming $\mu = 0$) by maximum likelihood returning the estimate $\hat{\lambda}$, its standard-error, and the (maximized) log-likelihood:

```
> yule(trk)
$lambda
[1] 0.1438849

$se
[1] 0.04153599

$loglik
[1] -12.71274
```

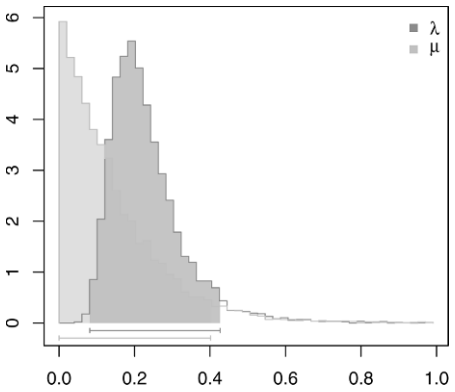

Fig. 6.13. Posterior distributions of speciation and extinction probabilities for the tree `trk`

```
attr(,"class")
[1] "yule"
```

diversitree has `make.yule` which works in the same way than above:

```
> find.mle(make.yule(trk))
$par
   lambda
0.1438849

$lnLik
[1] -12.71274
....
```

6.3.3 Time-Dependent Models

Speciation and extinction probabilities can vary at two levels: through time and among species. These two levels of variation have led to extend the basic birth–death model into two directions: time- and trait-dependent models. The latter is treated below (p. 271), while the former is the subject of the present section.

Several approaches have been developed in various packages: we are detailing them in order of increasing complexity. We first consider the Yule model which can be extended to have a speciation rate λ varying with time. laser

has four functions, `yule2rate`, `yule3rate`, `yule4rate`, `yule5rate`, fitting a model assuming constant speciation rates within two, three, four, or five time intervals (see [256] for the likelihood framework). We try the first one with a random Yule tree:

```
> tr <- rbdtree(0.1, 0, Tmax = 25)
> library(laser)
> yule2rate(branching.times(tr))
         LH      st1        r1       r2      aic
1 -17.56427 19.85357 0.2094644 0.108714 41.12853
```

By default, these functions take the breakpoint (or shift) time as one of the branching times: the one maximizing the likelihood function is taken as the estimate of the parameter denoted as 'st1'. This may be relaxed with the option `ints`.

A more flexible framework is provided by the function `yule.time` in `ape`, though it is more complex because it requires the user to build an R function coding the temporal variation in λ. Some examples are provided in the help page `?yule.time` where we take following the function to fit a model similar to the one used in `yule2rate`:

```
birth.step <- function(l1, l2, Tcl) {
    ans <- rep(l1, length(t))
    ans[t > Tcl] <- l2
    ans
}
```

where `l1`, `l2`, and `Tcl` are the same parameters than `r1`, `r2`, and `st1` above. The option `start` is used to specify starting values of the estimation procedure:

```
> yule.time(tr, birth.step, start = c(.1, .1, 15))
$estimate
        l1         l2        Tcl
 0.1295375  0.1049495 14.9994565

$se
        l1         l2        Tcl
0.04103592 0.02476942 0.13510211

$loglik
[1] -17.75767

attr(,"class")
[1] "yule"
```

The standard-errors are calculated with the traditional method using the second derivatives of the likelihood function. This fit can be compared with the one from a standard Yule model:

```
> yule(tr)
$lambda
[1] 0.1125819

$se
[1] 0.02127598

$loglik
[1] -17.89704

attr(,"class")
[1] "yule"
```

The increase in likelihood due to adding two parameters is very small (≈ 0.14) and does not require to compute the LRT.

Rabosky and Lovette [259] included extinction in the likelihood framework by developing three models of decreasing diversification caused by decreasing speciation rate, increasing extinction rate, or both. These are implemented in the functions `fitSPVAR`, `fitEXVAR`, and `fitBOTHVAR`, respectively. Like for the above functions in the same package, they take as main argument a set of branching times, and an option `init` to change the initial values of the likelihood optimization procedure. We try these functions with the random tree `tr` but just print their (optimized) log-likelihood value:

```
> fitSPVAR(branching.times(tr))$LH
[1] -17.83660
> fitEXVAR(branching.times(tr))$LH
[1] -17.88876
> fitBOTHVAR(branching.times(tr))$LH
[1] -17.83197
```

Unsurprisingly, no model provides a better fit compared to the Yule model.

An alternative model is to assume that diversification has decreased through time in relation to increasing diversity [217, 258]. This model, called the diversity-dependent diversification model, has a structure similar to the density-dependent models used in population ecology. Two possible formulations are:

$$\lambda_t = \lambda_0 N_t^{-x} ,$$
$$\lambda_t = \lambda_0 (1 - N_t/K) ,$$

where N_t is the diversity at time t, and x and K are parameters controlling the decrease in λ with respect to N_t. It is assumed that there is no extinction ($\mu = 0$). These two models reduce to the Yule model if $x = 0$ or $K = +\infty$. They are implemented in laser with the functions `DDX` and `DDL`, respectively:

```
> DDX(branching.times(tr))
$LH
[1] -17.82203

$aic
[1] 39.64405

$r1
[1] 0.1491700

$xparam
[1] 0.1095250
> DDL(branching.times(tr))
$LH
[1] -17.87188

$aic
[1] 39.74375

$r1
[1] 0.1219429

$kparam
[1] 206.3641
```

Again no significant increase in log-likelihood is observed.

It seems logical to attempt generalizing the above models into a framework where any time-dependent model of speciation and extinction could be fitted. diversitree provides `make.bd.t` that can fit a model defined by the user with R functions. For instance, if we want to fit a model where both λ and μ vary following a logistic function:

```
logis.t <- function(t, a, b) 1/(1 + exp(-a * t - b))
f <- make.bd.t(phy, list(logis.t, logis.t))
find.mle(f, c(0.02, -2, 0.02, -2))
```

However, such a likelihood estimation approach is problematic for some reasons I have exposed in a recent paper [232]. I proposed an alternative method that fits a model by minimizing the deviation between the expected and observed distributions of branching times with a least squares criterion. This method is implemented in the function `bd.time`. The user specifies the

speciation and extinction functions in the same way than with `yule.time` (p. 265) which is slightly different than above:

```
b.logis <- function(a, b) 1/(1 + exp(-a * t - b))
d.logis <- function(c, d) 1/(1 + exp(-c * t - d))
bd.time(phy, b.logis, d.logis)
```

`bd.time` returns the estimated parameters and the sum of squares of the fitted model. An F-test comparing two nested models can be computed with the ratio of the sum of squares divided by the number of additional parameters. The numbers of degrees of freedom are the number of additional parameters and the residual number of freedom. Confidence in the parameter estimates can be assessed with a bootstrap.

Currently, `make.bd.t` and `bd.time` run slowly for any slightly complicated model because they need to compute numerical integrals repeatedly during the fitting process. A detailed comparison showed that least squares work better than maximum likelihood in some situations than the former [232]. It will surely need more work to determine the respective merits of each method, and this is an exciting area of future research which will help us to know how much we can infer on evolution with phylogenies.

6.3.4 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. Two broad, non-exclusive situations can be defined.

- A phylogeny is resolved at high taxonomic levels and the number of species within these higher taxa is known.
- Monophyletic groups have been identified and the phylogenetic relationships among and within them are unknown; however, the date of the origin of each group is known.

We set aside the situation of complete ignorance where species have not yet been discovered.

Magallón and Sanderson [195] developed a simple and general framework that applies well to the second situation above. They used the fact that under a constant-rate birth–death model, the expected number of species after a time t is given by $\mathrm{E}(N_t) = N_0 e^{rt}$ with N_0 being the initial number of species and, as before, $r = \lambda - \mu$. If t is the crown age of the group, then $N_0 = 2$; otherwise if t is its stem age, then $N_0 = 1$. We have the estimator $\hat{r} = (\ln N_t - \ln N_0)/t$. Note that this a moment estimator, not a maximum likelihood estimator, because it is based on the expection of a quantity, not on the probability distribution of some observed variables.

`rate.estimate` in `geiger` implements Magallón and Sanderson's method. Its options are:

Table 6.1. Functions to fit birth–death models with respect to the type of data. This shows that inference is limited when a dated tree is not available. "Richness" means number of species. The functions separated by a comma fit the same model but with different algorithms, methods, or parameterizations

Model	Data		
	Dated tree	High-level dated tree + richness	Ages + richness
Constant rate			
μ fixed (= 0)	`yule`	`fitNDR_1rate`[a]	`rate.estimate`
μ estimated	`birthdeath`, `make.bd`	`bd.ext`	
Time-dependence			
μ fixed (= 0)	`DDX`, `DDL`	`fitNDR_2rate`[a]	
	`yule` *[2-5]* `rate`		
μ estimated	`fitSPVAR`		
	`fitEXVAR`		
	`fitBOTHVAR`		
General time			
μ fixed (= 0)	`yule.time`		
μ estimated	`bd.time`, `make.bd.t`		
Trait-dependence			
μ fixed (= 0)	`yule.cov`		
μ estimated	`make.bisse`	`make.bisse`	
	`make.musse`		
	`make.quasse`		

[a]In these two functions, μ is fixed by the user and can be different from zero

```
rate.estimate(time = 0, n = 0, phy = NULL, epsilon = 0,
          missing = 0, crown = TRUE, kendall.moran = FALSE)
```

`time` and `n` are two vectors giving t and N_t, respectively. By default, `time` are considered are crown ages: `crown = FALSE` results in treating them as stem ages. `epsilon` is the assumed value of extinction rate. `phy` is an alternative way to specify the data as a tree of class `"phylo"` (in which case a single estimate of r will be computed); `missing` is the number of species possibly absent from this tree. The last option allows one to calculate the Kendall–Moran estimate of the speciation rate if `phy` is given (which is the same as the one calculated by `yule`). As a simple example of this function:

```
> rate.estimate(10, 10)
[1] 0.1609438
> rate.estimate(10, 10, crown = FALSE)
[1] 0.2302585
```

The estimate assuming a stem age is higher because it implies one additional branching event for the same time interval. Given a pair of values for r and μ,

the function `crown.limits` calculates a 95% confidence interval of the extant number of species for a vector of times:

```
> crown.limits(r = 0.1609, epsilon = 0, time = 8:10)
         [,1]     [,2]
[1,] 2.411291 18.76913
[2,] 2.542183 22.31456
[3,] 2.702569 26.47092
> crown.limits(r = 0.1609, epsilon = 0.9, time = 8:10)
         [,1]     [,2]
[1,] 1.729496 106.3033
[2,] 1.898394 130.8073
[3,] 2.096817 159.5878
```

This clearly shows the increasing variance in diversity when extinction rate is increased even though the expected diversity $\mathrm{E}(N_t)$ is the same because r is kept constant. The function `stem.limits` does the same calculations but with $N_0 = 1$.

laser has four functions which perform exactly the same calculations than in geiger: these are `lambda.crown.ms01`, `lambda.stem.ms01`, `lambda.stem.ml`, and `lambda.stem.ci`.

In the first situation described above, the data are in the form of a phylogenetic tree with species richness associated with each tip. Pybus et al. [251] have approached this problem using simulations and randomization procedures. A more formal and general approach has been developed independently by Bokma [28] and myself [228]. 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 probabilities of species diversity are computed conditional on no extinction, thus avoiding a downward bias in extinction rate estimation [257].

The approach developed in [228] is implemented in the function `bd.ext` in ape. Let us consider the phylogeny of bird orders (Fig. 4.14). The number of species in each order can be found in Sibley and Monroe [285]. 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 [228] for examples).

laser has the function `fitNDR_1rate` that implements a method close to the one `bd.ext`, the distinction is that the parameter a ($= \mu/\lambda$) is kept constant (zero by default) [257]. The data must be prepared with the function `getTipdata`:

```
> fitNDR_1rate(getTipdata(S, bird.orders))
         LH      aic         r    lambda eps
1 -200.6427 403.2855 0.2839806 0.2839806   0
```

6.3.5 Trait-Dependent Models

In this section, we will see in details two relatively sophisticated approaches to analyze diversification with respect to one or several species traits. Ther represent another extension of the birth-death model with, in this case, speciation and extinction rates depending on variables observed on the recent species.

The Yule Model with Covariates

Modeling the variation in diversification rates with respect to one or several species traits is appealing biologically because a major issue in biology is to identify the biological traits that lead to higher speciation and / or extinction rates [92, 143].

I proposed [230] to model speciation rates using a generalized linear model (GLM) written as

$$\ln \frac{\lambda_i}{1-\lambda_i} = \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \alpha \,, \tag{6.16}$$

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 (6.16) 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:

$$\lambda_i = \frac{1}{1 + e^{-x_i^T \beta}} \,. \tag{6.17}$$

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 (6.16) giving an observed phylogeny and values of x. 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 [230]. 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 [230] 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. An application of these functions with the *Sylvia* data is detailed below (Section 6.6.1).

Binary and Multistate Traits

This section details some aspects of the package diversitree because these are integrated with the binary state speciation and extinction (BiSSE) model introduced by Maddison et al. [194]. This model considers a single binary trait which states are, for simplicity, denoted as 0 and 1. The speciation and extinction rates depend on this trait which itself evolves accoring to a Markov model. There are thus six parameters: λ_0, λ_1, μ_0, μ_1, q_{10}, and q_{01} where the subscripts 0 and 1 are the two states of the trait, and q_{ij} is the rate of the transition $i \to j$. See page 327 for simulating data from this model.

The first step to analyze some data with this model is to call `make.bisse` in order to build a likelihood function that will be subsequently treated by other functions. The basic usage is to pass a tree of class `"phylo"` and a vector, with names matching the tip labels of the tree, giving their states. Other options give information on a possible incompleteness of the data (see below). We

explore this function and others in diversitree with some completely random data:

```
> tr <- rcoal(20)
> x <- sample(0:1, size = 20, replace = TRUE)
> names(x) <- tr$tip.label
```

We build the likelihood function of the BiSSE model:

```
> library(diversitree)
> f <- make.bisse(tr, x)
> f
BiSSE likelihood function:
function (pars, ...)
ll.bisse(cache, pars, branches, ...)
<environment: 0xb6ec10>
```

`f` stores the information on the data and on the model. We fit the model by maximum likelihood with `find.mle`; this requires to give some initial values for the six parameters:[9]

```
> out <- find.mle(f, rep(0.1, 6))
> out
$par
     lambda0      lambda1          mu0          mu1
4.122753e-07 1.497310e+01 1.047648e+01 6.740341e+00
         q01          q10
7.058190e+02 6.832535e+02

$lnLik
[1] -10.88618
....
```

The estimates of q_{10} and q_{01} are large which is typical when state values are randomly distributed among the tips of the tree. The model we have just fitted can be seen as the "full" BiSSE model with six parameters. The function `constrain` allows us to define more restricted models (i.e., more parsimonious in the statistical sense), and so to test hypotheses such as: Are μ_0 and μ_1 significantly different? We call `constrain` with the likelihood function of the full model, and a formula that defines a contraint on the parameter(s). We may then fit this new model:

```
> f.mu <- constrain(f, mu1 ~ mu0)
> out.mu <- find.mle(f.mu, rep(0.1, 5))
> out.mu
$par
```

[9] diversitree has the function `starting.point.bisse` to help finding initial values.

```
      lambda0      lambda1          mu0          q01
3.358217e-05 1.467586e+01 8.435833e+00 7.378969e+02
         q10
7.060755e+02

$lnLik
[1] -10.88675
....
```

The decrease in log-likelihood is very small. Fortunately, diversitree provides an `anova` method to calculate easily the tests (see below).

It is possible to define a third, more parsimonious, model by adding the additional constraint $\lambda_0 = \lambda_1$:

```
> f.lambda <- constrain(f.mu, lambda1 ~ lambda0)
> out.lambda <- find.mle(f.lambda, rep(0.1, 4))
```

Instead of building a series of models of increasing parsimony, we could add several constraints at once by giving several formulae separated by commas with the likelihood function of the full model:

```
> f.null <- constrain(f, lambda1~lambda0, mu1~mu0, q10~q01)
```

It is also possible to constrain a parameter to a fixed value (e.g., `mu1 ~ 0`), or proportional to another parameter (`lambda1 ~ 2 * lambda0`). We finally fit the null model, and compare all the fitted models with `anova`:

```
> out.null <- find.mle(f.null, rep(0.1, 3))
> anova(out, out.mu, out.lambda, out.null)
        Df   lnLik    AIC   ChiSq Pr(>|Chi|)
full     6 -10.886 33.772
model 1  5 -10.887 31.774 0.00116     0.9729
model 2  4 -11.408 30.817 1.04423     0.5933
model 3  3 -11.458 28.916 1.14355     0.7666
```

As expected, none of the variation in λ, μ or q can be explained by `x`.

As seen above for the simple birth-death model (p. 263), Bayesian estimation can be done with the BiSSE model with the same function `mcmc`. A typical usage could be (using the maximum likelihood estimates as starting values):

```
mcmc(f, coef(out), 1e3, rep(0.1, length(coef(out))))
```

This appears to be relatively slow with the current version of diversitree. However, a nice feature of this function is that if it is interrupted before the completion of the chain, then the results until this stage are returned.

The original formulation of the BiSSE model [194] assumes that the phylogeny is fully known. FitzJohn et al. [86] extended this to the case of incompletely resolved phylogenies. This case is similar to the first situation described

in Section 6.3.4: some tips of the tree represent clades whose number of species are known. In addition here, it is known how many species are in the states 0/1 in each clade. The third argument of `make.bisse` is `unresolved` and, if used, this must be given a data frame including four columns named `tip.label`, `Nc`, `n0`, and `n1`. These are, respectively, the labels of the tips that are actually clades with several species, the number of species in these clades, and the number of species that are *known* to be in state 0 or 1. `n0` and `n1` may be equal to zero. For instance, suppose that the tips "t1" and "t2" are clades with ten species each.

```
> u <- data.frame(tip.label = I(c("t1", "t2")),
+                 Nc = c(10, 10), n0 = 0, n1 = 0)
```

The function `I()` serves to "isolate" the character vector because by default character strings are treated as factors in data frames (alternatively one could do `u$tip.label <- as.character(u$tip.label)` after calling `data.frame`). We now define and fit a BiSSE model with these new data:

```
> f.u <- make.bisse(tr, x, u)
> out.u <- find.mle(f.u, rep(0.1, 6))
> out.u
$par
     lambda0      lambda1          mu0          mu1
1.736204e-04 5.397004e+01 3.560434e-04 5.635014e+01
         q01          q10
4.560002e+01 3.762528e+01

$lnLik
[1] -21.02996
```

To get an idea of the potential consequence of taking this phylogenetic uncertainty for our inference, we define and fit a null model and compare it with the full one:

```
> f.u.null <- constrain(f.u, lambda1~lambda0, mu1~mu0, q10~q01)
> out.u.null <- find.mle(f.u.null, rep(0.1, 3))
> out.u.null
$par
  lambda0       mu0       q01
 23.69290  24.87693 679.78359

$lnLik
[1] -23.46765
....
> anova(out.u, out.u.null)
        Df   lnLik    AIC  ChiSq Pr(>|Chi|)
full     6 -21.030 54.060
```

```
model 1  3 -23.468 52.935 4.8754     0.1812
```

The null model is again the best, but the estimates of λ and μ are much higher because the number of species is now 38 instead of 20, for the same tree. This result would be quite different if we had specify `n0 = 10` in `u`.

Once a BiSSE model has been fitted, it is possible to calculate the ancestral values of the binary trait with the function `asr.marginal`:[10]

```
> asr.marginal(f.null, coef(out.null))
          [,1] [,2] [,3]      [,4] [,5] [,6] [,7] [,8]
[1,] 0.5000001  0.5  0.5 0.5000001  0.5  0.5  0.5  0.5
[2,] 0.4999999  0.5  0.5 0.4999999  0.5  0.5  0.5  0.5
          [,9]     [,10] [,11] [,12] [,13]     [,14] [,15]
[1,] 0.5000001 0.4999999   0.5   0.5   0.5 0.4999988   0.5
[2,] 0.4999999 0.5000001   0.5   0.5   0.5 0.5000012   0.5
         [,16]     [,17]     [,18]      [,19]
[1,] 0.4999999 0.5000001 0.4999995 0.03554341
[2,] 0.5000001 0.4999999 0.5000005 0.96445659
```

It could be checked that very close values are obtained with `ace`. We check this by printing the estimated value of the rate of change of `x` (q):

```
> coef(out.null)["q01"]
     q01
679.2548
> ace(x, tr, type = "d")$rates
[1] 680.1794
```

Also, the same results are found with the unresolved data (not shown).

The BiSSE model is extended by four other functions still under development in diversitree. `make.musse` implements the multistate speciation and extinction (MuSSE) model where the trait influencing λ and μ has more than two states (which is equivalent to consider multiple binary traits). `make.quasse` implements the quantitative speciation and extinction (QuaSSE) model where a single quantitative trait affects λ and μ [85]. At the moment these two functions require a fully resolved phylogeny (i.e., the option `unresolved` is not available, but see the option `f.sampling` described in the next paragraph). `make.bisse.split` implements a BiSSE model with parameters changing along the tree (specified with the option `nodes`). It can consider unresolved phylogenies with the option `unresolved`. The fourth function, `make.bisse.td`, implements a time-dependent version of BiSSE with its option `n.epoch`. It has also an option `unresolved`. The last version of diversitree (0.7-2) includes similar functions for the MuSSE model, `make.musse.split` and `make.musse.td`, and two functions which extend `make.bd.t` (p. 267), `make.bisse.t` and `make.musse.t`.

[10] diversitree has also the functions `asr.joint` and `asr.stoch` to do stochastic reconstructions but they currently work only with `make.mk2`.

The functions of diversitree listed in this section can take uncertainty in the trait into account with the option `sampling.f`. It gives the proportion of species in each state that have been included in the phylogeny. This is a way to include information from species not included in the tree. If the state of a species included in the tree is unknown, it must be `NA` in the appropriate vector (which is the case for unresolved clades).

The Yule model with covariates (`yule.cov`) and the MuSSE class of models have their own respective strengths. The former can model several traits simultaneously—which has proved useful in the case of the effect of body mass on diversification of primates [230]. The fact of assuming no extinction leads to fast model fitting, and fairly large data sets may be handled easily. At the moment, no approach has been developed to include missing data in the Yule model with covariates, though some simple ad-hoc solutions are possible.

The MuSSE class of models has the advantage of modeling extinction explicitly, though with fairly large variance in the estimates [194], and the availability of several tools to take data uncertainty into account. Both maximum likelihood and Bayesian estimation are possible (though Bayesian estimation can be done with the Yule model with covariates).

6.3.6 Other Methods

This section develops two methods developed in the late 1990's. They are slightly outdated given the recent developments and extensions of the birth–death model.

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 [226].

Typical survival data are times to failure of individuals or objects [47]. 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.

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. It is denoted δ ($= \lambda - \mu$). In theory a variety of models could be used, but only three are implemented in ape (see [226] 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;[11]
- 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 [226]. This genus of passerine birds includes eight species: six of them were studied by Hackett [116] who resolved their phylogenetic relationships. For the two remaining species, some approximate dates of branching could be inferred from data reported in [116]. 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
```

[11] R uses a different parameterization of the Weibull distribution (see `?Weibull`): the scale parameter, denoted as b, is equal to $1/\beta$.

```
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 β (`beta`) and the values of the LRTs are scale independent.

In spite of its simplicity, this method has some good statistical performance in terms of type I and II error rates, and its χ^2 tests are quite powerful with small trees. It has also the merit to be easily computed.

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 [229]. However, it is clear that extinctions affect the distribution of branching times of a given tree [126, 214]. 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) [291, 295]. 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 [227]. 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 [292]. The critical values of both tests have been determined by Stephens [291]. 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 [250]. 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)))
```

6.3.7 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 [5, 10, 166] 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:

$$I_S = \sum_{i=1}^{n} d_i \ ,$$

$$I_C = \sum_{j=1}^{n-1} |L_j - R_j| \ ,$$

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 page 319.

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, 50, model = "yule")[[1]]
> 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 [9, 10] introduced a graphical method where, for each node, the number of descendants of both subclades from this node are plotted one *versus*

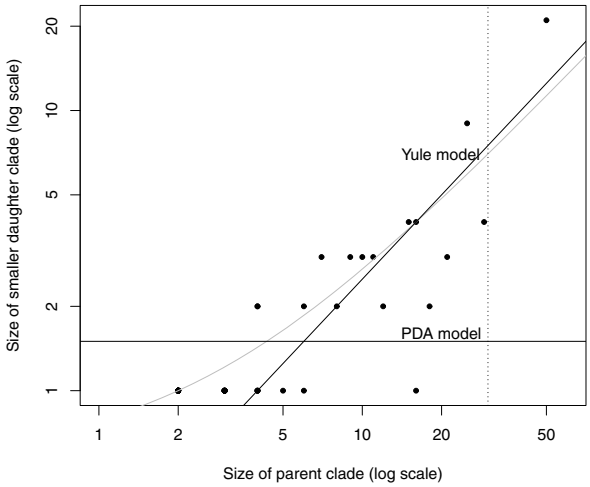

Fig. 6.14. Plot of the number of descendants of both subclades for each node of a tree with 50 tips simulated under a Yule model. The labeled lines are the expected distribution under these models, and the grey curve is a quantile regression on the points

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` 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. 6.14):

```
aldous.test(trs, xmin = 1)
```

The expected lines are labeled with the null models just above them. The grey curve is a quantile regression on the points.

6.3.8 Tests of Diversification Shifts

Several methods have been developed to test for the presence of shifts in diversification using various kinds of data. One of the earliest tests was proposed by Slowinski and Guyer [286] by analyzing species richness in a series of sister-clades where one is characterized by a trait or a variable that is supposed to affect the rate of diversification. The null hypothesis is that the trait has no effect on diversification, whereas the alternative hypothesis is that the trait increases diversification rate (this is a one-tailed test: it cannot detect

if the trait decreases diversification). The test is implemented in the function `slowinskiguyer.test` taking as argument a matrix with two columns giving the species richness with and without the trait, respectively—each row being a pair of sister-clades.

Several researchers have pointed out that the Slowinski–Guyer test is prone to type II error when richness data are overdispersed [111, 304]. In addition, I have shown that this test has relatively low statistical power—and so is prone to type I error—in a wide range of realistic situations [233]. Three alternative tests are available in `ape`.

The diversity contrast test is nonparametric and can be written in a generic form as $\text{sign}(x_{1i} - x_{2i}) \times C_i$ where x_{1i} and x_{2i} are the numbers of species from the ith pair of sister-clades and C_i is a contrast calculated with the x_i's [304]. Four variants of this test have been proposed in the literature [19, 20, 277, 313]. An extensive simulation study showed that it is the simplest form with $C_i = |x_{1i} - x_{2i}|$, as proposed by Sargent [277], that has the best statistical proporties [233]. The four versions are available in the function `diversity.contrast.test`. By default, the null hypothesis that the distribution of the diversity contrasts is centered on zero is tested with a Wilcoxon test. If the option `nrep` is used, then a randomization test is done. The test can be one- or two-tailed.

The McConway–Sims test [201], implemented in `mcconwaysims.test`, is based on a likelihood-ratio test. This is a two-tailed test. The likelihood function assumes a geometric distribution of species richness. This test suffers from the same low power and lack of robustness than the Slowinski–Guyer one [233].

I have proposed a new test based on the distribution of species richnesses from a Yule model. The probability to observe n species after a time t from an initial single species is given by:

$$\mathrm{e}^{-\lambda t}(1 - \mathrm{e}^{-\lambda t})^{n-1} \ .$$

A simple likelihood-ratio test may be calculated by fitting a null model with a single value of λ and an alternative model with λ depending on the trait. This is implemented in the function `richness.yule.test`. Some simulations have shown it is more robust and powerful than the previous tests even when the values of t are unknown [233]. The data considered by these tests are easily simulated, for instance:

```
> n0 <- replicate(10, Ntip(rbdtree(.1, .09, Tmax = 35)))
> n1 <- replicate(10, Ntip(rbdtree(.15, .1, Tmax = 35)))
```

`n1` contains the species richnesses in clades where diversification has increased. We then bind the two vectors of species richnesses putting the clades with the trait supposed to increase diversification first (the order is important for the Slowinski–Guyer test):

```
> x <- cbind(n1, n0)
> slowinskiguyer.test(x)
     chisq df     P.val
 28.18434 20 0.1051071
> mcconwaysims.test(x)
     chisq df     P.val
 8.151978 10 0.6139948
> richness.yule.test(x, 100)
     chisq df       P.val
 6.739111  1 0.009432162
```

Note the robustness of the Yule-based test to the presence of extinctions even though we speficied a wrong value of t (100).

These four tests require to identify a priori the clades that are susceptible to show increased diversification. Two other tests try to find the nodes in a phylogeny where diversification has increased without identifying them a priori. They are implemented in the functions `rc` in geiger and `shift.test` in apTreeshape. The first function implements a method developed by Purvis et al. [249] based on the same principle than the Slowinski–Guyer test: at a given time it evaluates all combinations that are less favorable to the null hypothesis of equal diversification in all lineages. Arbitrarily, the time points where the test is conducted are the branching times of the tree, so the analysis essentially tests, for each node, whether diversification has increased in this clade compared to the rest of the tree (Fig. 6.15).

The `shift.test` function provides an alternative test developed by Moore et al. [207] based on the relative likelihoods of the distributions of species richness comparing the two hypotheses with or without shifts in diversification at a particular node. By contrast to Purvis et al.'s method, the present one does not require branth lengths. It relies on calculating for each node of the tree the Δ_1 statistic:

$$\Delta_1 = \Lambda_O - \Lambda_I \ ,$$
$$\Lambda = \ln L(x|\mathrm{H}_1) - \ln L(x|\mathrm{H}_0) \ ,$$

where Λ is the likelihood ratio of the hypothesis, H_1, that diversification rate shifted at the node against the null hypothesis, H_0, of homogeneous diversification, and x are the data (i.e., the species richness of the outgroup and the two sister-clades of the ingroup). The likelihood ratio is calculated at the node ancestor of the ingroup and the outgroup (Λ_O), and at the node ancestor of the ingroup (Λ_I; Fig. 6.15). The significance of Δ_1 is assessed with a Monte Carlo procedure.

These two methods are mainly exploratory, and their results should be taken as indications where diversification has been heterogeneous in a tree. The interpretation of the level of significance of the possibly numerous tests can be problematic [30].

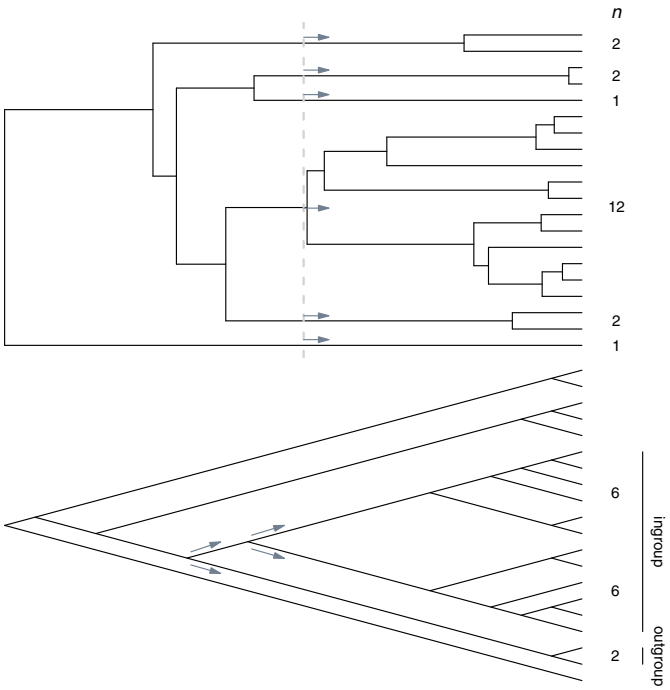

Fig. 6.15. Analysis of a phylogeny with Purvis et al.'s (top) and Moore et al.'s (bottom) methods. The first method considers the number of descendants of the lineages at a point in the past (e.g., the dashed grey line). The test calculates the probability that the lineage with twelve surviving species had a higher diversification rate compared to the five other lineages present at the same time. The present test is significant ($P = 0.029$). Moore et al.'s method calculates the likelihood ratios Λ_O for the pair {12,2} and Λ_I for the pair {6,6}, and is slightly significant ($\Delta_1 = 1.999$, $P = 0.049$). The present tree was simulated using a Yule model with $\lambda = 0.05$

Alfaro et al. [11] developed a more sophisticated approach similar to stepwise regressions. The analysis starts by fitting a constant-parameter model which, in a second step, is complexified by adding a shift in parameters at a node. Each node is tested in turn, and the one giving the best significant increase in model fit is selected. More shifts are added in the subsequent steps. A model is selected based on an increase in the AIC value, hence the name of the method: MEDUSA (modeling evolutionary diversification using stepwise AIC). In its current implementation, there is no objective stopping criterion: diversification shifts are added until a limit specified by the user is reached. The function in `geiger` is named `runMedusa`:

```
runMedusa(phy, richness, estimateExtinction = TRUE,
          modelLimit = 20, cutAtStem = TRUE,
```

```
                startR = 0.05, startE = 0.5, ...)
```

`phy`, as usual, is a tree of class `"phylo"`, and `richness` is a data frame with the taxa names (matching the tip labels of `phy`) and the species richnesses. The third option specifies whether to estimate extinction rate (the default), or to assume $\mu = 0$; `modelLimit` is the number of models output; `cutAtStem` specifies if the shift concerns also the basal branch of the clade (the default); and the last two options give starting values for the estimation process. The model fitted is a birth–death model combining phylogenetic and taxonomic data (Section 6.3.4). There is also a function `summaryMedusa` to summarize the output of `runMedusa` and plot a phylogeny displaying the branches where a significant shift in diversification was found.

`fitNDR_2rate` in laser does a similar analysis with only a single shift in the tree. Its use is similar to `fitNDR_1rate` (p. 271).

6.4 Ecology and Biogeography

This section treats new topics compared to the first edition. It witnesses of the vigorous developments that occurred in the last four years in this area.

6.4.1 Phylogenetic Diversity

Faith [73] was one of the first to propose the use of phylogenies to quantify biodiversity with the logic that three distantly related species represent more biological diversity than three, or even more, closely related ones. For a single tree, phylogenetic diversity is computed as the length of the branch lengths:

```
> sum(tree.primates$edge.length)
[1] 3.32
```

picante has several functions to perform this kind of computations. They require two mandatory arguments: a data frame giving information on the distribution of the species in local communities (or samples), and a phylogeny with these species. For instance, we build a simple data frame for our five primate species (as columns) with a single sample (as rows):

```
> d <- as.data.frame(t(rep(1, 5)))
> names(d) <- tree.primates$tip.label
> pd(d, tree.primates)
    PD SR
1 3.32  5
```

Phylogenetic diversity (PD) and species richness (SR) are computed for each community. Table 6.2 describes functions in picante that performs various

diversity calculations. These indices are closely related to the concept of phylogenetic covariances [132]. The phylogenetic relationships are established independently of the community structures (i.e., there is a single phylogeny for all species). Note that PSE = PSV with presence / absence (0 / 1) data. It is possible to compute the expected variances for PSV and PSR with the option `compute.var` (`TRUE` by default).

Table 6.2. Phylogenetic diversity functions in picante. In addition `psd` is a wrapper calling the last four functions

Function	Quantity(ies)	Description
`pd`	Phylogenetic diversity (PD)	Sum of branch lengths linking species in the community
	Species richness (SR)	Number of species in the community
`psv`	Phylogenetic species variability (PSV)	Mean of phylogenetic correlations among species in the community; varies between 0 (phylogenetic and community structures coincide) and 1 (no phylogenetic relationship among species in a community)
`psr`	Phylogenetic species richness (PSR)	PSR = PSV × SR
`pse`	Phylogenetic species evenness (PSE)	Similar to PSV but taking relative species abundances into account (PSE = PSV with presence / absence data)
`psc`	Phylogenetic species clustering (PSC)	Mean of the deepest phylogenetic correlations in the community; varies between 0 (species are connected with zero-branch length) and 1 (species are connected by a star tree to the root)

Nee and May [215] used Faith's idea to devise an algorithm that optimizes phylogenetic diversity in the face of extinction. This is implemented in `optimEH` (*optimize evolutionary history*) in ade4. Consider our five-species phylogeny and assume we want to optimize diversity while one species is going extinct. Which species should be selected as "candidate" for extinction?

```
> optimEH(tree.primates, 4) # we want to keep 4 species
$value
[1] 3.11

$selected.sp
          names
1 Homo OR Pongo
2        Macaca
3        Ateles
4        Galago
```

So we should "sacrifice" either men or orang utans. (Sadly, the choice of the latter seems to have been done already.) If one of these two species goes extinct, the phylogenetic diversity will decrease from 3.32 to 3.11. See also `randEH` to perform the same operation but after random extinction(s).

`ade4` has another function, `originality`, to calculate the evolutionary originality of each species according to seven methods described in Table 6.3:

```
> round(originality(tree.primates, method = 1:7), 4)
           VW      M   NWU*    NWW QEbased     ED eqsplit
Homo   0.1071 0.1071 0.1071 0.1421  0.1122 0.4883    0.43
Pongo  0.1071 0.1071 0.1071 0.1421  0.1122 0.4883    0.43
Macaca 0.1429 0.1429 0.1429 0.1827  0.1764 0.6283    0.65
Ateles 0.2143 0.2143 0.2143 0.3198  0.2234 0.7150    0.81
Galago 0.4286 0.4286 0.4286 0.2132  0.3757 1.0000    1.00
```

The last two methods, which are very close in definition and present results, are also available in `evol.distinct` in `picante`. See also `orisaved` in `ade4` to calculate the proportion of originality preserved after species extinction.

Table 6.3. The seven methods used to calculate species's originality in the function `originality`. Δ is the distance matrix among species

	Description	Ref.
`VW`	Inverse of number of nodes from the tip to the root	[305]
`M`	Id. but each node is weighted by the number of branches descending from it	[200]
`NWU*`	Id. than `VW` but each node is weighted by 1 if its SR is higher than the sister-node, 0 otherwise	[219]
`NWW`	Inverse of SR of each subclade where the species belongs to	[219]
`QEbased`	Maximize the quadratic entropy (i.e., calculate $\Delta^{-1}\mathbf{1}/\mathbf{1}^T\Delta^{-1}\mathbf{1}$) resulting in giving greater weights to the more distinctive species	[238]
`ED`	Sum of branch lengths (weighted by their lengths divided by their number of descendants) from the tip to the root	[147]
`eqsplit`	Id. than `ED` but weights are lengths divided by degree of the basal node	[263]

Rao's quadratic entropy [239, 261] takes the phylogenetic distance among species into account to compute diversity:

$$D = \sum_{i,j=1}^{n} d_{i,j} x_i x_j \ , \tag{6.18}$$

where i and j are pairs of species, $d_{i,j}$ is the distance between them, and the x's are abundances. So Rao's D will be high for communities made of many, abundant, distantly-related species.

raoD in picante takes as arguments a community matrix (communities as rows, species as columns) and an ultrametric "phylo" tree.[12] The second argument is optional (in which case equidistance among species is assumed); the tree is pruned if needed. For illustration, we add a second sample to the above data frame d with random values from a Poisson distribution:

```
> d[2, ] <- rpois(5, 10)
> rownames(x) <- paste("Comm", 1:2)
> d
       Homo Pongo Macaca Ateles Galago
Comm 1    1     1      1      1      1
Comm 2    8     9      8     19     10
> raoD(d, tree.primates)
$Dkk
   Comm 1    Comm 2
0.5640000 0.5598491

$Dkl
          Comm 1    Comm 2
Comm 1 0.5640000 0.5725185
Comm 2 0.5725185 0.5598491

$H
           Comm 1     Comm 2
Comm 1 0.00000000 0.01059396
Comm 2 0.01059396 0.00000000

$total
[1] 0.5618443

$alpha
[1] 0.5602009

$beta
[1] 0.001643419

$Fst
[1] 0.002925043
```

Dkk: Rao's D computed within each community (x_i and x_j are taken in the same row of the matrix);

Dkl: the same but between pairs of communities (x_i and x_j are taken in two rows of the matrix) so its diagonal repeats Dkk;

[12] The function divc in ade4 calculates Rao's D but the distances are square-rooted.

`H`: between community diversity excluding within community diversity (`Dkl` minus the mean of the `Dkk`'s);
`total`: Rao's D computed on the pooled communities;
`alpha`: mean local diversity (sum of the `Dkk`'s weighted by the relative abundance in each community);
`beta` = `total` − `alpha`;
`Fst` = `beta`/`total`.

Pavoine, Baguette and Bonsall [237] further developed the decomposition of trait diversity using Rao's quadratic entropy. They provide R code accompanying their paper.

6.4.2 Community Structure

The methodology of the analysis of phylogenetic community structure relies on two main aspects: the use of a metric to quantify the distances among species or communities, and the application of randomization procedures to test the discrepancy between observed patterns and predictions from a null model [303].

picante is the main package to analyze community structure in a phylogenetic framework. Most of the functions in this package need a matrix community × species and either a phylogeny or a distance matrix for these species. The matrix may give presence/absence or absolute or relative abundances. Some methods also analyze species traits.

Table 6.4 lists four functions computing distances among communities. The first two analyze a community data frame and a matrix of distances among species. They both have an option `abundance.weighted` which is `FALSE` by default and says that the data must be treated as presence/absence (must be switched to `TRUE` to analyze abundances). The two other functions take a community and a tree as arguments: they calculate complementary quantities. These four functions return an object of class `"dist"`.

Table 6.4. Phylogenetic community distances in picante

Function	Description
`comdist`	Mean pairwise distance between communities[a]
`comdistnt`	Mean nearest taxon distance; 0 if species compositions are the same
`phylosor`	Shared fraction of branch lengths between two communities
`unifrac`	Non-shared fraction of branch lengths between two communities

[a] Same than `Dkl` in `raoD` but here the distances are not divided by two and they can be non-ultrametric

Two functions in picante assess phylogenetic community structure by comparing co-occurences and phylogenetic distances: `comm.phylo.cor` and `phylostruct`. Their arguments are:

```
comm.phylo.cor(samp, phylo, metric = "cij",
     null.model = "sample.taxa.labels", runs = 999, ...)
phylostruct(samp, tree, env = NULL, metric = "psv",
     null.model = "frequency", runs = 100, it = 1000,
     alpha = 0.05, fam = "binomial")
```

The option `null.model` specifies how the data should be randomized to generate a null distribution of the test statistic; the values are listed below.

`"sample.taxa.labels"`: shuffle phylogeny tip labels (only the species present in the community data).
`"pool.taxa.labels"`: shuffle phylogeny tip labels (all taxa in the phylogeny).
`"frequency"`: randomize community data matrix abundances within species (maintains species occurence frequencies).
`"richness"`: randomize community data matrix abundances within samples (maintains sample species richness).
`"independentswap"`: randomize community data matrix maintaining species occurrence frequencies and site richness using independent swaps.
`"trialswap"`: randomize community data matrix maintaining species occurrence frequencies and site richness using trial swaps.

Only the last four randomization procedures are available for `phylostruct`.

`comm.phylo.cor` tests the null hypothesis of no correlation between the phylogenetic and community structures. `phylostruct` calculates a null distribution a the specified metric (PSV by default):

```
> comm.phylo.cor(d, tree.primates)
$obs.corr
[1] -0.07202592

$obs.corr.p
[1] 0.8432581

$obs.rank
[1] 247.5

$runs
[1] 999

$obs.rand.p
[1] 0.2475
....
> phylostruct(d, tree.primates)
```

```
$metric
[1] "psv"

$null.model
[1] "frequency"

$runs
[1] 100

$it
[1] NA

$mean.obs
[1] 0.705

$mean.null
[1] 0.705

$quantiles.null
 2.5% 97.5%
0.705 0.705
....
```

The output of these functions include all values calculated at each run.

spacodiR provides another set of tools to assess phylogenetic community structuring based on Rao's quadratic entropy. They use modified estimators of Rao's D inspired from variance component estimators where a bias correction is applied with respect to sample size:

$$\hat{D}_k = \frac{n_k}{n_k - 1} D_k \ ,$$

where n_k is the number of individuals in the kth community and D_k is computed with (6.18) for the N communities ($k = 1, \ldots, N$). Hardy and Senterre [124] used this estimator to characterize diversity into a local, D_S (site), and a global, D_T (total), component:

$$\hat{D}_S = \frac{1}{N} \hat{D}_k \ ,$$
$$\hat{D}_T = \frac{1}{N(N-1)} D_T \ ,$$

where D_T is computed with (6.18) using the pooled community data. This leads to the following index:

$$\frac{\hat{D}_T - \hat{D}_S}{\hat{D}_T} \ .$$

Two versions of this index are derived: I_{ST} calculated without taking phylogeny into account (similar to assuming a star-tree among species), and P_{ST} which takes phylogeny into account. Hardy and Senterre developed a third index, Π_{ST}, which is similar to P_{ST} but considering only presence / absence data. Hardy and Jost [123] proposed a fourth index, B_{ST}, which quantifies how much species in the same community are more closely related than those in different communities, so $B_{ST} \approx P_{ST} - I_{ST}$.

These four indices are computed by the function `spacodi.calc`. The main argument is a matrix with species as rows and communities as columns, which is the transpose of the format required by picante. Fortunately, spacodiR has several functions to convert community matrices among different packages including the program phylocom. The second argument is the phylogeny:

```
> spacodi.calc(as.spacodi(d), tree.primates)
Formatting assumed to be that used with picante
$Ist
[1] -0.1134949

$Pst
[1] -0.1118464

$Bst
[1] -0.0003915409

$PIst
[1] 0
....
```

Because here $N = 2$, $\hat{D}_T < \hat{D}_S$ leading to negative indices.

Overall, picante and spacodiR offer a wide range of tools for community phylogenetics. Currently, these approaches are mainly descriptive or correlative. Recently, Ives and Helmus proposed an approach based on generalized linear mixed-effects models aiming to test more complex hypotheses of community structure and trait evolution [148]. Merging the community approach with models of diversification and trait evolution will be an exciting area of future research leading to more mechanistic models of biodiversity.

6.4.3 Phylogenetic Biogeography

Biogeography is the science of the geographical distribution of living species. There is a long tradition of integrating biogeography with evolutionary biology [e.g., 25, 191, 209], and in the past few years, some attempts have been made to develop a phylogenetic approach to biogeographical inference [189]. Lamm and Redelings [175] gave an overview of these recent developments which are mostly based on the use of Markovian models for discrete character change to

model changes in geographic distributions over time. By coding the geographic ranges as discrete character(s), the methodology outlined on pages 139 and 247 can be utilized to infer ancestral ranges and the rates of biogeographical processes such as dispersal or regional extinction.

Thus it appears that some functions in various packages can be used for biogeographical analyses with no pain. As a first example, Ree and Smith [264] summarized a parametric approach named dispersal–extinction–cladogenesis (DEC). In their model, space is divided into discrete units labeled A, B, C, etc., and the geographic ranges of species are thus A if the species is present in unit A, AB if it is present in both A and B, and so on. These define the states for the Markov model. Two basic events are considered: dispersal with rate d leading to the colonization of a single new unit (e.g., A $\rightarrow$ AB), and extinction with rate e leading to a range contraction by a single unit (e.g., AB $\rightarrow$ A) or to complete extinction (e.g., A $\rightarrow \emptyset$). Only single-unit steps are permitted as transitions. For a two-unit area, the model would therefore be:

$$\begin{array}{c} \\ \emptyset \\ \mathrm{A} \\ \mathrm{B} \\ \mathrm{AB} \end{array} \begin{array}{c} \begin{array}{cccc} \emptyset & \mathrm{A} & \mathrm{B} & \mathrm{AB} \end{array} \\ \left[\begin{array}{cccc} . & 0 & 0 & 0 \\ e & . & 0 & d \\ e & 0 & . & d \\ 0 & e & e & . \end{array}\right] \end{array}.$$

As above, the zeros indicate transitions that cannot be done directly. This model can be built for an analysis with `ace` (p. 252) as follows:

```
> m <- matrix(0, 4, 4)
> m[c(2:3, 8, 12)] <- 1
> m[14:15] <- 2
> m
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    1    0    0    2
[3,]    1    0    0    2
[4,]    0    1    1    0
```

The trait is coded with a four-state vector or factor where, obviously, the first state ($\emptyset$) cannot be observed. The likelihood values of the ancestral states give a reconstruction of the ancestral ranges along the phylogeny.

This formulation can be generalized to more than two area units, for instance with three units [264, 265]:

$$\begin{array}{c} \\ \emptyset \\ \mathrm{A} \\ \mathrm{B} \\ \mathrm{C} \\ \mathrm{AB} \\ \mathrm{AC} \\ \mathrm{BC} \\ \mathrm{ABC} \end{array} \begin{array}{c} \begin{array}{cccccccc} \emptyset & \mathrm{A} & \mathrm{B} & \mathrm{C} & \mathrm{AB} & \mathrm{AC} & \mathrm{BC} & \mathrm{ABC} \end{array} \\ \left[\begin{array}{cccccccc} . & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ e & . & 0 & 0 & d & d & 0 & 0 \\ e & 0 & . & 0 & d & 0 & d & 0 \\ e & 0 & 0 & . & 0 & d & d & 0 \\ 0 & e & e & 0 & . & 0 & 0 & 2d \\ 0 & e & 0 & e & 0 & . & 0 & 2d \\ 0 & 0 & e & e & 0 & 0 & . & 2d \\ 0 & 0 & 0 & 0 & e & e & e & . \end{array}\right] \end{array}.$$

The transition AB → ABC happens at rate $2d$ because the colonization may occur from unit A or from unit B. Coding this model is problematic with `ace` because of this parameterization resulting in some constraints on transition rates. `make.mkn` from diversitree may be used here because it accepts linear constraints (a rate being proportional to another; `ace` accepts only equality constraints where a rate is equal to another). However, coding the above model directly in `make.mkn` will be somewhat tedious because of the many constraints to code (p. 273). An alternative is to reparameterize the model to simplify it thanks to the assumption of isotropic colonization (dispersal can reach any unit from any occupied one). Thus geographic range may be recoded with a four-state character giving the number of occupied units: 0 (∅), 1 (A, B, or C), 2 (AB, AC, or BC), and 3 (ABC). The transition matrix becomes:

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \left[\begin{array}{cccc} . & 0 & 0 & 0 \\ e & . & d & 0 \\ 0 & e & . & 2d \\ 0 & 0 & e & . \end{array}\right] \end{array}.$$

Manipulating the model in this form with `make.mkn` is now much easier.

On the other hand, Ree and Sanmartín [264] proposed to relax the assumption of isotropic dispersal by adding proportional constraints (e.g., dispersal between units B and C occurs at rate $0.1d$). In this situation, the above simplification cannot be done and a complete rate matrix must be built. More steps in relaxing the model could be done by assuming different rates of dispersal among geographic units. In those cases, an anlysis with either `ace` or `make.mkn` may be conducted. With these functions, different models can be compared using the usual likelihood tools (LRT or AIC).

Instead of fitting a Markov model by maximum likelihood, a parsimony approach can be used as proposed by Ronquist [270]. This is the dispersal–vicariance (DIVA) method. Each kind of event is associated with a cost: zero for speciation, either by vicariance (e.g., splitting of AB into A and B), or by "local" divergence within a single unit, one for dispersal or local extinction. Other kinds of events must be compounded from these four basic events. For

instance, if an ancestral species has range AB and its two daughter-species as well, this implies speciation by vicariance followed by two dispersal events, and thus a total cost of two. For the two-unit case, the cost matrix is:

$$\begin{array}{c} \\ \text{A} \\ \text{B} \\ \text{AB} \end{array} \begin{array}{c} \begin{array}{ccc} \text{A} & \text{B} & \text{AB} \end{array} \\ \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 2 \end{bmatrix} \end{array},$$

which can be used in phangorn and the function `parsimony` with its option `cost`. The ancestral reconstructions are done with `ancestral.pars`. The cost matrix is built with:

```
> m <- c(0, 1, 0, 1, 0, 0, 1, 1, 2)
> dim(m) <- c(3, 3)
> m
     [,1] [,2] [,3]
[1,]    0    1    1
[2,]    1    0    1
[3,]    0    0    2
```

Before calling a function in phangorn the data must be converted into `"phyDat"` class. Suppose the ranges are stored in a character vector `x`:

```
> x
 [1] "AB" "B"  "A"  "B"  "B"  "A"  "AB" "AB" "AB" "AB" "A"
[12] "B"  "B"  "B"  "AB" "A"  "B"  "AB" "AB" "A"
> y <- phyDat(matrix(x), "USER", levels = unique(x))
```

The parsimony score and the ancestral values are calculated with:

```
parsimony(tr, y, "sankoff", cost = m)
ancestral.pars(tr, y)
```

Lamm and Redelings [175] pointed out that because DIVA minimizes the number of inferred dispersal and extinction events, this method tends to reconstruct widespread ancestral ranges. Obviously, the cost matrix can be easily modified to relax the assumption of greater cost of dispersal and extinction compared to speciation. The function `MPR` (p. 248) may also be used if the problem is reparameterized with the number of geographical units occupied.

Goldberg, Lancaster and Ree [110] used the MuSSE model framework (p. 276) to develop a biogeographical model with three states representing the geographic range: restricted (A or B) or widespread (AB). Their model has seven parameters: the speciation rates (λ_A, λ_B, λ_{AB}), the local extinction rates (μ_A, μ_B), and the dispersal rates from region A or B (d_A, d_B). It is implemented in the function `make.geosse` in the package diversitree. Hypotheses can be tested with the function `constrain` in the same way detailed above for the BiSSE model.

6.4.4 Niche and Bioclimatic Evolution

The availability of large data sets of species distributions and environmental variables has revived the interest in the concept of ecological niche [308]. Recent developments have attempted to quantify the niche of a species with empirical and theoretical distributions that are closely related to probability density functions. This approach is called environmental niche modeling (ENM). Basically, the output of such models is made of a vector of intervals of an environmental variable, and the corresponding expected proportions of occurence of the species. See Franklin [89] for an overview of methods.

Given a set of species with their respective ENM and a phylogeny, it is of interest to reconstruct the ancestral states of the niche. Graham et al. [114] used ancestral estimation for continuous characters to infer the ancestral values of the minimum and the maximum of the ENMs calculated for a group of frog species. This approach has the drawback of not integrating all the information in the niche model in the evolutionary analysis. To remedy this, Evans et al. [72] analyzed the output of ENMs by treating each component of the niche model and analyzing it with an ancestral reconstruction method.

phyloclim is a specialized package with methods to analyze and model bioclimatic niche evolution. The function `anc.clim` implements Evans et al.'s method:

```
anc.clim(target, posterior=NULL, pno, n=100, method="GLS")
```

`target` is a phylogeny and `pno` (predicted niche occupancy) is an output from a program estimating the niche of the species in `target`. The option `posterior` is an alternative to `target` as a list of trees: the estimation is done on each tree in the list. `n` controls the sampling process of `pno` before calling `ace` (p. 248) on each quantile point of the data in `pno`. As a simple example, we generate random data for an imaginary environmental variable varying from 1 to 10 for the five primate species:

```
> f <- function(n) {x <- runif(n); x/sum(x)}
> X <- as.data.frame(cbind(1:10, replicate(5, f(10))))
> names(X) <- c("variable", tree.primates$tip.label)
> X[1:2, 1:3]
  variable        Homo      Pongo
1        1 0.006232414 0.02663926
2        2 0.107894926 0.12386842
```

The columns of the data frame must be ordered like in this example. We can now call `anc.clim`:

```
o <- anc.clim(tree.primates, pno = X)
```

The companion funcion of `anc.clim` is `plotAncClim` which displays the result of the reconstruction of the ancestral niches with the phylogeny (Fig. 6.16):

```
plotAncClim(o, xspace = c(-0.3, -0.3), col = rep("black", 5))
```

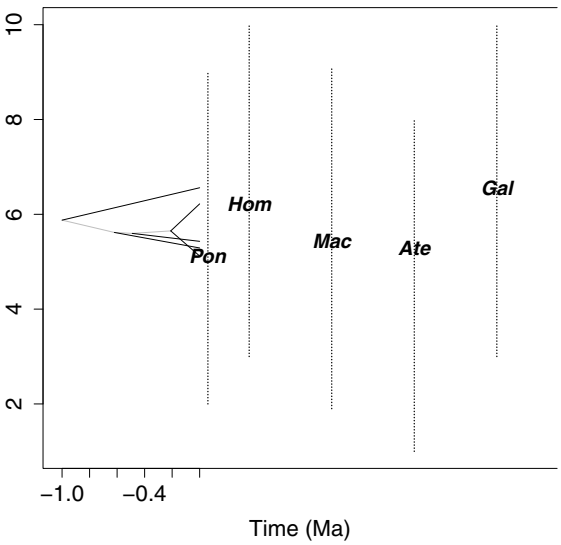

Fig. 6.16. Plot of ancestral niches

The function `niche.overlap` calculates two indices of niche overlap from a data frame similar to `X` or from data files on the disk storing similar information (e.g., output from a geographical information system).

```
> D <- niche.overlap(X)
> D
            Homo     Pongo    Macaca    Ateles    Galago
Homo          NA 0.7824326 0.7627297 0.6801493 0.6843619
Pongo  0.8122422        NA 0.7114907 0.6605182 0.5314554
Macaca 0.8325692 0.7982735        NA 0.7623236 0.6021067
Ateles 0.7742633 0.7451474 0.8582586        NA 0.5612213
Galago 0.8208744 0.6818138 0.7625051 0.7347804        NA
attr(,"class")
[1] "niolap"
```

The upper and lower triangles contain Schoener's similarity and a similarity based on the Hellinger distance, respectively. Schoener's similarity is derived from the Manhattan distance and is defined as [281]:

$$D(i,j) = 1 - \frac{1}{2}\sum |p_i - p_j| \ ,$$

so it can also be computed with `1 - dist(t(X[, -1]), "manhattan")/2`. The Hellinger distance is a measure of the dissimilarity between two probability distributions and is defined as:

$$H(i,j)^2 = \frac{1}{2}\sum \left(\sqrt{p_i} - \sqrt{p_j}\right)^2 \ .$$

This leads to the similarity index defined by Warren et al. [310] as:

$$I(i,j) = 1 - H(i,j)^2 \ .$$

The function `age.range.correlation` analyzes niche overlap quantifued with the above similarity indices: a linear regression is done with niche overlaps as response and divergence times as predictor. Significance is assessed by randomization controlled by the option `n = 1000` by default.

```
out.arc <- age.range.correlation(tree.primates, D)
```

The option `tri`, not used here, specifies which similarity matrix to use with two possible choices: `"upper"` (the default) or `"lower"`. The results are returned as a list whose elements can be plotted with (Fig. 6.17):

```
plot(out.arc$age.range.correlation)
apply(out.arc$MonteCarlo.replicates, 1, abline,
      lwd = 0.2, col = "grey")
abline(out.arc$linear.regression$coefficients)
```

6.4.5 Coevolutionary Phylogenetics

The study of cospeciation between phylogenies of parasites and their hosts has attracted a lot of attention. Currently, one method is available in `ape`: the ParaFit method by Legendre, Desdevises and Bazin [180]. It requires three matrices: the phylogenetic distances among the hosts (C), the phylogenetic distances among the parasites (B), and an association matrix (A) with hosts as rows and parasites as columns containing 1's for the observed pairs of host–parasite or 0's otherwise. A fourth matrix is computed as $D = CA^TB$. This double matrix product has for effect to multiply the pairwise evolutionary distances of the hosts and the parasites with the weights given by A. So the elements of D will be high if a pair of hosts and a pair of parasites have both diverged deeply and some of them are associated. On the other hand, they will be low if the pairs have diverged recently and / or are not associated. This leads to a test of global cospeciation among the hosts and the parasites defined as ParaFitGlobal $= \mathrm{tr}(D^TD) = \sum_{ij} d_{ij}^2$. The significance of this test is assessed by randomization of the rows of the matrix A. The null hypothesis is that the hosts and their parasites have speciated independently.

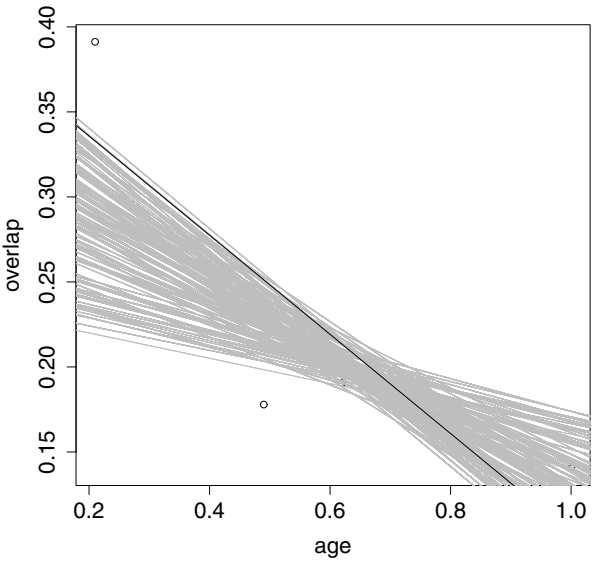

Fig. 6.17. Results of an analysis of niche overlap with respect to divergence time (age). The grey lines represent the randomized regressions

The ParaFit method is programmed in the function `parafit`: its first three arguments are mandatory and give the distances among hosts, the distances among parasites, and their association matrix. Other optional arguments control the number of replicates of the randomization tests and whether to perform individual host–parasite tests (see below). As an example, we simulate a pair of trees with perfect cospeciation, so the trees are identical (with different labels) and the association matrix is a diagonal matrix:

```
> H <- P <- rcoal(10)
> H$tip.label <- paste("host", 1:10)
> P$tip.label <- paste("parasite", 1:10)
> A <- diag(10)
> dimnames(A) <- list(H$tip.label, P$tip.label)
> A[1:4, 1:4]
       parasite 1 parasite 2 parasite 3 parasite 4
host 1          1          0          0          0
host 2          0          1          0          0
host 3          0          0          1          0
host 4          0          0          0          1
```

We thus expect a strongly significant test of global cospeciation:

```
> parafit(cophenetic(H), cophenetic(P), A)
```

```
n.hosts = 10 , n.parasites = 10
....
Test of host-parasite coevolution

Global test:  ParaFitGlobal = 1234.557
              p-value = 0.001 (999 permutations)
....
 Number of parasites per host
 host 1  host 2  host 3  host 4  host 5  host 6  host 7
      1       1       1       1       1       1       1
 host 8  host 9 host 10
      1       1       1

 Number of hosts per parasite
 parasite 1  parasite 2  parasite 3  parasite 4  parasite 5
          1           1           1           1           1
 parasite 6  parasite 7  parasite 8  parasite 9 parasite 10
          1           1           1           1           1
```

As an alternative example, we change the matrix `A` by randomly assigning links among hosts and parasites:

```
> A[] <- rbinom(100, size = 1, prob = 0.5)
> A[1:4, 1:4]
       parasite 1 parasite 2 parasite 3 parasite 4
host 1          1          1          1          1
host 2          1          0          0          1
host 3          0          1          1          1
host 4          1          0          1          0
> parafit(cophenetic(H), cophenetic(P), A)
n.hosts = 10 , n.parasites = 10
....
Test of host-parasite coevolution

Global test:  ParaFitGlobal = 184.2006
              p-value = 0.178 (999 permutations)
....
 Number of parasites per host
 host 1  host 2  host 3  host 4  host 5  host 6  host 7
      6       4       4       3       7       5       7
 host 8  host 9 host 10
      4       6       8

 Number of hosts per parasite
 parasite 1  parasite 2  parasite 3  parasite 4  parasite 5
          9           6           7           7           6
```

```
parasite 6  parasite 7  parasite 8  parasite 9 parasite 10
         5           3           5           4           2
```

So, logically, if the phylogenies of the hosts and the parasites are identical but their associations are random, there is no significant cospeciation.

`parafit` can also performs tests of individual host–parasite association with the option `test.links = TRUE`. These tests are based on the idea that each observed association contribute to the "ParaFitGlobal" test, so that deleting it decreases the value of the global test.

6.5 Perspectives

In the first edition of this book, some perspectives were drawn for the future developments of the methods for macroevolutionary analyses in R. Most of them have been accomplished during the last five years. For comparative methods, most published methods are available in several packages, and those not yet available can be more or less easily programmed. For diversification analyses, R provides all published methods and is now clearly the software *de choix* to address issues in diversification and macroevolution. The perspectives for the future are now of continuous progress and improvement. The tight integration with statistical, simulation, and graphical tools will lead to fast and reliable development for at least the next few years.

6.6 Case Studies

The case studies are here reduced to the *Sylvia* warblers for which we have estimated several trees in Chapter 5.

6.6.1 *Sylvia* Warblers

We begin by reading in the *Sylvia* data if necessary. We drop the outgroup species (*Chamaea fasciata*) for which we have no ecological data:

```
load("sylvia.RData")
nj.est <- read.tree("sylvia_nj_k80.tre")
nj.est <- drop.tip(nj.est, "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:[13]

```
DF <- sylvia.eco[nj.est$tip.label, ]
```

[13] 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.

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, nj.est, type = "d")
> syl.er
    Ancestral Character Estimation

Call: ace(x = DF$geo.range, phy = nj.est, type = "d")

    Log-likelihood: -22.16276

Rate index matrix:
         temp temptrop trop
temp        .        1    1
temptrop    1        .    1
trop        1        1    .

Parameter estimates:
 rate index estimate std-err
          1   5.4806  1.8448

Scaled likelihoods at the root
  (type '...$lik.anc' to get them for all nodes):
      temp   temptrop       trop
0.01809245 0.90972802 0.07217953
```

We now 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, nj.est, type="d", model="SYM")
> syl.sym
    Ancestral Character Estimation
```

```
Call: ace(x=DF$geo.range, phy=nj.est, type="d", model="SYM")

    Log-likelihood: -20.38674

Rate index matrix:
         temp temptrop trop
temp        .        1    2
temptrop    1        .    3
trop        2        3    .

Parameter estimates:
 rate index estimate std-err
          1   3.4911  1.9330
          2   0.0000     NaN
          3  10.1228  4.3518

Scaled likelihoods at the root
  (type '...$lik.anc' to get them for all nodes):
       temp    temptrop        trop
0.001717209 0.777724754 0.220558038
```

This model 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:

```
> anova(syl.er, syl.sym)
Likelihood Ratio Test Table
  Log lik. Df Df change Resid. Dev Pr(>|Chi|)
1  -22.163  1
2  -20.387  3         2      3.552     0.1693
```

This is not significant. We may want to try an intermediate "custom" model where only the transitions `temp` ↔ `temptrop` ↔ `trop` are permitted. We define a symmetric matrix `mod` which 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 transition `temp` ↔ `temptrop`, and the second one for the transition `temptrop` ↔ `trop`.

```
> syl.mod <- ace(DF$geo.range, nj.est, type="d", model=mod)
> syl.mod
    Ancestral Character Estimation

Call: ace(x=DF$geo.range, phy=nj.est, type="d", model=mod)

    Log-likelihood: -20.38674

Rate index matrix:
         temp temptrop trop
temp        .        1    0
temptrop    1        .    2
trop        0        2    .

Parameter estimates:
 rate index estimate std-err
          1   3.4911  2.0072
          2  10.1228  4.3878

Scaled likelihoods at the root
  (type '...$lik.anc' to get them for all nodes):
       temp    temptrop        trop
0.001717212 0.777724598 0.220558190
```

This model has no nestedness relationship with the two others (i.e., it cannot be derived from the symmetrical model with an equality constraint, and it cannot be reduced to the equal-rate model either). Therefore, we compare these three models with their AIC values:

```
> sapply(list(syl.er, syl.sym, syl.mod), AIC)
[1] 46.32552 46.77348 44.77348
```

The slighly smaller AIC value of the equal-rate model compared to the symmetrical one confirms the result of the likelihood ratio test. However, the custom model has the smallest AIC value among the three models and we select it for further analyses.

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. A good starting point is to take rate index matrix returned by `ace`, then setting its diagonal to zero:

```
> Q <- syl.mod$index.matrix
> Q
     [,1] [,2] [,3]
[1,]   NA    1    0
```

```
[2,]    1   NA    2
[3,]    0    2   NA
> diag(Q) <- 0
```

We can then extract the estimated rates and fill the appropriate elements of the matrix:

```
Q[1, 2] <- Q[2, 1] <- syl.mod$rates[1]
Q[2, 3] <- Q[3, 2] <- syl.mod$rates[2]
```

The above syntax is clear but there is a more general one that works in all situations:

```
Q[] <- c(0, syl.mod$rates)[Q + 1]
```

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,] -3.491143   3.491143   0.00000
[2,]  3.491143 -13.613993  10.12285
[3,]  0.000000  10.122850 -10.12285
```

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 $t = 0.05$:

```
> P <- matexpo(0.05 * Q)
> rownames(P) <- c("temp", "temptrop", "trop")
> colnames(P) <- rownames(P)
> P
              temp  temptrop      trop
temp     0.8509512 0.1201852 0.0288636
temptrop 0.1201852 0.5861571 0.2936578
trop     0.0288636 0.2936578 0.6774786
```

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"
```

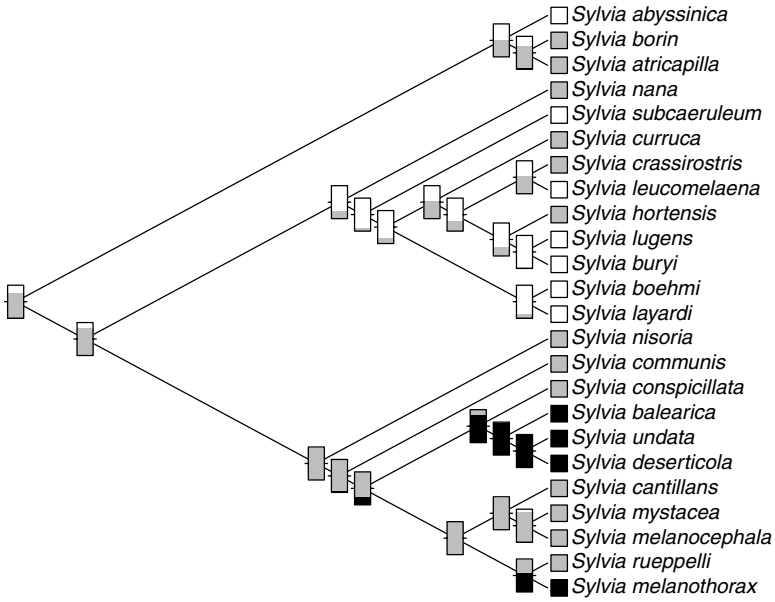

Fig. 6.18. Ancestral estimates of geographical range for 24 species of *Sylvia*. The thermometers on the nodes show the relative likelihoods of the three states: temperate (black), temperate-tropical (grey), tropical (white). The state of the recent species are shown on the tips of the tree

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. 6.18):

```
plot(nj.est, "c", FALSE, no.margin = TRUE, label.offset = 1)
tiplabels(pch = 22, bg = co, cex = 2, adj = 1)
nodelabels(thermo = syl.mod$lik.anc, cex = 0.8,
           piecol = c("black", "grey", "white"))
```

From this analysis we can infer that the ancestor of the genus *Sylvia* was, probably, a temperate-tropical bird. Because all temperate-tropical *Sylvia* are also migratory, this genus probably evolved from a temperate-tropical migratory species.

We continue with an analysis of diversification by first reading back the estimated chronogram in R:

```
sylvia.chrono <- read.tree("sylvia.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(sylvia.chrono)
Constant rate Yule model likelihood function:
$lambda
[1] 0.2474897

$se
[1] 0.05276498

$loglik
[1] -1.113822
```

The estimated speciation probability is quite high ($\hat{\lambda} = 0.25 \pm 0.05$). We now try to fit the simple birth–death model:

```
> birthdeath(sylvia.chrono)

Estimation of Speciation and Extinction Rates
            with Birth-Death Models

     Phylogenetic tree: sylvia.chrono
        Number of tips: 24
              Deviance: 2.068224
        Log-likelihood: -1.034112
   Parameter estimates:
      d / b = 0.2668499   StdErr = 0.4236002
      b - d = 0.2056182   StdErr = 0.08229464
   (b: speciation rate, d: extinction rate)
   Profile likelihood 95% confidence intervals:
      d / b: [0, 0.6152018]
      b - d: [0.1271991, 0.3129619]
```

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 extinctions [229]. The estimated parameters are $\hat{a} = 0.27$ and $\hat{r} = 0.21$. By back-substitution using $\lambda = r/(1 - a)$ and $\mu = \lambda a$, we obtain $\hat{\lambda} = 0.28$ and $\hat{\mu} = 0.07$. 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*(-1.034112 - -1.113822), 1)
[1] 0.6896911
```

This is not significant 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 [229].

We now explore the possible effect of geographic range on speciation rate using the Yule model with covariates. This analysis requires ancestral values of this trait which can be obtained from the previous analysis. However, we need to repeat it with the chronogram instead of the NJ tree (which would eventually show that our inference relative to the ancestral states of geographic range are overall robust to the phylogeny we used, though we do not detail this point here).

```
x <- sylvia.eco[sylvia.chrono$tip.label, "geo.range"]
ANC <- ace(x, sylvia.chrono, type = "d", model = mod)
```

The element `lik.anc` of the object `ANC` stores the relative likelihoods of the ancestral states:

```
> ANC$lik.anc[1:3, ]
           temp  temptrop      trop
[1,] 0.11797210 0.3921956 0.4898323
[2,] 0.00686829 0.6445059 0.3486258
[3,] 0.01302076 0.4493441 0.5376351
```

A simple way to derive ancestral states from this matrix is to find the maximum value for each row:

```
> anc <- apply(ANC$lik.anc, 1, which.max)
> anc
 [1] 3 2 3 3 3 2 2 1 1 1 2 2 2 1 3 3 3 3 2 3 3 3 3
```

Because `x` is ordered along the tips and `anc` is ordered along the nodes, we can concatenate them to create a factor which will be used as a predictor in `yule.cov`:

```
> X <- factor(c(x, anc))
> yule.cov(sylvia.chrono, ~ X)

---- Yule Model with Covariates ----

    Phylogenetic tree: sylvia.chrono
       Number of tips: 24
      Number of nodes: 23
             Deviance: -0.07197108
       Log-likelihood: 0.03598554

  Parameter estimates:
               Estimate    StdErr
(Intercept) -0.0535529 0.6887034
```

```
X2          -1.4608019 0.7619809
X3          -0.9775966 0.7469707

Null Deviance: 2.227645
  Test of the fitted model: chi^2 = 2.3   df = 2   P = 0.317
```

We recall that the predictors in the Yule model with covariates are interpreted in the same way than in a linear model; so a factor with k levels is modeled with $k-1$ coefficients. The predicted values of speciation rates for the three levels can be calculated with the inverse logit function (6.17):

```
> 1 / (1 + exp(-(-0.0535529)))
[1] 0.486615
> 1 / (1 + exp(-(-0.0535529 -1.4608019)))
[1] 0.1802943
> 1 / (1 + exp(-(-0.0535529 -0.9775966)))
[1] 0.2628613
```

This shows quite important differences but they appear as not significant from the likelihood ratio test comparing this model with the null Yule model ($\chi^2_2 = 2.3$, $P = 0.317$).

This analysis assumes that the ancestral values are given, which is not a reflection of the reality. Furthermore, we have inferred relative likelihoods for each node, so we can use them to sample the three states of the geographic range, and repeat the above analysis a large number of times. The goal of this analysis is to assess the impact of the choice of particular ancestral values on the results of the Yule model with covariates.

In order to perform a random sample of the three states, we use the function `sample` with its option `prob`. We thus build a function which will be applied to each row of `ANC$lik.anc` as we did above with `which.max`:

```
fsamp <- function(x) sample(length(x), size = 1, prob = x)
```

Note that we used `length(x)` instead of '3' so that this code can be used with any number of states. We can now repeat the model fitting procedure a large number of times. We here simply focus on the P-value of the likelihood ratio test:

```
nrep <- 1e3
Pvls <- numeric(nrep)
for (i in 1:nrep) {
    anc <- apply(ANC$lik.anc, 1, fsamp)
    X <- factor(c(x, anc))
    Pvls[i] <- yule.cov(sylvia.chrono, ~ X)$Pval
}
```

We can then compute a summary of the distribution of `Pvls` or plot a histogram of its relative frequencies together with a smoothed density curve (Fig. 6.19):

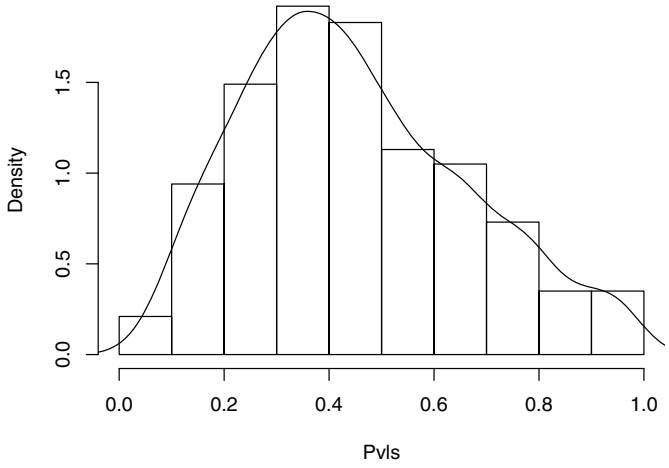

Fig. 6.19. Histogram and smoothed density curve of 1000 P-values from resampling of ancestral states before fitting a Yule model with covariates on the *Sylvia* data

```
hist(Pvls, freq = FALSE, main = "")
lines(density(Pvls))
```

It appears that some P-values are less than 0.05 but they occurred at a low frequency. So the results may be affected by the choice of ancestral values, but using the evidence from the likelihood values, we can conclude that the previous results were not affected by the choice of the most likely ancestral values.

6.7 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. Repeat the analyses on the primate data with Moran's I (Section 6.1.2) using weights computed from the variance-covariance matrix (see `?vcv`).

3. Simulate two variables with `rnorm` and two with `rTraitCont`. You will analyze these pairs of variables with phylogenetically independent contrasts (Section 6.1.1) and with generalized least squares (Section 6.1.5); you will do all regressions with and without intercept. Find when the regression coefficients are the same with both methods.

4. 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.

5. Simulate a standard Brownian motion process with $\mathrm{d}t = 0.1$ until $t = 100$ (hint: there should be 999 iterations in the simulation). Calculate the variance among replicates of this process and compare with the one simulated above with $\mathrm{d}t = 1$. Which parameter of the latter we should modify to obtain similar variances?

6. Implement Desdevises et al.'s [54] method in R (see p. 216).

7. Consider the phylogenies estimated for *Sylvia* (Section 5.8.1). Compute the phylogenetically independent contrasts for migration distance using the following branch lengths:
   - The neighbor-joining estimates (Fig. 5.17);
   - From the chronogram estimated by penalized likelihood (Fig. 5.21);
   - 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.

8. 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.

7

Simulating Phylogenies and Evolutionary Data

Data simulation has a critical place in evolutionary biology. Because evolutionary patterns and processes unroll over long time spans, their simulations has become an important ingredient of evolutionary research [e.g., 8]. From a statistical inference point of view, it is necessary to simulate data to assess the statistical properties of hypothesis tests and parameter estimators. This practice still needs to be generalized: it is quite common to read papers describing new data analysis methods without presenting some simulation results in order to asses the bias and variance of the estimators, or the type I and II error rates of the test. On the other hand, the recent rise in uses of simulation procedures for statistical inference (e.g., Markov chains Monte Carlo) has emphasized the need for simulating data under a wide range of situations.

7.1 Trees

7.1.1 Simple Trees

`ape` has several functions to generate random trees under different models. `rtree` generates a tree by recursive 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", and so on, are given to the tips. `br` specifies the function to generate random branch lengths: further arguments for this function are given in place of '`...`'. By default, a uniform distribution between 0 and 1 is used. Use `br = NULL` for a tree with no branch length. The algorithm used by `rtree` is:

1. Draw randomly an integer a on the interval $[1, n-1]$. Set $b = n - a$.
2. If $a > 1$, apply (recursively) step 1 after substituting n by a.

E. Paradis, *Analysis of Phylogenetics and Evolution with R*, Use R,
DOI 10.1007/978-1-4614-1743-9_7, © Springer Science+Business Media, LLC 2012

3. Repeat step 2 with b in place of a.
4. Assign randomly the n tip labels to the tips.

This algorithm is simple and easily implemented in R (p. 336). If step 4 is deleted, this gives an algorithm to simulate random unlabelled topologies. The above algorithm seems to have interesting properties in that the simulated trees are not biased in a particular direction, though no detailed study has assessed this.

7.1.2 Coalescent Trees

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

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

where the options are similar to `rtree`. The algorithm may be sketched as follows:

1. Generate a set of $n - 1$ random coalescent times.
2. Draw randomly two tips among the n. Set the branching time between them as the first coalescent time generated in step 1.
3. Aggregate these two tips into a single one, and delete the first coalescent time from the set.
4. Repeat steps 2 and 3 substituting n by $n - 1$ until all coalescent times have been assigned.

Note that the default for `br` is to generate branch lengths under a coalescent model with constant population parameter Θ. This parameter is traditionally given by $4N_e\nu$ where N_e is the effective population size and ν is the mutation rate [309] (the constant 4 may vary depending on the type of locus considered). The R code to generate a set of coalescent times among n individuals in a population assuming constant Θ is simple:

```
2 * rexp(n - 1)/(n:2 * (n - 1):1)
```

Instead of its default value, `br` may be a numeric vector computed beforehand. This gives the possibility to simulate coalescent trees with $\Theta(t)$ variable through time (though homogeneous across lineages) by rescaling the coalescent times t simulated under constant Θ with:

$$t' = \frac{1}{\Theta(0)} \int_0^t \Theta(u)\mathrm{d}u \; .$$

For instance, for a population following an exponential growth model with parameter ρ [171]:

$$t' = \frac{e^{\rho t} - 1}{\rho} \; .$$

The corresponding R code is:

```
x <- 2 * rexp(n - 1)/(n:2 * (n - 1):1)
tr <- rcoal(n, br = (exp(rho * x) - 1)/rho)
```

Both `rtree` and `rcoal` generate a single tree: they must be called repeatedly to generate a sample of trees. This is what `rmtree` does: it returns an object of class `"multiPhylo"`.

Coalescent trees can also be simulated with the function `ms` from phyclust. This package includes a port of the ms program developed by Hudson [141]. The `ms` function does simulation under a wide range of situations: single or multiple populations linked with migration rate(s), exponential population growth with rate(s) that can change through time as well as with respect to population size, and populations can be split and / or merged.

The details of all options can be displayed from R with `ms()`. The first argument `nsam` is the number of sequences n and must be specified. The second argument is the number of replications (1 by default). The third argument is a character string giving the options to pass to ms. The option `"-T"` is needed if one wants to output the tree. The output is an object of class `"ms"` which is a vector of mode character where the first element is the command call to ms, then come the Newick strings separated by `"//"`. These Newick strings can be converted into `"phylo"` objects with `read.tree` using the option `text`:

```
> library(phyclust)
> x <- ms(2, 2, opts = "-T")
> x
ms 2 2 -T
//
(1: 0.290178149939,2: 0.290178149939);
//
(1: 1.012070059776,2: 1.012070059776);
> x[c(3, 5)]
[1] "(1: 0.290178149939,2: 0.290178149939);"
[2] "(1: 1.012070059776,2: 1.012070059776);"
> read.tree(text = x[c(3, 5)])
2 phylogenetic trees
```

The output may also include the number of segregating sites and their positions:

```
> ms(2, opts = "-s 2 -T")
ms 2 1 -s 2 -T
//
(1: 0.041870053858,2: 0.041870053858);
segsites: 2
positions:    0.6410705887    0.7874408402
00
11
```

It is possible to generate only the segregating sites:

```
> ms(2, opts = "-s 1")
ms 2 1 -s 1
//
segsites: 1
positions:    0.3150193479
1
0
```

The option `"-F"` is used to output only the sites with a frequency of minor allele greater than or equal to a specified value (e.g., `"-F 0.1"`).

7.1.3 Speciation–Extinction Trees

Two functions in `ape` generate trees under a birth–death model of random speciation and extinction. Their options are:

```
rlineage(birth, death, Tmax = 50, BIRTH = NULL, DEATH = NULL,
         n0 = 2, eps = 1e-6)
rbdtree(birth, death, Tmax = 50, BIRTH = NULL, DEATH = NULL,
        n0 = 2, eps = 1e-6)
```

They simulate a continuous-time birth–death process. `rlineage` simulates a complete phylogeny including those that go extinct before `Tmax` while `rbdtree` does not consider them and returns an ultrametric tree. `birth` and `death` may be either single numeric values or functions specifying how speciation and extinction probabilities vary through time. `BIRTH` and `DEATH` are optional functions giving the primitive functions of `birth` and `death` used to speed computations. `n0` is the number of initial species: by default the tree is simulated from its root node; setting `n0 = 1` leads to simulate a tree with a root edge. Finally, `eps` gives the required numerical resolution of the branch lengths: if computing times with these functions are long, it may be judicious to increase this parameter (say `eps = 1e-3`). The function `drop.fossil` allows to remove the extinct species from a tree simulated with `rlineage`.

Here are four examples of trees simulated with `rlineage`:

```
t1 <- rlineage(0.1, 0, Tmax = 25)
t2 <- rlineage(0.1, 0.025)
b <- function(t) 1/(1 + exp(-0.03*t + 3))
t3 <- rlineage(b, 0.1)
t4 <- rlineage(b, 0.05)
```

The first tree is simulated with a Yule process since the extinction probability $\mu = 0$, the second tree is a simple birth–death tree, and the third and fourth trees are simulated with the same time-dependent speciation probability $\lambda(t)$ and two different values of μ. These trees can be plotted with (Fig. 7.1):

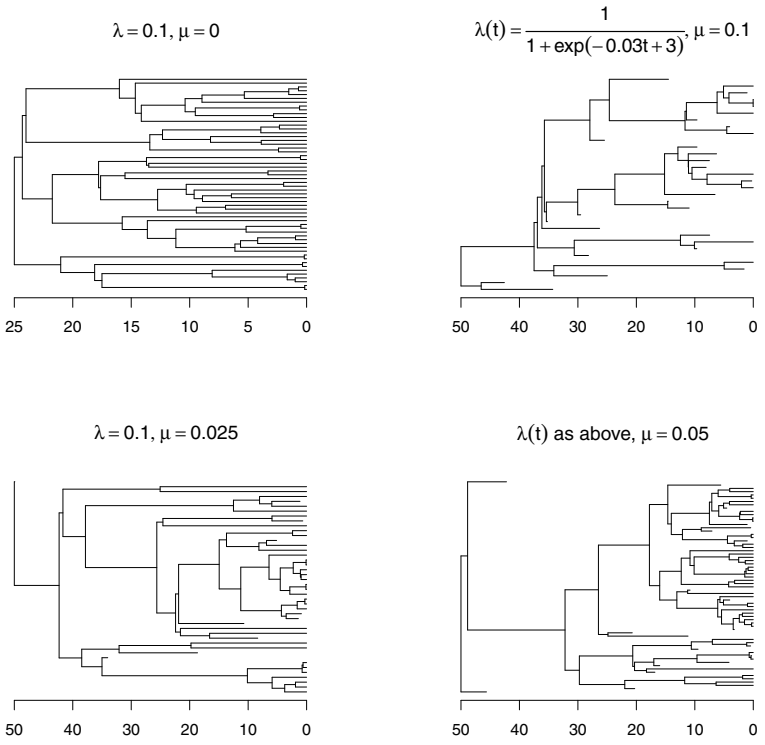

Fig. 7.1. Four random birth–death trees generated with `rlineage`

```
layout(matrix(1:4, 2, 2))
plot(t1, show.tip.label = FALSE); axisPhylo()
title(expression(list(lambda == 0.1, mu == 0)))
plot(t2, show.tip.label = FALSE); axisPhylo()
title(expression(list(lambda == 0.1, mu == 0.025)))
plot(t3, show.tip.label = FALSE); axisPhylo()
title(expression(list(lambda(t) ==
    frac(1, 1 + exp(-0.03*t + 3)), mu == 0.1)))
plot(t4, show.tip.label = FALSE); axisPhylo()
title(expression(lambda(t)*" as above, "* mu == 0.05))
```

The algorithms used by these two functions are detailed in the supplementary information of [232]. The general idea is based on simulating times to events which may be speciations and / or extinctions; each lineage being considered successively. In the case of `rlineage`, this can be sketched as follows (for simplicity, the procedure of recording nodes and branch lengths is ignored):

Generate a time-to-event on a lineage: is it smaller than `Tmax`?

1. Yes: is it a speciation or an extinction event?
   (a) Speciation: apply the whole procedure recursively to each daughter-lineage.
   (b) Extinction: record the tip and consider the next lineage.
2. No: record the tip and consider the next lineage.

Note that with this algorithm, it could happen that no lineage survives until `Tmax`. The algorithm for `rbdtree` is simpler because only the speciation events are considered:

Is there a speciation event on a lineage before `Tmax`?

1. Yes: apply the whole procedure recursively to each daughter-lineage.
2. No: record the tip and consider the next lineage.

Both algorithms generate trees with random n, and the topologies are not biased because all lineages are treated equally.

The functions `tree.bd` and `tree.yule` in diversitree implement the same algorithm than `rbdtree` but with constant parameters only; `tree.yule(0.1)` is a short-cut to `tree.bd(c(0.1, 0))`. The interface is:

```
tree.bd(pars, max.taxa=Inf, max.t=Inf, include.extinct=FALSE)
```

Either `max.taxa` or `max.t` must be specified to stop the simulation. diversitree has also functions to do tree–trait simulations which are detailed in Section 7.3. `tree.bd` assumes a single species at the start of the simulation (though the resulting root edge is not output).

The package geiger has the function `birthdeath.tree` which implements the continuous-time simulation algorithm with constant probabilities and two initial species that may go extinct before the end of the simulation. The options are similar to those in `tree.bd` with the defaults `time.stop = 0` and `taxa.stop = 0`.

A note of caution must be kept in mind with respect to the simulation of birth–death trees with a fixed number of tips. In a random birth–death process, the number of living species at any time is a random variable. `birthdeath.tree` uses an algorithm similar to `rbdtree`, but if `taxa.stop` is specified, the recursive procedure is stopped. This has the consequence of not treating all lineages equally, and so producing trees with unbalanced topologies. With `tree.bd`, this bias is avoided by selecting randomly the next lineage in the procedure. The trees thus generated are—apparently at least—topologically unbiased.

The package TreeSim implements different algorithms to simulate birth–death trees under constant parameters, but conditioned on a fixed number of species at the end of the simulation. The driving idea of these algorithms is still one of simulating time to events, but under various conditioning rules. The main functions in this package are `sim.bd.taxa`, `sim.bd.age`, `sim.gsa.taxa`, and `sim.rateshift.taxa`. Some methods to simulate incomplete sampling of the species are also implemented.

7.1.4 Tree Shapes

apTreeshape has the function `rtreeshape` which 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, `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 [9]. 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 four 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;
- Aldous's model (`model = "aldous"`) where a clade of size n splits in two clades of size i and $n-i$ with probability proportional to $1/(i(n-i))$ [9];
- 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 [166]. The value of p is given by the argument `p`.

If the option `FUN` is used, the splitting probabilities are specified through a function denoted as $Q_n(i)$ 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, say, `Q` which is an R function of the form `Q(n, i)`. For instance, for a completely unbalanced tree we would 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. 7.2. Note that Aldous's model can be coded with:

```
Q <- function(n, i) if (i %in% c(0, n)) 0 else 1/(i*(n - i))
```

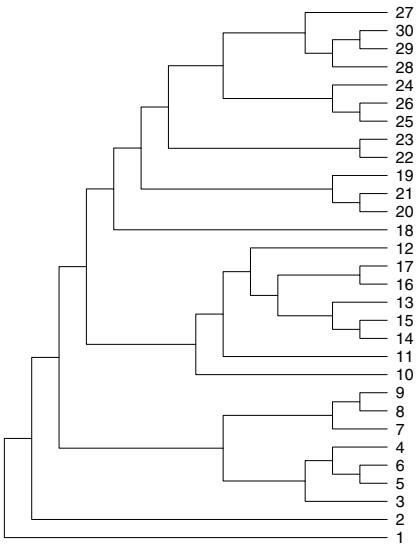

Fig. 7.2. A random tree generated with `rtreeshape` with splitting probabilities proportional to clade size

7.2 Phenotypic Data

Phenotypic traits are simulated in an evolutionary framework using phylogenies where traits evolve along their branches. This process creates covariance among the trait values at the tip of the phylogeny: species closely more related are more alike than those distantly related. Thus there are two approaches for trait simulation on a phylogeny:

- Compute the expected variance-covariance matrix of the trait values at the tips, and use standard methods to simulate multivariate data.
- Simulate the traits along the branches of the tree starting from the root to the tips. This allows to have the trait values at any point in time.

7.2.1 Covariance-Based Simulation

The first approach is particularly suited for continuous characters because correlated variables may be simulated with a linear combination of independent variables. Suppose we want to simulate two correlated variables from a multivariate normal distribution: we first simulate x_1 and x_2 independently from a standard normal distribution, then replace them by $ax_1 + bx_2$ and $cx_1 + dx_2$. This transformation is easily written in matrix form:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} .$$

Of course, if the correlation is zero we want to have $a = d = 1$ and $b = c = 0$. We see two methods to do this. The first one is based on the decomposition of the correlation matrix in its eigenvalues and eigenvectors. This is used by the function `mvrnorm` in the package MASS. The coefficients are computed with:

$$\Psi \times \begin{bmatrix} \sqrt{\lambda_1} & 0 \\ 0 & \sqrt{\lambda_2} \end{bmatrix} ,$$

where Ψ is the matrix of eigenvectors and λ are the eigenvalues. For illustration take a strong correlation of 0.9:

```
> R <- matrix(c(1, 0.9, 0.9, 1), 2)
> R
     [,1] [,2]
[1,]  1.0  0.9
[2,]  0.9  1.0
> e <- eigen(R, EISPACK = TRUE)
> e
$values
[1] 1.9 0.1

$vectors
           [,1]       [,2]
[1,] 0.7071068  0.7071068
[2,] 0.7071068 -0.7071068
```

So the matrix of coefficients is:

$$\begin{bmatrix} 0.707 & 0.707 \\ 0.707 & -0.707 \end{bmatrix} \times \begin{bmatrix} \sqrt{1.9} & 0 \\ 0 & \sqrt{0.1} \end{bmatrix} = \begin{bmatrix} 0.975 & 0.224 \\ 0.975 & -0.224 \end{bmatrix} ,$$

computed in R with:

```
> e$vectors %*% diag(sqrt(e$values))
           [,1]       [,2]
[1,] 0.9746794  0.2236068
[2,] 0.9746794 -0.2236068
```

These calculations may be done directly with `mvrnorm` which simulates from a multivariate normal distribution with known variance-covariance matrix. This matrix can be computed from a phylogeny with the function `vcv` in ape:

```
> tr <- rcoal(4)
> V <- vcv(tr)
> V
```

```
          t1       t3       t2       t4
t1 1.558640 1.302033 0.000000 0.000000
t3 1.302033 1.558640 0.000000 0.000000
t2 0.000000 0.000000 1.558640 1.389731
t4 0.000000 0.000000 1.389731 1.558640
```

We guess easily that the topology of the tree was ((t1,t3),(t2,t4)). `vcv` is a generic function with methods for objects of class `"phylo"` and of class `"corPhyl"` that describes phylogenetic correlations (see Section 6.1.5). This allows to simulate traits under different models of phenotypic evolution. Here we simulate three variables with expected mean equal to zero:

```
> library(MASS)
> mvrnorm(3, mu = rep(0, 4), Sigma = V)
            t1       t3         t2         t4
[1,] 0.2741027 1.038781 -1.5651816 -1.8108536
[2,] 1.5845245 2.066122 -0.5960545 -1.0833940
[3,] 1.4304574 1.667481  0.5220741  0.4967692
```

The pairs of tips identified above look indeed similar. Note that the variables are arranged as rows, but the matrix can be transposed.

The second method is based on a Choleski decomposition of the correlation matrix which gives directly the coefficients:

```
> chol(R)
     [,1]      [,2]
[1,]    1 0.9000000
[2,]    0 0.4358899
```

Here the coefficients are arranged columnwise, so the terms in the matrix product are swapped and the vector of variables is transposed:

$$\left[\begin{array}{cc} x_1 & x_2 \end{array}\right] \times \left[\begin{array}{cc} 1 & 0.900 \\ 0 & 0.436 \end{array}\right] .$$

This gives a different result than the first method even if x_1 and x_2 are the same. As mentioned above, we may check that the coefficients are correct if there is no correlation:

```
> chol(matrix(c(1, 0, 0, 1), 2))
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

This second method is easier to use if we want to combine the correlation among observations (due to the phylogeny) with correlations among the variables (the three variables generated above with `mvrnorm` are independent, in other words, they evolved independently along the tree):

```
> X <- matrix(rnorm(8), 4)
> t(X %*% chol(R)) %*% chol(V)
            t1          t3        t2          t4
[1,] 0.7418616 -0.30547800 -1.549183 -0.6135896
[2,] 0.8165200  0.04249274 -2.300228 -1.0549728
```

The correlation between the two variables (again as rows) is obvious. This code, originally posted to r-sig-phylo by Liam Revell,[1] works with any number of variables. Note that this procedure can be generalized to the first method as well:

```
> f <- eigen(V, EISPACK = TRUE)
> A <- f$vectors %*% diag(sqrt(f$values))
> B <- e$vectors %*% diag(sqrt(e$values))
> A %*% t(B %*% t(X))
           [,1]       [,2]
[1,] -1.9654785 -2.0440239
[2,] -0.8323502 -1.4442619
[3,]  0.3123180  0.3247053
[4,]  1.2426276  0.9332076
```

The matrix is now correctly arranged (observations as rows, variables as columns) and both forms of correlation are clear.

The simulation of correlated variables may be combined with linear or nonlinear relationships among variables. In the examples above no formal relationship among the variables was assumed, only correlation. A simple model of two coevolving variables may be $y_i = \beta x_i + \epsilon_i$ with $x_i \sim \mathcal{MN}(\mu, V)$ and $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$; x is a variable evolving according to Brownian motion with expected mean μ (the value at the root of the tree), β is the coefficient linking x and y, and ϵ are independent random variables. This might be simulated with (after setting the values of `mu`, `beta`, `sigma`, and `n` as the number of species):

```
x <- mvrnorm(1, mu = rep(mu, n), Sigma = V)
y <- beta * x + rnorm(n, 0, sigma)
```

This assumes that y "follows" the evolution of x, so if $\beta = 0$, the y's are not phylogenetically correlated. We may relax the assumption of independence of the ϵ_i with:

```
y <- beta * x + mvrnorm(1, mu = rep(0, n), Sigma = V)
```

Different variance-covariance matrices may be used to generate the x_i's and the ϵ_i's. This gives a lot of possibilities, including nonlinear, multivariate models.

[1] https://stat.ethz.ch/pipermail/r-sig-phylo/2009-May/000369.html

7.2.2 Time-Explicit Simulation

The second approach simulates the traits along the branches of the tree. The idea is simple: given a model of evolution we may predict the distribution of the trait after the time given by the branch lengths. It is only required to know the value at the root of the tree.

The second approach has two main advantages compared to the first one: though covariance-based simulations are more direct, their computating times are expected to increase with n^2 because of the manipulation of $n \times n$ matrices while they will be proportional to n for the second approach. The second advantage is that it is possible to output trait values at the node of the tree as well as at the tips.

`ape` has two functions for time-explicit trait simulation: `rTraitCont` and `rTraitDisc`. These functions must be seen as general trait simulators because they accept user-defined models allowing for a wide range of possibilities. The interface of `rTraitCont` is:

```
rTraitCont(phy, model = "BM", sigma = 0.1, alpha = 1,
           theta = 0, ancestor = FALSE, root.value = 0, ...)
```

The option `model` may be `"BM"`, `"OU"`, or a function created by the user that specifies how the trait evolves. This function is of the form `model(x, l)` where `x` is the ancestral (or initial) trait value and `l` is the branch lenght. For instance, the Brownian motion model with $\sigma^2 = 0.1$ could be coded with:

```
model <- function(x, l) x + rnorm(1, 0, sqrt(l * 0.1))
```

A speciational model where change occurs only after a speciation event may be coded with (`l` is not used inside the function, but it still needs to be included as argument):

```
model <- function(x, l) x + rnorm(1, 0, sqrt(0.1))
```

If `model = "BM"`, a Brownian motion (also called Wiener) process is simulated with parameter σ (`sigma`). If `model = "OU"`, an Ornstein–Uhlenbeck process with parameters σ, α, the strength of the constraint, and θ, the value of the optimum (`sigma`, `alpha`, and `theta`). In both models, the parameters may be different for each branch of the tree, making possible to simulate a wide range of scenarios with varying selective regimes, evolutionary optima, and / or levels of noise. The exact updating formula by Gillespie [107] are used ensuring correct simulation output in all situations.

The function simulating discrete traits is somewhat similar:

```
rTraitDisc(phy, model = "ER",
      k = if (is.matrix(model)) ncol(model) else 2,
      rate = 0.1, states = LETTERS[1:k], freq = rep(1/k, k),
      ancestor = FALSE, root.value = 1, ...)
```

here `model` is `"ER"`, `"SYM"`, `"ARD"` (see Section 6.2.2 for details), or a function coding the evolution of the discrete trait of the same form as described above.

Both `rTraitCont` and `rTraitDisc` simulate a single trait. A matrix of traits can be simulated with `replicate`, e.g., `p` continuous traits:

```
X <- replicate(p, rTraitCont(tr))
```

These `p` traits are independent among themselves. The same procedure used above can be used to create correlation among the traits (`R` is here a $p \times p$ correlation matrix):

```
X %*% chol(R)
```

Relationships among variables can also be simulated as sketched above for covariance-based simulations. Another possibility is to use the function `rTraitMult`:

```
rTraitMult(phy, model, p = 1, root.value = rep(0, p),
       ancestor = FALSE, asFactor = NULL,
       trait.labels = paste("x", 1:p, sep = ""), ...)
```

where some options are similar to those in the two previous functions but here `model` can only be a function, and `p` is the number of traits. The returned object is a data frame with `p` columns; the argument `asFactor` specifies those that should be returned as factors and thus considered as discrete variables. The fact that `model` is a function makes possible to simulate any kind of model. For instance, consider a model with two traits x and y following two correlated Brownian motion processes:

$$x_{t+\delta} \sim \mathcal{N}(x_t + \gamma y_t, \delta\sigma^2) \,,$$
$$y_{t+\delta} \sim \mathcal{N}(y_t + \gamma x_t, \delta\sigma^2) \,.$$

Fixing $\gamma = 0.5$ leads to the R function:

```
mod <- function(x, l) {
    out1 <- rnorm(1, x[1] + 0.5 * x[2], sqrt(l * 0.1))
    out2 <- rnorm(1, 0.5 * x[1] + x[2], sqrt(l * 0.1))
    c(out1, out2)
}
```

This is used in the same way than above but here we specify the number of traits:

```
> tr <- rcoal(20)
> x <- rTraitMult(tr, mod, 2)
> x[1:4, ]
            x1         x2
t20 -0.7623281 -0.6179466
t1  -0.8454842 -0.5600850
```

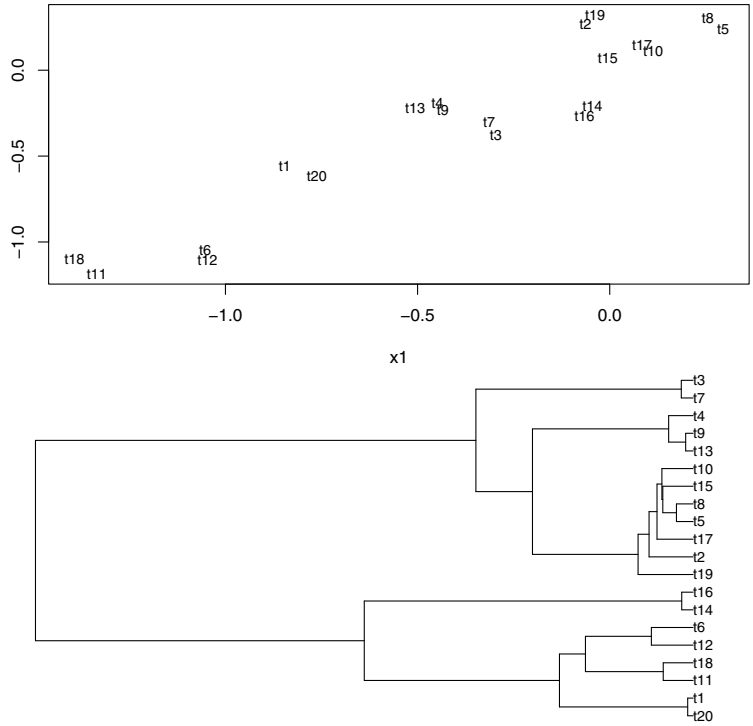

Fig. 7.3. Two correlated traits (top) and the phylogeny (bottom) used to simulate them

```
t11 -1.3345170 -1.1848345
t18 -1.3926053 -1.1020585
```

We may plot both traits together as well as the tree (Fig. 7.3):

```
layout(matrix(1:2, 2))
plot(x, type = "n")
text(x, labels = rownames(x), cex = 0.7)
plot(tr, font = 1, cex = 0.7)
```

Some extra arguments may be passed with the '...' if they are already defined in the model function, such as:

```
mod <- function(x, l, gamma) {
    out1 <- rnorm(1, x[1] + gamma * x[2], sqrt(l * 0.1))
    out2 <- rnorm(1, gamma * x[1] + x[2], sqrt(l * 0.1))
    c(out1, out2)
}
```

This allows to call the same model with different parameter values:

```
rTraitMult(tr, mod, 2, gamma = 0.5) # same than above
rTraitMult(tr, mod, 2, gamma = 0.2) # weaker correlation
```

7.3 Joint Tree–Trait Simulation

The package diversitree allows us to simulate the evolution of a clade where speciation and extinction probabilities are affected by a discrete character which evolves itself according to a Markov model. There are two such functions. `tree.bisse` simulates with a binary character:

```
tree.bisse(pars, max.taxa = Inf, max.t = Inf,
           include.extinct = FALSE, x0 = NA, ...)
```

with `pars` being a numeric vector with six values for the parameters λ_0, λ_1, μ_0, μ_1, q_{10}, and q_{01} where 0 and 1 are the two states of the trait, λ and μ are the speciation and extinction probabilities, and q_{ij} is the transition rate from state i to state j. The argument `x0` is the initial state at the root; by default, a random value is drawn. The three other options control the simulation: either `max.t` or `max.taxa` must be given a finite value, and `include.extinct` controls whether the extinct species must be pruned from the tree ('`...`' is actually unused).

`tree.musse` is similar to `tree.bisse` but with multiple state traits. The argument `pars` is a numeric vector with the required parameters; for instance, twelve parameters are needed for a three-state trait (the number of parameters is $k + k^2$ if k is the number of states).

These two functions use a continuous-time algorithm based on the principle of time-to-event outlined above: here the events are speciation, extinction, and transition among the states. The object returned is a tree of class `"phylo"` with additional elements that describe the evolution of the discrete trait. The function `history.from.sim.discrete` extracts this information and return an object of class `"history"` which is a list of vectors describing the state changes of the trait along each branch of the tree. There is a `plot` method allowing to display these changes with colors on the branches of the plotted tree. This may include the extinct species as well. Figure 7.4 shows an example with two states and equal speciation rates, but $\mu_1 = 2\mu_0$, and the transition rates are asymmetric:

```
p <- c(0.1, 0.1, 0.03, 0.06, 0.1, 0.5)
tr.bisse <- tree.bisse(p, max.taxa = 30, x0 = 0,
                       include.extinct = TRUE)
h <- history.from.sim.discrete(tr.bisse, 0:1)
tr <- prune(tr.bisse)
hp <- history.from.sim.discrete(tr, 0:1)
layout(matrix(1:2, 2))
par(mar = rep(0, 4))
```

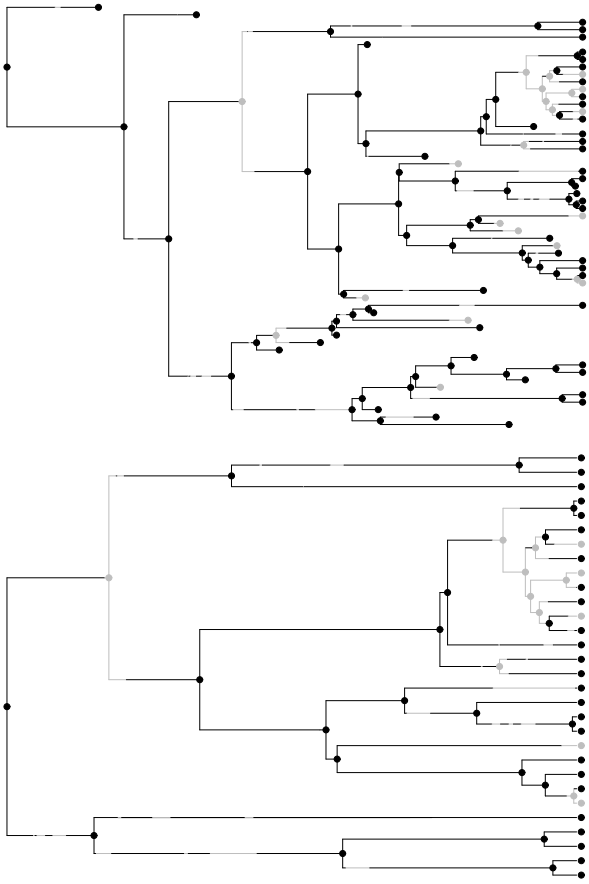

Fig. 7.4. Joint simulation of a binary trait and a phylogeny with the complete (top) and the pruned phylogeny (bottom). State 0 is in black, and state 1 in grey

```
plot(h, tr.bisse, c("black", "grey"), show.tip.label = FALSE)
plot(hp, tr, c("black", "grey"), show.tip.label = FALSE)
```

The tree with the extinct species removed is obtained with diversitree's special function `prune` which edits the history.

7.4 Molecular Sequences

Molecules are discrete characters, so molecular sequences could be simulated with some of the functions described in the previous section. However, there are models specific for this kind of data so that some specific functions have been written for simulating them. These models are described in Section 5.2.1.

simSeq in phangorn gives a general framework for simulating sequences. Its interface is:

```
simSeq(tree, l = 1000, Q = NULL, bf = NULL, rootseq = NULL,
       type = "DNA", model = "USER", levels = NULL, rate = 1,
       ancestral = FALSE)
```

This does not look so dissimilar to what we have seen above. If Q is not specified, a single-rate model is used, and if bf is not specified as well, equal frequencies of the states are assumed. Thus by default a JC69 model is used because type = "DNA" by default. The option model is used only if type = "AA": the four possible choices are "WAG", "JTT", "LG", and "Dayhoff".

The function stree may be useful to generate, for instance, star trees with the aim of simulating sequences on regular tree topologies. The following commands generate a star tree with all branch lengths equal to 10, or drawn randomly from a uniform distribution between 0 and 1, respectively:

```
compute.brlen(stree(5), 10)
compute.brlen(stree(5), runif(5))
```

simSeq returns an object of class "phyDat".

It is interesting to mention the function seqgen in phyclust which is a port to R of the program Seq-Gen [260]. The same options as in the stand-alone program can be used in phyclust. By default, the function returns the available options (similar to seq-gen -h from a shell):

```
> seqgen()
Sequence Generator - seq-gen
Version 1.3.2
(c) Copyright, 1996-2004 Andrew Rambaut and Nick Grassly
....
 Substitution model options:
  -m: MODEL = HKY, F84, GTR, JTT, WAG, PAM, BLOSUM, MTREV, GENERAL
      HKY, F84 & GTR are for nucleotides the rest are for amino acids
  -a: # = shape (alpha) for gamma rate heterogeneity [default = none].
  -g: # = number of gamma rate categories [default = continuous].
  -i: # = proportion of invariable sites [default = 0.0].
....
```

There are less substitution models of DNA evolution compared to simSeq, but it is possible here to specify some forms of intersite variation in substitution rates—though this can be simulated with simSeq by successive calls with different parameter values. The options of seqgen are:

```
seqgen(opts = NULL, rooted.tree = NULL, newick.tree = NULL,
       input = NULL)
```

where the first argument is the list of options given as a single character string (e.g., "-mHKY -l1000 -a1 -g4").

7.5 Exercises

1. Show that Aldous's branching model produces trees that tend to be balanced. Demonstrate this directly (i.e., without simulating trees) and by using tree simulations.

2. Simulate several discrete traits on a phylogenetic tree. Find some procedures to simulate a correlated evolution among these traits.

3. Simulate a sequence of nucleotides coding for a protein using as substitution rates 1, 0.8, and 2 for the three codon positions, respectively.

4. Simulate three continuous characters and three discrete ones along a phylogeny. Prepare a data frame with these six characters setting the rownames correctly.

5. Simulate some data in the same way than done to produce Fig. 7.3. Repeat the simulation outputting the ancestral values. Plot the two traits but displaying the nodes together with the tips. What analyses from Chapter 6 would you do on these data?

8

Developing and Implementing Phylogenetic Methods in R

> In our view it is somewhat ironic that even very substantial software contributions do not seem to attract the same academic credit as refereed publications: in reality nearly every user of software becomes a more meticulous and critical reviewer than most anonymous referees!
>
> —Venables & Ripley [306]

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* [254] and the *R Language Definition* [253].

8.1 Features of R

R is a language that is qualified as a dialect of S, a language for statistics [22]. 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

[1] http://cran.r-project.org/manuals.html.
[2] http://cran.r-project.org/faqs.html.

E. Paradis, *Analysis of Phylogenetics and Evolution with R*, Use R,
DOI 10.1007/978-1-4614-1743-9_8, © Springer Science+Business Media, LLC 2012

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`.

8.1.1 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 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>
```

[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.

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, use.edge.length = TRUE,
    use.tip.label = TRUE, index.return = FALSE, tolerance =
    .Machine$double.eps^0.5, scale = NULL, ...)
{
    same.node <- function(i, j) {
....
```

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).

Often, a data structure is derived or closely similar to another one. In this situation, it is useful to relate them so that the derived structure can use the functions that are already defined for other(s). Consider the function haplotype in pegas: it returns the unique sequences from an object of class "DNAbin" together with additional information. Because, both data structures are of the same nature, it seems natural to define a class "haplotype" derived from "DNAbin": this is done with *class inheritance*. In this case, the class is not made with one but two (or possibly more) character strings:

```
> h <- haplotype(woodmouse)
> class(h)
[1] "haplotype" "DNAbin"
```

We say that the class "haplotype" inherits the class "DNAbin". The proper way to test for class inheritance is with the function inherits (and not with the '==' operator):

```
> inherits(h, "DNAbin")
[1] TRUE
```

Now that the class of h is made of two strings, what happens when we call a generic function such as print? R first searches for a function print.haplotype; if it there is none, it then searches for print.DNAbin. Thus the new class benefits from the previously written functions. The available methods of a given class can be found with the function methods using the option class:

```
> methods(class = "DNAbin")
 [1] as.character.DNAbin as.list.DNAbin
 [3] as.matrix.DNAbin    cbind.DNAbin
 [5] c.DNAbin            [.DNAbin
 [7] labels.DNAbin       makeLabel.DNAbin
 [9] print.DNAbin        rbind.DNAbin
```

We only need to write new methods if we want the results to be different or if they do not exist:

```
> methods(class = "haplotype")
[1] plot.haplotype  print.haplotype
```

Non-generic functions may also be used, which is maybe even more important for us. For instance, we can compute genetic distances among the haplotypes with `dist.dna` because the internal coding is the same:

```
> dist.dna(h[1:3, ], "n")
     I II
II  16
III 13  5
```

A class may be made of more than two strings, like with the phylogenetic correlation structures met in Section 6.1.5:

```
> class(bm.prim)
[1] "corBrownian" "corPhyl"     "corStruct"
```

The search for methods is done sequentially. This allows us to write some functions common to all phylogenetic correlation structures we may build:

```
> methods(class = "corPhyl")
[1] Initialize.corPhyl vcv.corPhyl
```

8.1.2 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 these two very simple functions:

```
f <- function() print(x)
g <- function() {x <- 2; print(x)}
```

They are the same except that `g` has a variable named `x` in its body. Because no variable named `x` has been created within `f`, 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).

```
> x <- 1
> f()
[1] 1
> g()
[1] 2
```

If an object `x` is created within our function, the value of `x` in the global environment is not changed:

```
> x
[1] 1
```

Now `print(x)` uses the object `x` that is defined within its environment, that is, the environment of `g`.

The word *enclosing* above is important. In our two example functions, there are two environments: the global one and the one of the function `f` or `g`. 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.

An environment can be seen as a portion of the memory of the computer that contains some objects (data, functions, ...). When R is running, several distinct environments are open, hence the possibility to have different objects with the same name. Basically, the different environments are:

- one for each loaded package (hence distinct functions with the same name can be used in the same session);
- one for each function call (see example above);
- the global environment (usually named workspace);
- any that may be created by the user.

Environments in R are objects, and there are functions to create, manipulate, or delete them. Several basic functions have an `envir` argument to pass them an environment (e.g., `ls`, `get`, `assign`). One aspect that is worth considering with attention is that the enclosing environment of a function is the environment *where it was defined*—and not where it was called. This is important for recursive functions (that call themselves): each recursive call has the same enclosing environment, so that they can modify the same object with the super-assignment operator `<<-` which does assigment in the enclosing environment instead of the current one. To understand how this can be used, let us consider the structure below—it does not matter what is computed here:

```
FUN <- function(n) {
    if (....) return(NULL) # stops the recursion

    .... # code that computes 'y' and finds 'i'
    x[i] <<- y
```

```
        .... # code that computes 'm'
        FUN(m)
    }
```

The variables `i`, `y`, and `m` are located in the environment created by each call to `FUN`: if there are 20 iterative calls, there will be 20 environments each with these three variables (of course, with possibly different values). On the other hand, `x` is located in the enclosing environment of `FUN` which is the same for all recursive calls. There must be a way to stop the process to prevent infinite recursion which is what is done by the first line. Such a function is used with something like:

```
x <- vector(N)
FUN(N)
```

This makes possible to build efficient algorithms in R as used in the function `rtree` in `ape` (Section 7.1.1). The general structure of this function is:

```
rtree <- function(n) {
    FUN <- function(n) {
        ....
        edge[i, ] <<-
        ....
        FUN(m)
    }
    edge <- matrix(NA, n, 2)
    FUN(n)
    list(edge = edge, ....
}
```

Such a coding structure is very efficient because the object that is built (here `edge`) is created at the correct size once, the different calls to `FUN` have access to the very same object. Further, the position along the `edge` matrix is passed through the recursive calls so they are easily updated. We shall see other aspects of efficient computations in Section 8.5.

8.1.3 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. 8.1). 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.

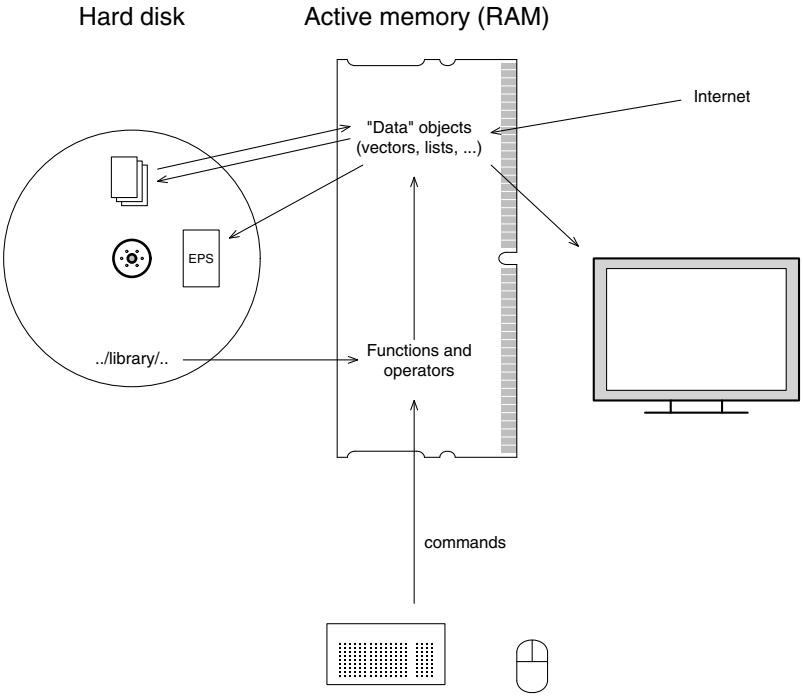

Fig. 8.1. A schematic view of how R works

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.

8.2 Writing Functions in R

Writing functions can be somehow extrapolated from what can be read 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)) # takes x in the workspace
[1] "logical"
> f(x = 1)       # takes x in the environment of f
[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 8.4.

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")
```

8.2.1 Programming Methods and Generics

Creating a new generic function is extremely simple. Below are two examples from `ape`:

```
vcv <- function(phy, ...) UseMethod("vcv")

as.phylo <- function (x, ...)
{
    if (class(x) == "phylo")
        return(x)
    UseMethod("as.phylo")
}
```

The arguments may be of any number and name(s). The methods are then named as outlined above, e.g., `vcv.phylo`, `as.phylo.hclust`, etc. It is required that a method contains at least the same arguments than its generic in the same order, including the '`...`'.

Most operators in R are generic and it is often useful to program methods especially because the extracting or subsetting operators (`$`, `[[`, `[`) drop extra attributes, including the class. We see an example from `ape` with the indexing operator for the class `"DNAbin"`:

```
"[.DNAbin" <- function(x, i, j, drop = FALSE)
{
    oc <- oldClass(x)
    class(x) <- NULL
    ....
    ans <- x[i, j, drop = drop]
    ....
    class(ans) <- oc
    ans
}
```

The first line saves the original class of the data which is restored before returning the result: it is done this way in order to work with the classes that inherit the present one such as `"haplotype"` (Section 8.1.1). The second line deletes the class to avoid infinite recursive calls of the present method. Note also the default of `drop` which is different than the standard '`[`' operator.

The replacement versions of these operators are also generic. Here are the first few lines of the code of `[[<-` for the class `"multiPhylo"`:

```
`[[<-.multiPhylo` <- function(x, ..., value)
{
    if (!inherits(value, "phylo"))
        stop('trying to assign an object not of class
       "phylo" into an object of class "multiPhylo".')
....
```

This is used with `x[[...]] <- value`, so it is not possible to assign an object not of class `"phylo"` into an object of class `"multiPhylo"`.

8.2.2 S3 *Versus* S4

The scheme described in the previous section actually refers to S3 classes. In the late 1990's, another scheme was defined: S4 classes (see Chambers's book [38] for a historical account). Why was another standard needed? There are two main reasons. First, when an object of a given class is created, R does not check whether its contents is correct and agrees with what some functions expect. For instance, nothing prevents us to create an object of class `"phylo"` which is not a tree:

```
> x <- 2
> class(x) <- "phylo"
> str(x)
Class 'phylo'  num 2
> x
Error in x$tip.label :
       $ operator is invalid for atomic vectors
```

Clearly, the function `print.phylo` does not know how to handle `x`.

The second problem is more subtle and is related to class inheritance in S3 which appears here as a two-edged sword. In the case of the object `bm.prim` on page 334, the class inherance and hierarchy seems obvious: a Brownian correlation is a special type of phylogenetic correlation which is itself a special statistical correlation structure. But we can create an object and not respect the class hierarchy:

```
> y <- bm.prim
> class(y) <- c("corBrownian", "corStruct")
```

The change may seem harmless (as above, it is permitted) but the consequence will be an error that is difficult to track down:

```
> gls(longevity ~ body, correlation = y)
Error in corMatrix.corBrownian(object) :
  object have not been initialized.
```

Trying to compute the variance-covariance matrix even suggests that an appropriate method is missing:

```
> vcv(y)
Error in UseMethod("vcv") :
  no applicable method for 'vcv' applied to an object
  of class "c('corBrownian', 'corStruct')"
```

Thus, one could be tempted to write a function `vcv.corBrownian` which may have dramatic effects because this would disrupt in an unpredictable way computations on objects of the correct class like `bm.prim`. The problem with the object `y` is called *inconsistent class inheritance* by Chambers [38].

Before considering some details about S4, we may ponder that the weaknesses of S3 classes are not disastrous: most of R relies on S3 and this is has not prevented it to become the *de facto* standard software for statistical analysis. The examples given here are rather extreme: very rarely you will need to set the class of an object. Besides, programming errors are facts of life: the example of inconsistent class inheritance observed by Chambers [38, p. 368] in the base function `Sys.time` has since been fixed.

By contrast to S3, an S4 class must be defined explicitly in R before we can create an object. This is done with the function `setClass`:

```
> setClass("phylo", representation(edge = "matrix",
+                                  tip.label = "character"))
[1] "phylo"
```

In this example, we have created an S4 class `"phylo"` which is similar to its S3 homonym without branch lengths. The new definition is stored in an object named `.__C__phylo` (hence hidden to `ls()`):

```
> .__C__phylo
Class "phylo" [in ".GlobalEnv"]

Slots:

Name:       edge tip.label
Class:   integer character
```

The mandatory elements of an S4 class are called *slots*. An S4 object is created with the function `new`. With our S4 `"phylo"` class, this could be:

```
> x <- new("phylo", edge = tree.primates$edge,
+          tip.label = tree.primates$tip.label)
> x
An object of class "phylo"
Slot "edge":
     [,1] [,2]
[1,]    6    7
[2,]    7    8
[3,]    8    9
[4,]    9    1
[5,]    9    2
[6,]    8    3
[7,]    7    4
[8,]    6    5

Slot "tip.label":
[1] "Homo"   "Pongo"  "Macaca" "Ateles" "Galago"
```

The slots are extracted with the `@` operator:

```
> x@tip.label
[1] "Homo"   "Pongo"  "Macaca" "Ateles" "Galago"
```

What happens if we try to create a "fake" object as we did above?

```
> new("phylo", edge = 2)
Error in validObject(.Object) :
  invalid class "phylo" object: invalid object for
  slot "edge" in class "phylo": got class "numeric",
  should be or extend class "matrix"
```

We now avoid this obvious mistake, but we can still create invalid objects:

```
> new("phylo", edge = matrix("a", 2, 2))
An object of class "phylo"
Slot "edge":
     [,1] [,2]
[1,] "a"  "a"
[2,] "a"  "a"

Slot "tip.label":
character(0)
```

`setClass` defines the slot `edge` as a matrix but it does not say that it should be integer. The function `setValidity` allows us to define what should be a valid S4 `"phylo"` object:

```
setValidity("phylo", f)
```

where `f` is a function (we will usually choose a more appropriate name) that tests the validity of the class `"phylo"`:

```
valid.phylo <- function(object) {
    if (!is.integer(object@edge)) {
        cat("slot 'edge' not integer\n")
        return(FALSE)
    }
    if (!is.character(object@tip.label)) {
        cat("slot 'tip.label' not character\n")
        return(FALSE)
    }
    TRUE
}
```

We then associate this validity checker with the class:

```
> setValidity("phylo", valid.phylo)
Class "phylo" [in ".GlobalEnv"]

Slots:

Name:       edge tip.label
Class:    matrix character
```

If we try again the above operation, this now returns an error:

```
> new("phylo", edge = matrix("a", 2, 2))
slot 'edge' not integer
Error in validObject(.Object) : invalid class
    "phylo" object: FALSE
```

Of course, we can still create an invalid object as long as it passes the tests in `valid.phylo`:

```
> new("phylo", edge = matrix(0L, 2, 2))
An object of class "phylo"
Slot "edge":
     [,1] [,2]
[1,]    0    0
[2,]    0    0

Slot "tip.label":
character(0)
```

Further tests can be added to avoid this and other problems.

If an S4 class inherits from other class(es), this must be defined in the call to `setClass` with the option `contains`: in this case all the slots of the inherited class are included in the new one. Generic functions and their methods are defined with the functions `setGeneric` and `setMethod`, respectively. All these functions and others relative to S4 classes are in the package methods which is part of the base installation of R.

Choosing between S3 and S4 for a given project is not trivial. "Old" S3 classes are flexible and easy to use and still widespread to prove their usefulness. For projects focused on computing with simple data structures, they are certainly sufficient. If complex data structures are necessary, and their integrity in especially important, S4 classes may be more appropriate. This will be at the expense of more programming work, and less efficient computation due to mandatory validity data checking. Usually, choosing between S3 or S4 will have little consequences for the end-users. Finally, some projects may do not even need to define new classes (e.g., porting an existing library of C/C++ code will only require to write the appropriate interface and possibly a driver). One strength of R—probably its most prevailing—is its flexibility. The more

sophisticated S4 classes should not be viewed as adding more constraints compared to S3, but rather as giving an alternative choice to developpers with its own benefits and costs.

8.3 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.

8.3.1 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 2.3.1). 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 8.4).

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.

8.3.2 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 [254].

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.

8.4 Writing R Packages

All the details of writing an R package are explained in a clear way in [254]. 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.

8.4.1 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.

8.4.2 The Documentation System

Every function written in R when distributed in a package must be documented. 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 the directory pkg/man/, it is possible to check the package with:

R CMD check pkg

This command will do many operations on the files within the 'pkg/' directory, including:

- Checking format of the *.Rd files (mismatching braces, ...)
- Checking consistency between the function definition (arguments) and their documentation, including the default values.
- A number of checks of the syntax in *.R files, foreign function calls, package dependies, etc.
- Executing all examples given in the *.Rd files.

The last item emphasizes the value of examples in R package documentation. These examples, if carefully selected, are very useful to most users, especially in packages directed to non-statisticians. Furthermore, they may be efficient tests of the reliability of a package, particularly if it depends on others because CRAN tests daily all packages deposited in its archive. This may allow package maintainers to quickly find bugs related to new features in other packages.

8.5 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 8.1 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.

Table 8.1. 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.
`.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`.

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)
    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(nb.edge),
   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 8.2).[4]

Two comments arise from this comparison. First, a program written in pure R can be very fast with small data sets: 0.004 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

[4] All computing times have been updated for this second edition with a laptop running Ubuntu 9.10 and R 2.12.1 with 8 Gb RAM and 8 Gb swap.

Table 8.2. Computing times (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	0.004	< 0.0001
1000	0.311	0.0001
2000	1.232	0.0002
5000	6.767	0.0004
10,000	29.014	0.0008

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 0.84 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 0.002 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 1.2 s to be completed. A vectorized version is possible with logical indexing:

```
sum(x[x < 0])
```

The computation time is now 0.033 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)
{
    ans <- .C("sum_neg", as.double(x), as.integer(length(x)),
              double(1), PACKAGE = "pkg")
    ans[[3]]
}
```

The time needed to complete `sumneg(x)` is 0.018 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.

This can be further improved: `.C` has two options to speed-up how data are passed to the C code. By default, data are duplicated before being sent to C, so that any change at the C level will not affect the corresponding R object. If we are sure that this will not occur, like in our present example, we may avoid data duplication with `DUP = FALSE`. This should be done carefully, however, and in case of doubt it is recommended to leave this option as its default. The second option is, by default, `NAOK = FALSE` so that the data are first checked for missing values (`NA`) before being sent to C. If we are sure about the absence of missing values, we may avoid this check. Thus a faster version of our R function would be (this does not require to recompile the C functions):

```
sumneg.fast <- function(x)
{
    ans <- .C("sum_neg", as.double(x), as.integer(length(x)),
              double(1), DUP=FALSE, NAOK=TRUE, PACKAGE="pkg")
    ans[[3]]
}
```

Its computing time is 0.005 s (0.008 s if only `DUP = FALSE` is used). These two options are used in several places in `ape` like in the above example.

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 complicated 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 [157]. This estimator can be computed for a tree, say `tr`, relatively easily:

```
(tr$Node - 1) / sum(tr$edge.length)
```

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`.

8.6 Computing with the class `"phylo"`

The class `"phylo"` was created in 2002 for coding phylogenetic trees in `ape`. A major modification of the internal coding happened in 2006 while keeping the general design of the class; this was released with `ape` 1.9 in November of that year. Over the years, this class has become an implicit standard for coding and manipulating phylogenies in R. Package developers have used it for its flexibility within R as well as the high- and low-level graphical functions in `ape` allowing them to annotate trees easily (Chapter 4), and the input / output functions using the standard Newick and NEXUS file formats (Chapter 3). Other features have been added in recent years that have increased the utility of using the class `"phylo"`. The aim of this section is to summarize them and indicate to the reader where to find detailed technical information.

8.6.1 The Main Design and its Variants

The design of the class `"phylo"` is borrowed from what computer scientists call a *graph*: a structure made of a set of vertices (nodes and tips) linked by edges (branches). This data coding structure is different from what is called a *tree* in computer science: a hierarchical structure, often binary and recursive. Interestingly, the latter structure was used in most computer programs to code phylogenetic trees [79]. The next section compares both structures and how to compute with them.

Figure 8.2 shows the basic structure of a `"phylo"` object. If a tree has no reticulation and its number of tips is n and its number of nodes is m, then its number of edges is $n + m - 1$. Such a structure is called an *acyclical graph*. In the `"phylo"` representation, each node is coded with an integer: from 1 to n for the tips, and from $n + 1$ to $n + m$ for the nodes. The matrix `edge` contains this topological information. This coding is called an *adjacency list*.

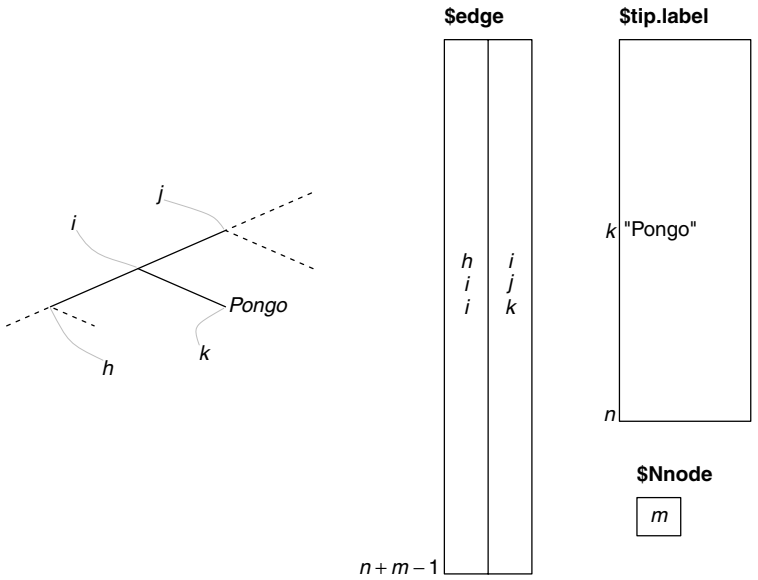

Fig. 8.2. An extract from a phylogeny (left) and its coding as a "phylo" object (right). The grey lines on the tree shows the node numbers that are assigned by the coding. n is the number of tips (also called leaves or terminal nodes) of the tree, and m is its number of (internal) nodes

This topological information can also be stored in an *adjacency matrix* which can be built from an object of class "phylo" (say tr) with:

```
M <- Ntip(tr) + Nnode(tr)
A <- matrix(0L, M, M)
A[tr$edge] <- 1L
A[tr$edge[, 2:1]] <- 1L
```

This adjacency matrix is square and symmetric and so can be decomposed with its eigenvalues and eigenvectors (see Section 6.1.3).

The basic structure of the class "phylo" is extended using two different mechanisms to match data. The first one is matching by indices where data are matched because they have the same position (index) in different vectors, lists, or matrices. This is widely used in the class "phylo", for instance, to store branch lengths using a vector edge.length of length $n + m - 1$, so that the length of the edge coded as the ith row in edge is given by edge.length[i].

The second mechanism is matching by labels and is used to match trees with external data such as vectors or data frames. Chambers [38, p. 180] points out that this mechanism is generally more meaningful that the previous one. There is indeed little meaning in saying that *Pongo* is the kth tip in our tree.

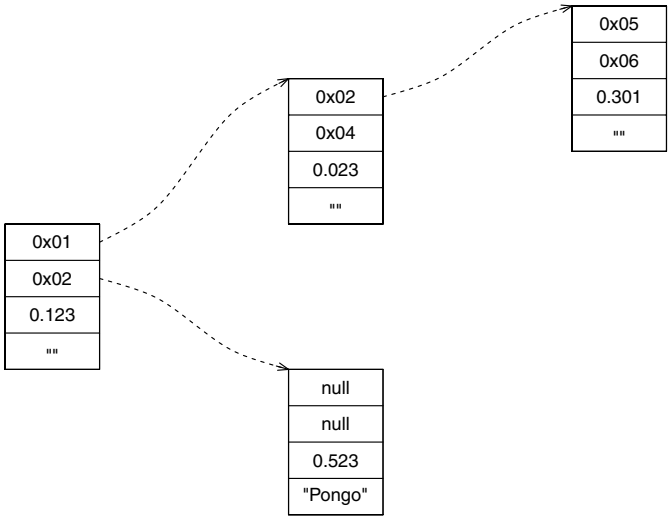

Fig. 8.3. The four nodes from Fig. 8.2 coded in a recursive tree-like structure. The four stacked rectangles represent four variables storing, from top to bottom, two memoty addresses (symbolized as 0x...), a branch length, and a label

On the other hand, matching with indices is much faster than with labels because the former allows computing code to access the data directly.

The general rule therefore is to match data with labels externally while using index matching internally. Typically, matching with labels is done at the R level using efficient base functions, and matching with indices can be done at the R or C level because it is an easy and fast operation. This rule appears advantageous in many situations, such as resampling for bootstrap or dealing with a large set of trees where the tips are numbered in the same way.

8.6.2 Iterative *Versus* Recursive Computing

The graph-like data structure used for coding `"phylo"` objects contrasts with the recursive structures used in the majority of phylogenetic computer programs. Figure 8.3 shows how the nodes displayed on Fig. 8.2 would be coded in a tree-like coding structure. Basically, each node (internal or terminal) is coded with four variables: the identities of its two node-sons, the length of the subtending edge, and a label. Typically, the identities will be memory addresses (stored in pointer variables in languages like C or C++).

Compared to the `"phylo"` structure, two weaknesses appear. The first one is the memory requirement. If we exclude the branch lengths and the labels (which have similar memory requirements in both structures), the tree-like structure requires $2 \times (n + m)$ pointers while an `edge` matrix requires

$2 \times (n + m - 1)$ integers. This is nearly the same on 32-bit machines because pointers and integers are stored on four bytes, but on 64-bit architectures, pointers are stored on eight bytes while integers are still stored on four bytes. Thus, on modern computers a tree-like data structure will use twice as much memory as a graph-like one.

The second weakness is linked to the fundamental recursive nature of tree-like data so that computing must be done recursively. On the other hand, the "phylo" structure allows the programmer either iterative or recursive computing. An example of the latter in R has been seen in Section 8.1.2, and similar recursive computing on the edge matrix can be done in C as well.[5] Iterative computing is usually faster than its recursive counterpart because it avoids successive function calls. To illustrate this, we take the classic example of calculating the factorial of an integer n: $n! = 1 \times 2 \times \ldots n$. The recursive version is a textbook example and very simple in R:

```
fact.recu <- function(n)
{
    if (n <= 1) return(1)
    n * fact.recu(n - 1)
}
```

The iterative version is even simpler but almost 80 times faster than the above with $n = 100$:[6]

```
fact.iter <- function(n) prod(1:n)
```

A typical use of iterative computation with the "phylo" class has been mentioned above with the calculation of the sum of branch lengths which is as efficient in R than in C. On the other hand, with a data structure as the one sketched on Fig. 8.3 the same operation can be done only with recursive computation.

With the "phylo" class, some operations or algorithms can be done using a combination of recursive and iterative computations. The generic function reorder acts on "phylo" objects by reordering the rows of the edge matrix with respect to two methods: "cladewise" or "pruningwise". The latter is interesting here because it will order the branches so that various algorithms can be applied iteratively either from the tips to the root of the tree (descending along the rows of the edge matrix), or from the root to the tips (ascending along the rows). This is used by several functions in ape such as pic which computes the phylogenetically independent contrasts from tips to root, or the functions simulating traits along a tree described in Section 7.2.2 from root to tips. phangorn uses also this approach to compute the likelihood of characters (Chapter 5). Valiente provides a complete theoretical and algorithmic treatment of these issues [302].

[5] See the file 'src/tree_phylo.c' in ape's sources.

[6] This function is about 1.8 times slower than R's factorial(n) which uses the Γ function and works also with real numbers.

The growing acceptance of the class `"phylo"` as a coding standard for phylogenetic trees in R has pointed to the need of improved documentation for developers. A Web site has been created in February 2008 with centralized information on ape.[7] It includes:

- A detailed description of the structure of the class `"phylo"` as well as some others used in ape;
- An application programming interface (API) describing the C code including in ape and how to call it from R;
- A repository of ape's source regularly updated (using the subversion versioning system);
- General information on ape, its history, citations, tips on its installation, etc.

The whole contents of the site is regularly updated particularly when a new version of ape is released.

[7] http://ape.mpl.ird.fr/

A

Short Course on Regular Expressions

Manipulating character strings flexibly appears particularly advantageous in R. A crucial aspect of this strength relies on the implementation of *regular expressions* ("regexp" in short). R can handle extended regexp as well as PERL-like ones. This short appendix focuses on the former.

A regexp in R is a character string which is interpreted in a special way by some functions (`sub`, `gsub`, `strsplit`, `grep`, `gregexpr`, and a few others). Some of the most basic regexp are:

`.`	any character
`[azerty]`	any one of the character within brackets
`[a-h]` or `[0-9]`	same than `[abcdefgh]` or `[0123456789]`
`[^a]`	any character but `a`
`a{6}`	same than `aaaaaa`
`a{n,}`	`a` is repeated n times or more
`a*`	same than `a{0,}`
`a+`	same than `a{1,}`
`a{n,m}`	`a` is repeated between n and m times
`^aze`	start of the string
`rty$`	end of the string

Note that `[A-Za-z]` means "an upper- or lowercase letter", whereas `[A-Z][a-z]` means "an uppercase letter followed by a lowercase one". An alternative to using explicitly the characters (which depends on the locale) is to use predefined classes. Some of them are given below (more can be found with `?regexp`):

`[:alpha:]`	upper- and lowercase letters
`[:digit:]`	digits
`[:lower:]`	lowercase letters
`[:space:]`	spaces, tabulations and new lines
`[:upper:]`	uppercase letters
`[:alnum:]`	`[:alpha:]` and `[:digit:]`

E. Paradis, *Analysis of Phylogenetics and Evolution with R*, Use R,
DOI 10.1007/978-1-4614-1743-9, © Springer Science+Business Media, LLC 2012

The characters {, }, [,], +, ^, and $ are part of the syntax of regexp and must be preceded by \\. However, this is context-dependent: the regexp "[[]" means "a left bracket" and is thus the same than "\\[".

We note that the backward slash is doubled: the reason is that R uses a C-like syntax to code strings where the character next to '\' is interpreted in a special way (e.g., \n is a new line, \t a tabulation, \" a quote—these are actually single characters as can be checked with nchar("\n")). Therefore, to code for a backward slash in a character string it must be doubled, but in a regexp it must be quadrupled:

```
> x <- "C:\\Program Files\\"
> cat(x, "\n")
C:\Program Files\
> gsub("\\\\", "/", x)
[1] "C:/Program Files/"
```

Regexp's may sometimes be tedious to find, if not always for the really useful ones. It is good to test them with gsub to make sure they work as expected; cat is also useful to see if the escaped characters are interpreted correctly like in the example above. An example is provided by the code of read.nexus where comments inserted in the Newick string are deleted with gsub:

```
> x <- "Pan:1.2[a comment],Homo:1.2[another comment]"
> gsub("\\[[^]]*\\]", "", x)
[1] "Pan:1.2,Homo:1.2"
```

The vertical bar means an alternative between two regexp:

```
> gsub("Pan|Homo", "XXX", x)
[1] "XXX:1.2[a comment],XXX:1.2[another comment]"
```

Most functions handling regexp in R have the option fixed = TRUE to inactivate the regexp matching mechanism and treat its main argument as a standard character string (or literate regexp):

```
> gsub("[a comment]", "", x)
[1] "P:1.2[],H:1.2[hr]"
> gsub("[a comment]", "", x, fixed = TRUE)
[1] "Pan:1.2,Homo:1.2[another comment]"
```

Parentheses have the meaning, similar to most languages, of grouping expressions, making possible to build complicated ones. For instance:

```
"((C|T)A){10,}G{5}"
```

means "the pair of characters, either CA or CT, repeated ten times or more, and followed by G five times".

References

[1] Abascal F., Posada D. & Zardoya R. 2007. MtArt: A new model of amino acid replacement for Arthropoda. *Molecular Biology and Evolution* **24**: 1–5.

[2] Abouheif E. 1999. A method for testing the assumption of phylogenetic independence in comparative data. *Evolutionary Ecology Research* **1**: 895–909.

[3] Adachi J. & Hasegawa M. 1996. Model of amino acid substitution in proteins encoded by mitochondrial DNA. *Journal of Molecular Evolution* **42**: 459–468.

[4] Adachi J., Waddell P. J., Marin W. & Hasegawa M. 2000. Plastid genome phylogeny and a model of amino acid substitution for proteins encoded by chloroplast DNA. *Journal of Molecular Evolution* **50**: 348–358.

[5] 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.

[6] Akaike H. 1973. Information theory and an extension of the maximum likelihood principle. In: *Proceedings of the second international symposium on information theory*, Petrov B. N. & Csaki F., editors, pages 267–281. Akadémia Kiado, Budapest.

[7] Albert J. 2007. *Bayesian computation with R.* Springer, New York.

[8] Aldous D. 1995. Darwin's log: a toy model of speciation and extinction. *Journal of Applied Probability* **32**: 279–295.

[9] Aldous D. 1996. Probability distributions on cladograms. In: *Random discrete structures*, Aldous D. & Pemantle R., editors, pages 1–18. Springer, New York.

[10] Aldous D. J. 2001. Stochastic models and descriptive statistics for phylogenetic trees, from Yule to today. *Statistical Science* **16**: 23–34.

[11] Alfaro M. E., Santini F., Brock C., Alamillo H., Dornburg A., Rabosky D. L., Carnevale G. & Harmon L. J. 2009. Nine exceptional radiations

plus high turnover explain species diversity in jawed vertebrates. *Proceedings of the National Academy of Sciences USA* **106**: 13410–13414.

[12] Baldauf S. L. 2003. Phylogeny for the faint of heart: a tutorial. *Trends in Genetics* **19**: 345–351.

[13] 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.

[14] Bandelt H. J., Chepoi V. & Eppstein D. 2010. Combinatorics and geometry of finite and infinite squaregraphs. *SIAM Journal On Discrete Mathematics* **24**: 1399–1440.

[15] Bandelt H. J., Forster P. & Röhl A. 1999. Median-joining networks for inferring intraspecific phylogenies. *Molecular Biology and Evolution* **16**: 37–48.

[16] Bandelt H. J., Forster P., Sykes B. C. & Richards M. B. 1995. Mitochondrial portraits of human populations using median networks. *Genetics* **141**: 743–753.

[17] Bandelt H. J., Macaulay V. & Richards M. 2000. Median networks: speedy construction and greedy reduction, one simulation, and two case studies from human mtDNA. *Molecular Phylogenetics and Evolution* **16**: 8–28.

[18] 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.

[19] Barraclough T. G., Harvey P. H. & Nee S. 1995. Sexual selection and taxonomic diversity in passerine birds. *Proceedings of the Royal Society of London. Series B. Biological Sciences* **259**: 211–215.

[20] Barraclough T. G., Harvey P. H. & Nee S. 1996. Rate of *rbc*L gene sequence evolution and species diversification in flowering plants (angiosperms). *Proceedings of the Royal Society of London. Series B. Biological Sciences* **263**: 589–591.

[21] Barry D. & Hartigan J. A. 1987. Asynchronous distance between homologous DNA sequences. *Biometrics* **43**: 261–276.

[22] Becker R. A., Chambers J. M. & Wilks A. R. 1988. *The New S Language*. Chapman & Hall, London.

[23] Billera L. J., Holmes S. P. & Vogtmann K. 2001. Geometry of the space of phylogenetic trees. *Advances in Applied Mathematics* **27**: 733–767.

[24] Blomberg S. P., Garland Jr. T. & Ives A. R. 2003. Testing for phylogenetic signal in comparative data: behavioral traits are more labile. *Evolution* **57**: 717–745.

[25] Blondel J. 1986. *Biogéographie évolutive*. Masson, Paris.

[26] Blondel J., Catzeflis F. & Perret P. 1996. Molecular phylogeny and the historical biogeography of the warblers of the genus *Sylvia* (Aves). *Journal of Evolutionary Biology* **9**: 871–891.

[27] 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.

[28] Bokma F. 2003. Testing for equal rates of cladogenesis in diverse taxa. *Evolution* **57**: 2469–2474.

[29] Bortolussi N., Durand E., Blum M. & François O. 2006. apTreeshape: statistical analysis of phylogenetic tree shape. *Bioinformatics* **22**: 363–364.

[30] Bouchenak-Khelladi Y., Verboom G. A., Hodkinson T. R., Salamin N., Francois O., Chonghaile G. N. & Savolainen V. 2009. The origins and diversification of C_4 grasses and savanna-adapted ungulates. *Global Change Biology* **15**: 2397–2417.

[31] Boussau B. & Daubin V. 2010. Genomes as documents of evolutionary history. *Trends in Ecology & Evolution* **25**: 224–232.

[32] Britton T., Oxelman B., Vinnersten A. & Bremer K. 2002. Phylogenetic dating with confidence intervals using mean path lengths. *Molecular Phylogenetics and Evolution* **24**: 58–65.

[33] Bruna E. M. 2010. Scientific journals can advance tropical biology and conservation by requiring data archiving. *Biotropica* **42**: 399–401.

[34] Buckland S. T., Burnham K. P. & Augustin N. H. 1997. Model selection: an integral part of inference. *Biometrics* **53**: 603–618.

[35] 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.

[36] Campbell V., Legendre P. & Lapointe F.-J. 2009. Assessing congruence among ultrametric distance matrices. *Journal of Classification* **26**: 103–117.

[37] Campbell V., Legendre P. & Lapointe F.-J. 2011. The performance of the Congruence Among Distance Matrices (CADM) test in phylogenetic analysis. *BMC Evolutionary Biology* **11**: 64.

[38] Chambers J. M. 2008. *Software for Data Analysis: Programming with R*. Springer, New York.

[39] Charif D. & Lobry J. R. 2007. SeqinR 1.0-2: a contributed package to the R project for statistical computing devoted to biological sequences retrieval and analysis. In: *Structural approaches to sequence evolution: Molecules, networks, populations*, Bastolla U., Porto M., Roman H. E. & Vendruscolo M., editors, pages 207–232. Springer Verlag, New York.

[40] 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.

[41] Chessel D., Dufour A.-B. & Thioulouse J. 2004. The ade4 package. I. One-table methods. *R News* **4**: 5–10.

[42] 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.

[43] Chor B. & Tuller T. 2005. Maximum likelihood of evolutionary trees: hardness and approximation. *Bioinformatics* **21**: i97–i106.

[44] Claeskens G. & Hjort N. L. 2008. *Model Selection and Model Averaging*. Cambridge University Press, Cambridge.

[45] Cook D. & Swayne D. F. 2007. *Interactive and Dynamic Graphics for Data Analysis with GGobi*. Springer, New York.

[46] Cornillon P. A., Pontier D. & Rochet M. J. 2000. Autoregressive models for estimating phylogenetic and environmental effects: accounting for within-species variations. *Journal of Theoretical Biology* **202**: 247–256.

[47] Cox D. R. & Oakes D. 1984. *Analysis of Survival Data*. Chapman & Hall, London.

[48] Crosbie S. F. & Manly B. F. J. 1985. Parsimonious modelling of capture-mark-recapture studies. *Biometrics* **41**: 385–398.

[49] Darwin C. 1859. *On the Origin of Species by Means of Natural Selection*. John Murray, London.

[50] Dayhoff M. O., Schwartz R. M. & Orcutt B. C. 1978. A model of evolutionary change in proteins. In: *Atlas of Protein Sequence and Structure, Vol. 5, suppl. 3*, Dayhoff M. O., editor, pages 345–352. National Biomedical Research Foundation, Washington, DC.

[51] de Vienne D. M., Aguileta G. & Ollier S. In press. The Euclidean nature of phylogenetic distance matrices. *Systematic Biology*.

[52] Degnan J. H. & Salter L. A. 2005. Gene tree distributions under the coalescent process. *Evolution* **59**: 24–37.

[53] 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.

[54] Desdevises Y., Legendre P., Azouzi L. & Morand S. 2003. Quantifying phylogenetically structured environmental variation. *Evolution* **57**: 2647–2652.

[55] Desper R. & Gascuel O. 2002. Fast and accurate phylogeny reconstruction algorithms based on the minimum-evolution principle. *Journal of Computational Biology* **9**: 687–705.

[56] Diaconis P. W. & Holmes S. P. 1998. Matchings and phylogenetic trees. *Proceedings of the National Academy of Sciences USA* **95**: 14600–14602.

[57] 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.

[58] Duponchelle F., Paradis E., Ribbink A. J. & Turner G. F. 2008. Parallel life history evolution in mouthbrooding cichlids from the African Great Lakes. *Proceedings of the National Academy of Sciences USA* **105**: 15475–15480.

[59] Eastman J. M., Paine C. E. T. & Hardy O. J. In press. spacodiR: Structuring of phylogenetic diversity in ecological communities. *Bioinformatics*.

[60] Edgar R. C. 2004. MUSCLE: Multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research* **32**: 1792–1797.
[61] Edwards A. W. F. 1992. *Likelihood (expanded edition).* Johns Hopkins University Press, Baltimore.
[62] 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.
[63] Edwards A. W. F. 2009. Statistical methods for evolutionary trees. *Genetics* **183**: 5–12.
[64] Edwards S. V. 2009. Is a new and general theory of molecular systematics emerging? *Evolution* **63**: 1–19.
[65] Edwards S. V., Liu L. & Pearl D. K. 2007. High-resolution species trees without concatenation. *Proceedings of the National Academy of Sciences USA* **104**: 5936–5941.
[66] Efron B. 1981. Nonparametric estimates of standard error: the jacknife, the bootstrap and other methods. *Biometrika* **68**: 589–599.
[67] Efron B. 1998. R. A. Fisher in the 21st century (with discussion). *Statistical Science* **13**: 95–114.
[68] Efron B., Halloran E. & Holmes S. 1996. Bootstrap confidence levels for phylogenetic trees. *Proceedings of the National Academy of Sciences USA* **93**: 13429–13434.
[69] Efron B. & Tibshirani R. 1991. Statistical analysis in the computer age. *Science* **253**: 390–395.
[70] Emerson B., Paradis E. & Thébaud C. 2001. Revealing the demographic histories of species using DNA sequences. *Trends in Ecology & Evolution* **16**: 707–716.
[71] Ernest S. K. M. 2003. Life history characteristics of placental nonvolant mammals. *Ecology* **84**: 3402. `http://www.esapubs.org/archive/ecol/E084/093/default.htm`.
[72] Evans M. E. K., Smith S. A., Flynn R. S. & Donoghue M. J. 2009. Climate, niche evolution, and diversification of the "bird-cage" evening primroses (*Oenothera*, sections *Anogra* and *Kleinia*). *American Naturalist* **173**: 225–240.
[73] Faith D. P. 1992. Conservation evaluation and phylogenetic diversity. *Biological Conservation* **61**: 1–10.
[74] Felsenstein J. 1981. Evolutionary trees from DNA sequences: a maximum likelihood approach. *Journal of Molecular Evolution* **17**: 368–376.
[75] Felsenstein J. 1985. Confidence limits on phylogenies: an approach using the bootstrap. *Evolution* **39**: 783–791.
[76] Felsenstein J. 1985. Phylogenies and the comparative method. *American Naturalist* **125**: 1–15.
[77] Felsenstein J. 1988. Phylogenies and quantitative characters. *Annual Review of Ecology and Systematics* **19**: 445–471.
[78] Felsenstein J. 1988. Phylogenies from molecular sequences: inference and reliability. *Annual Review of Genetics* **22**: 521–565.

[79] Felsenstein J. 2004. *Inferring Phylogenies*. Sinauer Associates, Sunderland, MA.

[80] Felsenstein J. 2004. *Phylip (Phylogeny Inference Package) version 3.68*. Department of Genetics, University of Washington, Seattle, USA. `http://evolution.genetics.washington.edu/phylip/phylip.html`.

[81] Felsenstein J. 2005. *Using the quantitative genetic threshold model for inferences between and within species* volume 360,.

[82] Felsenstein J. 2008. Comparative methods with sampling error and within-species variation: Contrasts revisited and revised. *American Naturalist* **171**: 713–725.

[83] 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.

[84] Finn R. D., Mistry J., Tate J., Coggill P., Heger A., Pollington J. E., Gavin O. L., Gunasekaran P., Ceric G., Forslund K., Holm L., Sonnhammer E. L. L., Eddy S. R. & Bateman A. 2010. The Pfam protein families database. *Nucleic Acids Research* **38**: D211–D222.

[85] FitzJohn R. G. 2010. Quantitative traits and diversification. *Systematic Biology* **59**: 619–633.

[86] FitzJohn R. G., Maddison W. P. & Otto S. P. 2009. Estimating trait-dependent speciation and extinction rates from incompletely resolved phylogenies. *Systematic Biology* **58**: 595–611.

[87] Ford D. J. Encodings of cladograms and labeled trees 2010.

[88] Fox J. 2009. Aspects of the social organization and trajectory of the R project. *The R Journal* **1**: 5–13.

[89] Franklin J. 2009. *Mapping Species Distributions: Spatial Inference and Prediction*. Cambridge University Press, Cambridge.

[90] Freckleton R. P. 2009. The seven deadly sins of comparative analysis. *Journal of Evolutionary Biology* **22**: 1367–1375.

[91] Freedman D. A. 2009. *Statistical Models: Theory and Practice (Revised Edition)*. Cambridge University Press, Cambridge.

[92] Futuyma D. J. 1998. *Evolutionary Biology (Third Edition)*. Sinauer Associates, Sunderland, MA.

[93] 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.

[94] 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.

[95] Garland Jr. T. & Adolph S. C. 1991. Physiological differentiation of vertebrate populations. *Annual Review of Ecology and Systematics* **22**: 193–228.

[96] Garland Jr. T. & Carter P. A. 1994. Evolutionary physiology. *Annual Review of Physiology* **56**: 579–621.

[97] 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.

[98] Gascuel O. 1997. BIONJ: an improved version of the NJ algorithm based on a simple model of sequence data. *Molecular Biology and Evolution* **14**: 685–695.

[99] Gascuel O. 1997. Concerning the NJ algorithm and its unweighted version, UNJ. In: *Mathematical hierarchies and biology*, Mirkin B., McMorris F. R., Roberts F. S. & Rzhetsky A., editors, pages 149–170. American Mathematical Society, Providence, RI.

[100] Gascuel O. 2000. Data model and classification by trees: the minimum variance reduction (MVR) method. *Journal of Classification* **17**: 67–99.

[101] Gascuel O. 2000. On the optimization principle in phylogenetic analysis and the minimum-evolution criterion. *Molecular Biology and Evolution* **17**: 401–405.

[102] Gascuel O., Bryant D. & Denis F. 2001. Strengths and limitations of the minimum evolution principle. *Systematic Biology* **50**: 621–627.

[103] Gascuel O. & Steel M. 2006. Neighbor-joining revealed. *Molecular Biology and Evolution* **23**: 1997–2000.

[104] Gentleman R. 2004. Some perspectives on statistical computing. *Canadian Journal of Statistics* **32**: 209–226.

[105] Giannini N. P. 2003. Canonical phylogenetic ordination. *Systematic Biology* **52**: 684–695.

[106] 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.

[107] Gillespie D. T. 1996. Exact numerical simulation of the Ornstein-Uhlenbeck process and its integral. *Physical Review E* **54**: 2084–2091.

[108] Gittleman J. L. 1986. Carnivore life history patterns: allometric, phylogenetic and ecological associations. *American Naturalist* **127**: 744–771.

[109] Gittleman J. L. & Kot M. 1990. Adaptation: statistics and a null model for estimating phylogenetic effects. *Systematic Zoology* **39**: 227–241.

[110] Goldberg E. A., Lancaster L. T. & Ree R. H. 2011. Phylogenetic inference of reciprocal effects between geographic range evolution and diversification. *Systematic Biology*.

[111] Goudet J. 1999. An improved procedure for testing the effects of key innovations on rate of speciation. *American Naturalist* **153**: 549–555.

[112] Gower J. C. 1971. A general coefficient of similarity and some of its properties. *Biometrics* **27**: 623–637. pas de TAP.

[113] Grafen A. 1989. The phylogenetic regression. *Philosophical Transactions of the Royal Society of London. Series B. Biological Sciences* **326**: 119–157.

[114] Graham C. H., Ron S. R., Santos J. C., Schneider C. J. & Moritz C. 2004. Integrating phylogenetics and environmental niche models to ex-

plore speciation mechanisms in dendrobatid frogs. *Evolution* **58**: 1781–1793.

[115] Guindon S. & Gascuel O. 2003. A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. *Systematic Biology* **52**: 696–704.

[116] Hackett S. J. 1996. Molecular phylogenetics and biogeography of tanagers in the genus *Ramphocelus* (Aves). *Molecular Phylogenetics and Evolution* **5**: 368–382.

[117] Hadfield J. D. 2010. MCMC methods for multi-response generalized linear mixed models: The MCMCglmm R Package. *Journal of Statistical Software* **33**: 1–22.

[118] Hadfield J. D. & Nakagawa S. 2010. General quantitative genetic methods for comparative biology: phylogenies, taxonomies and multi-trait models for continuous and categorical characters. *Journal of Evolutionary Biology* **23**: 494–508.

[119] Hall B. G. 2004. *Phylogenetic Trees Made Easy: A How-to Manual (Second Edition)*. Sinauer Associates, Sunderland, MA.

[120] Hanazawa M., Narushima H. & Minaka N. 1995. Generating most parsimonious reconstructions on a tree: a generalization of the Farris–Swofford–Maddison method. *Discrete Applied Mathematics* **56**: 245–265.

[121] Hansen T. F. 1997. Stabilizing selection and the comparative analysis of adaptation. *Evolution* **51**: 1341–1351.

[122] Hansen T. F. & Martins E. P. 1996. Translating between microevolutionary process and macroevolutionary patterns: the correlation structure of interspecific data. *Evolution* **50**: 1404–1417.

[123] Hardy O. J. & Jost L. 2008. Interpreting and estimating measures of community phylogenetic structuring. *Journal of Ecology* **96**: 849–852.

[124] Hardy O. J. & Senterre B. 2007. Characterizing the phylogenetic structure of communities by an additive partitioning of phylogenetic diversity. *Journal of Ecology* **95**: 493–506.

[125] Harmon L. J., Weir J. T., Brock C. D., Glor R. E. & Challenger W. 2008. GEIGER: investigating evolutionary radiations. *Bioinformatics* **24**: 129–131.

[126] Harvey P. H., May R. M. & Nee S. 1994. Phylogenies without fossils. *Evolution* **48**: 523–529.

[127] Harvey P. H. & Pagel M. D. 1991. *The comparative method in evolutionary biology*. Oxford University Press, Oxford.

[128] Harvey P. H. & Purvis A. 1991. Comparative methods for explaining adaptations. *Nature* **351**: 619–624.

[129] 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.

[130] Hastings W. K. 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* **57**: 97–109.

[131] 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.

[132] Helmus M. R., Bland T. J., Williams C. K. & Ives A. R. 2007. Phylogenetic measures of biodiversity. *American Naturalist* **169**: E68–E83.

[133] Hendy M. D. & Penny D. 1989. A framework for the quantitative study of evolutionary trees. *Systematic Zoology* **38**: 297–309.

[134] Higgins D. & Lemey P. 2009. Multiple sequence alignment. In: *The Phylogenetic Handbook: A Practical Approach to Phylogenetic Analysis and Hypothesis Testing (Second Edition)*, Lemey P. Salemi. M. & Vandamme A.-M., editors, pages 68–108. Cambridge University Press, Cambridge.

[135] Holder M. & Lewis P. O. 2003. Phylogeny estimation: traditional and Bayesian approaches. *Nature Reviews Genetics* **4**: 275–284.

[136] Holland B. R., Huber K. T., Dress A. & Moulton V. 2002. δ plots: a tool for analyzing phylogenetic distance data. *Molecular Biology and Evolution* **19**: 2051–2059.

[137] Holland B. R., Huber K. T., Moulton V. & Lockhart P. J. 2004. Using consensus networks to visualize contradictory evidence for species phylogeny. *Molecular Biology and Evolution* **21**: 1459–1461.

[138] Holmes S. 2003. Statistics for phylogenetic trees. *Theoretical Population Biology* **63**: 17–32.

[139] Hordijk W. & Gascuel O. 2005. Improving the efficiency of SPR moves in phylogenetic tree search methods based on maximum likelihood. *Bioinformatics* **21**: 4338–4347.

[140] Housworth E. A., Martins E. P. & Lynch M. 2004. The phylogenetic mixed model. *American Naturalist* **163**: 84–96.

[141] Hudson R. R. Generating samples under a Wright–Fisher neutral model 2002.

[142] Huelsenbeck J. P. & Ronquist F. 2001. MRBAYES: Bayesian inference of phylogenetic trees. *Bioinformatics* **17**: 754–755.

[143] Hunter J. P. 1998. Key innovations and the ecology of macroevolution. *Trends in Ecology & Evolution* **13**: 31–36.

[144] Hurvich C. M., Simonoff J. S. & Tsai C.-L. 1998. Smoothing parameter selection in nonparametric regression using an improved Akaike information criterion. *Journal of the Royal Statistical Society. Series B. Methodological* **60**: 271–293.

[145] Huson D. H. & Bryant D. 2006. Application of phylogenetic networks in evolutionary studies. *Molecular Biology and Evolution* **23**: 254–267.

[146] Ihaka R. & Gentleman R. 1996. R: a language for data analysis and graphics. *Journal of Computational and Graphical Statistics* **5**: 299–314.

[147] Isaac N. J. B., Turvey S. T., Collen B., Waterman C. & Baillie J. E. M. 2007. Mammals on the EDGE: conservation priorities based on threat and phylogeny. *PLoS ONE* **2**: e296.

[148] Ives A. R. & Helmus M. R. In press. Generalized linear mixed models for phylogenetic analyses of community structure. *Ecological Monographs*.

[149] Ives A. R., Midford P. E. & Garland Jr. T. 2007. Within-species variation and measurement error in phylogenetic comparative methods. *Systematic Biology* **56**: 252–270.

[150] Jombart T. 2008. *adegenet*: A R package for the multivariate analysis of genetic markers. *Bioinformatics* **24**: 1403–1405.

[151] Jombart T., Balloux F. & Dray S. 2010. *adephylo*: New tools for investigating the phylogenetic signal in biological traits. *Bioinformatics* **26**: 1907–1909.

[152] Jombart T., Pavoine S., Devillard S. & Pontier D. 2010. Putting phylogeny into the analysis of biological traits: A methodological approach. *Journal of Theoretical Biology* **264**: 693–701.

[153] Jones D. T., Taylor W. R. & Thornton J. M. 1992. The rapid generation of mutation data matrices from protein sequences. *Computer Applications in the Biosciences* **8**: 275–282.

[154] 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.

[155] 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.

[156] Katoh K., Misawa K., Kuma K. & Miyata T. 2002. MAFFT: A novel method for rapid multiple sequence alignment based on fast Fourier transform. *Nucleic Acids Research* **30**: 3059–3066.

[157] Keiding N. 1975. Maximum likelihood estimation in the birth-and-death process. *Annals of Statistics* **3**: 363–372.

[158] Kembel S. W., Cowan P. D., Helmus M. R., Cornwell W. K., Morlon H., Ackerly D. D., Blomberg S. P. & Webb C. O. 2010. Picante: R tools for integrating phylogenies and ecology. *Bioinformatics* **26**: 1463–1464.

[159] Kendall D. G. 1948. On the generalized "birth-and-death" process. *Annals of Mathematical Statistics* **19**: 1–15.

[160] Kendall D. G. 1949. Stochastic processes and population growth. *Journal of the Royal Statistical Society. Series B. Methodological* **11**: 230–264.

[161] 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.

[162] Kimura M. 1981. Estimation of evolutionary distances between homologous nucleotide sequences. *Proceedings of the National Academy of Sciences USA* **78**: 454–458.

[163] Kimura M. 1983. *The Neutral Theory of Molecular Evolution*. Cambridge University Press, Cambridge.

[164] Kimura M. & Ohta T. 1969. The average number of generations until fixation of a mutant gene in a finite population. *Genetics* **61**: 763–771.
[165] Kingman J. F. C. 2000. Origin of the coalescent: 1974–1982. *Genetics* **156**: 1461–1463.
[166] Kirkpatrick M. & Slatkin M. 1993. Searching for evolutionary patterns in the shape of a phylogenetic tree. *Evolution* **47**: 1171–1181.
[167] Konishi S. & Kitagawa G. 2008. *Information Criteria and Statistical Modeling*. Springer, New York.
[168] Kosakovsky Pond S. L. & Muse S. V. 2004. Column sorting: rapid calculation of the phylogenetic likelihood function. *Systematic Biology* **53**: 685–692.
[169] Kosiol C. & Goldman N. 2005. Different versions of the Dayhoff rate matrix. *Molecular Biology and Evolution* **22**: 193–199.
[170] Kuhner M. K. & Felsenstein J. 1994. Simulation comparison of phylogeny algorithms under equal and unequal evolutionary rates. *Molecular Biology and Evolution* **11**: 459–468.
[171] Kuhner M. K., Yamato J. & Felsenstein J. 1998. Maximum likelihood estimation of population growth rates based on the coalescent. *Genetics* **149**: 429–434.
[172] Kumar S. 2003. MacTrees made easy—Review of "Phylogenetic trees made easy: a how-to manual for molecular biologists" by Barry G. Hall. *Molecular Phylogenetics and Evolution* **27**: 165–167.
[173] 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.
[174] Lake J. A. 1994. Reconstructing evolutionary trees from DNA and protein sequences: paralinear distances. *Proceedings of the National Academy of Sciences USA* **91**: 1455–1459.
[175] Lamm K. S. & Redelings B. D. 2009. Reconstructing ancestral ranges in historical biogeography: properties and prospects. *Journal of Systematics and Evolution* **47**: 369–382.
[176] Lanave C., Preparata G., Saccone C. & Serio G. 1984. A new method for calculating evolutionary substitution rates. *Journal of Molecular Evolution* **20**: 86–93.
[177] Lavin S. R., Karasov W. H., Ives A. R., Middleton K. M. & Garland T. 2008. Morphometrics of the avian small intestine compared with that of nonflying mammals: A phylogenetic approach. *Physiological and Biochemical Zoology* **81**: 526–550.
[178] Le S. Q. & Gascuel O. 2008. An improved general amino acid replacement matrix. *Molecular Biology and Evolution* **25**: 1307–1320.
[179] Legendre P. & Desdevises Y. 2009. Independent contrasts and regression through the origin. *Journal of Theoretical Biology* **259**: 727–743.
[180] Legendre P., Desdevises Y. & Bazin E. 2002. A statistical test for host–parasite coevolution. *Systematic Biology* **51**: 217–234.

[181] 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.

[182] Lento G. M., Hickson R. E., Chambers G. K. & Penny D. 1995. Use of spectral analysis to test hypotheses on the origin of pinninpeds. *Molecular Biology and Evolution* **12**: 28–52.

[183] Liang K.-Y. & Zeger S. L. 1986. Longitudinal data analysis using generalized linear models. *Biometrika* **73**: 13–22.

[184] Light J. E. & Hafner M. S. 2008. Codivergence in heteromyid rodents (Rodentia : Heteromyidae) and their sucking lice of the genus *Fahrenholzia* (Phthiraptera : Anoplura). *Systematic Biology* **57**: 449–465.

[185] Liu L., Yu L. & Pearl D. K. 2010. Maximum tree: a consistent estimator of the species tree. *Journal of Mathematical Biology* **60**: 95–106.

[186] Lockhart P. J., Steel M. A., Hendy M. D. & Penny D. 1994. Recovering evolutionary trees under a more realistic model of sequence evolution. *Molecular Biology and Evolution* **11**: 605–612.

[187] Losos J. B. 1994. An approach to the analysis of comparative data when a phylogeny is unavailable or incomplete. *Systematic Biology* **43**: 117–123.

[188] Losos J. B. & Adler F. R. 1995. Stumped by trees? A generalized null model for patterns of organismal diversity. *American Naturalist* **145**: 329–342.

[189] Losos J. B. & Glor R. E. 2003. Phylogenetic comparative methods and the geography of speciation. *Trends in Ecology & Evolution* **18**: 220–227.

[190] Lynch M. 1991. Methods for the analysis of comparative data in evolutionary biology. *Evolution* **45**: 1065–1080.

[191] MacArthur R. H. & Wilson E. O. 1967. *The theory of island biogeography.* Princeton University Press, Princeton.

[192] Maddison W. P. 1997. Gene trees in species trees. *Systematic Biology* **46**: 523–536.

[193] Maddison W. P. & Knowles L. L. 2006. Inferring phylogeny despite incomplete lineage sorting. *Systematic Biology* **55**: 21–30.

[194] Maddison W. P., Midford P. E. & Otto S. P. 2007. Estimating a binary character's effect on speciation and extinction. *Systematic Biology* **56**: 701–710.

[195] Magallón S. & Sanderson M. J. 2001. Absolute diversification rates in angiosperm clades. *Evolution* **55**: 1762–1780.

[196] Magee L. 1990. R^2 measures based on Wald and likelihood ratio joint significance tests. *American Statistician* **44**: 250–253.

[197] Martins E. P. 1996. Conducting phylogenetic comparative studies when the phylogeny is not known. *Evolution* **50**: 12–22.

[198] Martins E. P. & Garland Jr. T. 1991. Phylogenetic analyses of the correlated evolution of continuous characters: a simulation study. *Evolution* **45**: 534–557.

[199] 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. *American Naturalist* **149**: 646–667. Erratum vol. 153, p. 488.

[200] May R. M. 1990. Taxonomy as destiny. *Nature* **347**: 129–130.

[201] McConway K. J. & Sims H. J. 2004. A likelihood-based method for testing for nonstochastic variation of diversification rates in phylogenies. *Evolution* **58**: 12–23.

[202] McCullough B. D. 1999. Assessing the reliability of statistical software: Part II. *American Statistician* **53**: 149–159.

[203] McCullough B. D. & Vinod H. D. 1999. The numerical reliability of econometric software. *Journal of Economic Literature* **37**: 633–665.

[204] McLeod A. I. 1993. Parsimony, model adequacy and periodic correlation in time-series forecasting. *International Statistical Review* **61**: 387–393.

[205] 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.

[206] Minin V., Abdo Z., Joyce P. & Sullivan J. 2003. Performance-based selection of likelihood models for phylogeny estimation. *Systematic Biology* **52**: 674–683.

[207] Moore B. R., Chan K. M. A. & Donoghue M. J. 2004. Detecting diversification rate variation in supertrees. In: *Phylogenetic Supertrees: Combining Information to Reveal the Tree of Life*, Bininda-Emonds O. R. P., editor, pages 487–533. Kluwer Academic, New York.

[208] Moran P. A. P. 1950. Notes on continuous stochastic phenomena. *Biometrika* **37**: 17–23.

[209] Morrone J. J. 2009. *Evolutionary Biogeography: An Integrative Approach with Case Studies*. Columbia University Press, New York.

[210] Motani R. & Schmitz L. 2011. Phylogenetic versus functional signals in the evolution of form–function relationships in terrestrial vision. *Evolution* doi:10.1111/j.1558-5646.2011.01271.x.

[211] Murrell P. 2006. *R Graphics*. Chapman & Hall/CRC, Boca Raton, FL.

[212] Nagelkerke N. J. D. 1991. A note on a general definition of the coefficient of determination. *Biometrika* **78**: 691–692.

[213] Narushima H. & Hanazawa M. 1997. A more efficient algorithm for MPR problems in phylogeny. *Discrete Applied Mathematics* **80**: 231–238.

[214] Nee S., Holmes E. C., Rambaut A. & Harvey P. H. 1995. Inferring population history from molecular phylogenies. *Philosophical Transac-*

tions of the Royal Society of London. Series B. Biological Sciences **349**: 25–31.

[215] Nee S. & May R. M. 1997. Extinction and the loss of evolutionary history. *Science* **278**: 692–694.

[216] 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.

[217] 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.

[218] Nei M. & Kumar S. 2000. *Molecular evolution and phylogenetics*. Oxford University Press, Oxford.

[219] Nixon K. C. & Wheeler Q. D. 1992. Measures of phylogenetic diversity. In: *Extinction and Phylogeny*, Novacek M. J. & Wheeler Q. D., editors, pages 216–234. Columbia University Press, New York.

[220] Notredame C., Higgins D. & Heringa J. 2000. T-Coffee: A novel method for multiple sequence alignments. *Journal of Molecular Biology* **302**: 205–217.

[221] Oakley T. H. 2003. Maximum likelihood models of trait evolution. *Comments on Theoretical Biology* **8**: 1–17.

[222] Ollier S., Couteron P. & Chessel D. 2006. Orthonormal transform to decompose the variance of a life-history trait across a phylogenetic tree. *Biometrics* **62**: 471–477.

[223] 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**: 37–45.

[224] Pagel M. 1999. Inferring the historical patterns of biological evolution. *Nature* **401**: 877–884.

[225] Pan W. 2001. Akaike's information criterion in generalized estimating equations. *Biometrics* **57**: 120–125.

[226] 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.

[227] 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.

[228] 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.

[229] Paradis E. 2004. Can extinction rates be estimated without fossils? *Journal of Theoretical Biology* **229**: 19–30.

[230] Paradis E. 2005. Statistical analysis of diversification with species traits. *Evolution* **59**: 1–12.

[231] Paradis E. 2010. pegas: An R package for population genetics with an integrated–modular approach. *Bioinformatics* **26**: 419–420.

[232] Paradis E. 2011. Time-dependent speciation and extinction from phylogenies: A least squares approach. *Evolution* **65**: 661–672.
[233] Paradis E. In press. Shift in diversification in sister-clade comparisons: A more powerful test. *Evolution*.
[234] Paradis E. & Claude J. 2002. Analysis of comparative data using generalized estimating equations. *Journal of Theoretical Biology* **218**: 175–185.
[235] Paradis E., Claude J. & Strimmer K. 2004. APE: analyses of phylogenetics and evolution in R language. *Bioinformatics* **20**: 289–290.
[236] Pauplin Y. 2000. Direct calculation of a tree length using a distance matrix. *Journal of Molecular Evolution* **51**: 41–47.
[237] Pavoine S., Baguette M. & Bonsall M. B. 2010. Decomposition of trait diversity among the nodes of a phylogenetic tree. *Ecological Monographs* **80**: 485–507.
[238] Pavoine S., Ollier S. & Dufour A.-B. 2005. Is the originality of a species measurable? *Ecology Letters* **8**: 579–586.
[239] Pavoine S., Ollier S. & Pontier D. 2005. Measuring diversity from dissimilarities with Rao's quadratic entropy: Are any dissimilarities suitable? *Theoretical Population Biology* **67**: 231–239.
[240] Pavoine S., Ollier S., Pontier D. & Chessel D. 2008. Testing for phylogenetic signal in phenotypic traits: New matrices of phylogenetic proximities. *Theoretical Population Biology* **73**: 79–91.
[241] Penny D. & Hendy M. D. 1985. The use of tree comparison metrics. *Systematic Zoology* **34**: 75–82.
[242] Pinheiro J. C. & Bates D. M. 2000. *Mixed-Effects Models in S and S-PLUS*. Springer, New York.
[243] Polanski A., Bobrowski A. & Kimmel M. 2003. A note on distributions of times to coalescence, under time-dependent population size. *Theoretical Population Biology* **63**: 33–40.
[244] 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.
[245] Posada D. & Crandall K. A. 1998. MODELTEST: testing the model of DNA substitution. *Bioinformatics* **14**: 817–818.
[246] Posada D. & Crandall K. A. 2001. Selecting the best-fit model of nucleotide substitution. *Systematic Biology* **50**: 580–601.
[247] 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.
[248] Purvis A. & Garland Jr. T. 1993. Polytomies in comparative analyses of continuous characters. *Systematic Biology* **42**: 569–575.
[249] Purvis A., Nee S. & Harvey P. H. 1995. Macroevolutionary inferences from primate phylogeny. *Proceedings of the Royal Society of London. Series B. Biological Sciences* **260**: 329–333.

[250] 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.

[251] 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.

[252] 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.

[253] R Development Core Team. 2011. *R language definition. Version 2.12.2.* R Foundation for Statistical Computing, Vienna, Austria.

[254] R Development Core Team. 2011. *Writing R extensions. Version 2.12.2.* R Foundation for Statistical Computing, Vienna, Austria.

[255] Rabosky D. L. 2006. LASER: A maximum likelihood toolkit for detecting temporal shifts in diversification rates from molecular phylogenies. *Evolutionary Bioinformatics* **2**: 257–260.

[256] Rabosky D. L. 2006. Likelihood methods for detecting temporal shifts in diversification rates. *Evolution* **60**: 1152–1164.

[257] Rabosky D. L., Donnellan S. C., Talaba A. L. & Lovette I. J. 2007. Exceptional among-lineage variation in diversification rates during the radiation of Australia's most diverse vertebrate clade. *Proceedings of the Royal Society of London. Series B. Biological Sciences* **274**: 2915–2923.

[258] Rabosky D. L. & Lovette I. J. 2008. Density dependent diversification in North American wood warblers. *Proceedings of the Royal Society of London. Series B. Biological Sciences* **275**: 2363–2371.

[259] Rabosky D. L. & Lovette I. J. 2008. Explosive evolutionary radiations: decreasing speciation or increasing extinction through time? *Evolution* **62**: 1866–1875.

[260] Rambaut A. & Grassly N. C. 1997. Seq-Gen: An application for the Monte Carlo simulation of DNA sequence evolution along phylogenetic trees. *Computer Applications in the Biosciences* **13**: 235–238.

[261] Rao C. R. 1982. Diversity and dissimilarity coefficients: A unified approach. *Theoretical Population Biology* **21**: 24–43.

[262] Raymond E. S. The Cathedral and the Bazaar 2000. `http://www.catb.org/~esr/writings/cathedral-bazaar/`.

[263] Redding D. W. & Mooers A. O. 2006. Incorporating evolutionary measures into conservation prioritization. *Conservation Biology* **20**: 1670–1678.

[264] Ree R. H. & Sanmartín I. 2009. Prospects and challenges for parametric models in historical biogeographical inference. *Journal of Biogeography* **36**: 1211–1220.

[265] Ree R. H. & Smith S. A. 2008. Maximum likelihood inference of geographic range evolution by dispersal, local extinction, and cladogenesis. *Systematic Biology* **57**: 4–14.

[266] Revell L. J. 2009. Size-correction and principal components for interspecific comparative studies. *Evolution* **63**: 3258–3268.
[267] Ridley M. 1992. Darwin sound on comparative method. *Trends in Ecology & Evolution* **7**: 37.
[268] Rochet M.-J., Cornillon P.-A., Sabatier R. & Pontier D. 2000. Comparative analysis of phylogenetic and fishing effects in life history patterns of teleost fishes. *Oikos* **91**: 255–270.
[269] Rodríguez F., Oliver J. L., Marín A. & Medina J. R. 1990. The general stochastic model of nucleotide substitution. *Journal of Theoretical Biology* **142**: 485–501.
[270] Ronquist F. 1997. Dispersal–vicariance analysis: a new approach to the quantification of historical biogeography. *Systematic Biology* **46**: 195–203.
[271] Rota-Stabelli O., Yang Z. & Telford M. 2009. MtZoa: a general mitochondrial amino acid substitutions model for animal evolutionary studies. *Molecular Phylogenetics and Evolution* **52**: 268–272.
[272] Rzhetsky A. & Nei M. 1992. A simple method for estimating and testing minimum-evolution trees. *Molecular Biology and Evolution* **9**: 945–967.
[273] Rzhetsky A. & Nei M. 1993. Theoretical foundation of the minimum-evolution method of phylogenetic inference. *Molecular Biology and Evolution* **10**: 1073–1095.
[274] Sakamoto M., Lloyd G. T. & Benton M. J. 2010. Phylogenetically structured variance in felid bite force: the role of phylogeny in the evolution of biting performance. *Journal of Evolutionary Biology* **23**: 463–478.
[275] Sanderson M. J. 1997. A nonparametric approach to estimating divergence times in the absence of rate constancy. *Molecular Biology and Evolution* **14**: 1218–1231.
[276] Sanderson M. J. 2002. Estimating absolute rates of molecular evolution and divergence times: a penalized likelihood approach. *Molecular Biology and Evolution* **19**: 101–109.
[277] Sargent R. D. 2004. Floral symmetry affects speciation rates in angiosperms. *Proceedings of the Royal Society of London. Series B. Biological Sciences* **271**: 603–608.
[278] Schliep K. P. 2011. phangorn: Phylogenetic analysis in R. *Bioinformatics* **27**: 592–593.
[279] Schluter D., Price T., Mooers A. Ø. & Ludwig D. 1997. Likelihood of ancestor states in adaptive radiation. *Evolution* **51**: 1699–1711.
[280] Schmitz L. & Motani R. 2011. Nocturnality in dinosaurs inferred from scleral ring and orbit morphology. *Science* **332**: 705–708.
[281] Schoener T. W. 1968. The *Anolis* lizards of Bimini: Resource partitioning in a complex fauna. *Ecology* **49**: 704–726.
[282] Shi P. & Tsai C.-L. 2002. Regression model selection—a residual likelihood approach. *Journal of the Royal Statistical Society. Series B. Methodological* **64**: 237–252.

[283] Shimodaira H. & Hasegawa M. 1999. Multiple comparisons of log-likelihoods with applications to phylogenetic inference. *Molecular Biology and Evolution* **16**: 1114–1116.

[284] Sibley C. G. & Ahlquist J. E. 1990. *Phylogeny and classification of birds: a study in molecular evolution*. Yale University Press, New Haven, CT.

[285] Sibley C. G. & Monroe Jr. B. L. 1990. *Distribution and taxonomy of birds of the world*. Yale University Press, New Haven, CT.

[286] Slowinski J. B. & Guyer C. 1993. Testing whether certain traits have caused amplified diversification: an improved method based on a model of random speciation and extinction. *American Naturalist* **142**: 1019–1024.

[287] Stamatakis A. 2006. RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics* **22**: 2688–2690.

[288] 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.

[289] 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.

[290] Steel M. A. & Penny D. 1993. Distributions of tree comparison metrics—Some new results. *Systematic Biology* **42**: 126–141.

[291] Stephens M. A. 1974. EDF statistics for goodness of fit and some comparisons. *Journal of American Statistical Association* **69**: 730–737.

[292] 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.

[293] 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.

[294] Swofford D. L. & Maddison W. P. 1987. Reconstructing ancestral character states under Wagner parsimony. *Mathematical Biosciences* **87**: 199–229.

[295] 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.

[296] 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.

[297] 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.

[298] 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.

[299] 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.

[300] Thorne J. L., Kishino H. & Painter I. S. 1998. Estimating the rate of evolution of the rate of molecular evolution. *Molecular Biology and Evolution* **15**: 1647–1657.

[301] Tufte E. R. 2001. *The Visual Display of Quantitative Information (Second Edition)*. Graphics Press, Cheshire, CT.

[302] Valiente G. 2009. *Combinatorial Pattern Matching Algorithms in Computational Biology Using Perl and R*. Chapman & Hall/CRC, Boca Raton.

[303] Vamosi S. M., Heard S. B., Vamosi J. C. & Webb C. O. 2009. Emerging patterns in the comparative analysis of phylogenetic community structure. *Molecular Ecology* **18**: 572–592.

[304] Vamosi S. M. & Vamosi J. C. 2005. Endless tests: guidelines for analysing non-nested sister-group comparisons. *Evolutionary Ecology Research* **7**: 567–579.

[305] Vane-Wright R. I., Humphries C. J. & Williams P. H. 1991. What to protect? Systematics and the agony of choice. *Biological Conservation* **55**: 235–254.

[306] Venables W. N. & Ripley B. D. 2000. *S Programming*. Springer, New York.

[307] Venables W. N. & Ripley B. D. 2002. *Modern Applied Statistics with S (Fourth Edition)*. Springer, New York.

[308] Wake D. B., Hadly E. A. & Ackerly D. D. 2009. Biogeography, changing climates, and niche evolution. *Proceedings of the National Academy of Sciences USA* **106**: 19631–19636.

[309] Wakeley J. 2009. *Coalescent Theory: An Introduction*. Roberts & Company Publishers, Greenwood Village, CO.

[310] Warren D. L., Glor R. E. & Turelli M. 2008. Environmental niche equivalency versus conservatism: quantitative approaches to niche evolution. *Evolution* **62**: 2868–2883.

[311] Whelan S. & Goldman N. 2001. A general empirical model of protein evolution derived from multiple protein families using a maximum-likelihood approach. *Molecular Biology and Evolution* **18**: 691–699.

[312] White W. T., Hills S. F., Gaddam R., Holland B. R. & Penny D. 2007. Treeness triangles: visualizing the loss of phylogenetic signal. *Molecular Biology and Evolution* **24**: 2029–2039.

[313] Wiegmann B., Mitter C. & Farrell B. 1993. Diversification of carnivorous parasitic insects: extraordinary radiation or specialized dead end? *American Naturalist* **142**: 737–754.

[314] Yang Z. 1994. Estimating the pattern of nucleotide substitution. *Journal of Molecular Evolution* **39**: 105–111.

[315] Yang Z. 1994. Maximum likelihood phylogenetic estimation from DNA sequences with variable rates over sites: approximate methods. *Journal of Molecular Evolution* **39**: 306–314.

[316] Yang Z. 1996. Among-site rate variation and its impact on phylogenetic analyses. *Trends in Ecology & Evolution* **11**: 367–372.

[317] Yang Z. 2006. *Computational Molecular Evolution*. Oxford University Press, Oxford.

[318] Yang Z., Nielsen R. & Hasegawa M. 1998. Models of amino acid substitution and applications to Mitochondrial protein evolution. *Molecular Biology and Evolution* **15**: 1600–1611.

Index