Comparative Methods and Data Analysis in R

Marguerite A. Butler[1,2], Brian C. O'Meara[3], and Jason Pienaar[1,4]

[1]Department of Zoology, University of Hawaii, Honolulu, HI 96822
[2]mbutler@hawaii.edu
[3]National Evolutionary Synthesis Center, 2024 West Main Street, Suite A200, Durham, NC 27705, bcomeara@nescent.org
[4]jasonpienaar@gmail.com

November 16, 2008

Contents

Chapter 1

Preliminaries

1.1 Computer Requirements and Installing R

Here is a list of the software we will be using in class. *If you've installed these software a long time ago, please update to recent versions to avoid compatibility issues.*

Some of these components are required in order to install packages from source code including: C compiler (gcc), a fortran compiler (e.g., gfortran), and X11. In addition, we will be using LaTex, so please install MacTeX-2007 and TeXShop.

Computers We will be using macintoshes running Tiger or Leopard (OS 10.4.X or 10.5.X).

R version 2.7.1 or later (The later versions in a series usually have bug fixes). You can install R from the binaries available at the R website http://www.r-project.org. They are available as disk images and very straightforward to use. On the left Menu bar, click on "CRAN" (the Comprehensive R Archive Network). Choose a mirror (the closest geographically), then click on your operating system (MacOS X) and click on R-2.7.1.dmg. Follow the directions from there.

Xcode Tools This contains the C/C++ compiler. Install from the system disks that came with your computer. For Tiger, if you don't have the disks, you can also download it from the Apple Developers site after signing up for a free account.

gfortran Included with your R binary.

X11 Comes with OS X, but it is an optional install.

LaTeX Install the tex engine (MacTeX-2007; this is the actual tex/LaTeX software) at http://www.tug.org/mactex and a gui frontend for the mac at http://www.uoregon.edu/~koch/texshop/obtaining.html (TeXShop; this is a nice interface

with menu items and other features). Note that after you install MacTeX-2007, all you have to do is click on the "Latest TeXShop". Nevermind the stuff about the "upgrade", this is a complete install of the gui frontend TeXShop (just without the TeX engine).

You can find detailed instructions on how to install these software components and links to the software itself at the R website , under FAQ's > F for Mac OS X FAQ > Building R from sources. Check out sections 2.1.1, 2, 4, and 8.

All of these installations are pretty straightforward, so if you are bringing your own computer, please try to do the installs prior to the course. If you have trouble, I will be happy to help you when we meet. However, especially with Leopard, please make sure you have Xcode Tools and X11 installed from the discs that came with your computer. This software is now encoded so that it is not possible to install from someone else's disk.

1.2 R packages

Many of the packages that we will be using are available on CRAN. The easiest way to install from CRAN is to do it from within R. From the "Packages & Data" menu option, choose "Package Installer". You may have to choose a mirror if you haven't done so already (choose a geographically close one). The package installer should open up with "CRAN (binaries)" already selected. Click on "Get list", which will refresh the menu with all the available packages and the version numbers that you have installed. Highlight the packages that you want to install, choose "Install Dependencies" then click on "Install Selected". You can also download the packages from the R website, on the left menu bar click on CRAN.

Install the following from CRAN (binaries):

ade4

geiger

subplex

rgl

xtable

And from CRAN (sources), after clicking on "Get list" again:

ape version 2.2-1 Install from source to get the latest version (if it's avaiable as a binary, though, go ahead and use it). http://ape.mpl.ird.fr/

Install the latest development version of ouch and phylobase from the R-forge repository (R-forge is devoted to packages under development, whereas those on CRAN should be finished or more fully tested packages):

ouch version 2.3-9 Select "Other Repository", enter the url in the textbox: `http://r-forge.r-project.org/` If you are running R 2.7.1, you can install ouch and phylobase from binaries. If you are running R 2.6.2, you must install from source. In this case, uncheck "Binary Format Packages" before clicking on "Get List" and "Install Selected".

phylobase version 0.3 Same installation procedure as `ouch`.

1.3 Comparative Methods in R References

Phylogenetic Comparative Methods Wiki Tutorials and overview of methods available in R for phylogenetic comparative analysis. The wiki grew out of a Hackathon on Comparative Methods in R held at the National Evolutionary Synthesis Center (NESCent) 10-14 December 2007.

Analysis of Phylogenetics and Evolution with R A book written by Paradis (2006). This book is a very useful reference on how to do evolutionary analyses using the `ape` package, written by one its developers. It is available from Springer and Amazon.

1.4 General R References

An introduction to R A comprehensive and easy-to-follow tutorial produced by the R Development Core Team.

R for Beginners A tutorial by Emmanuel Paradis.

1.5 General Comparative Methods References

The Comparative Method in Evolutionary Biology An book written by Harvey and Pagel (1991). It is getting a bit old now, but it is still a comprehensive general reference and a useful overview of many methods.

Phylogenies and the Comparative Method in Animal Behavior A volume edited by Martins (1996), with explanation of methods and nice examples. Available at books.google.com for preview.

Inferring Phylogenies Joe Felsenstein's (2004) book. It is mostly about phylogeny reconstruction, but there is a chapter on comparative methods that gives Felsenstein's perspective on comparative analysis.

1.6 Help! and Useful References

1.6.1 general R help

Jonathan Baron's R help page Bookmark this page! It is the best search engine to find R help. It searches the huge archives of the R-help listserv as well as all R documentation pages. For more technical help, you can also include the R-dev (developers) listserv in the search.

1 page R reference card by Jonathan Baron.

4 page R reference card by Tom Short.

1.6.2 comparative methods R help

R-sig-phylo mailing list The listserv for the R Phylogenetic Comparative Method Special Interest Group. It is archived here.

1.7 Directories and File organization

In order for R to interact with the files on your computer (i.e., for INPUT/OUTPUT), R needs to know the path to your working directory. This is where R is "parked" on your computer, and will look here for external files, or will write output files to here.

Mac the default working directory (on your computer) is your User directory. For example: "/Users/marguerite". Or where you opened your .R file (more on this later).

PC default is "C:/Program Files/R/R-2.7.1" (your installed R version number).

Linux/UNIX or running R in a terminal default is where you started R.

Chapter 2

Finding package specific help

R has a rich library of "packages", or source code written for a specific purposes by individuals out there in the open source community, as well as extensive help documentation. This section explains how to find package-specific help.

For example, you might want to know more about the phylogeny plotting function in `ape`. You can use`plot(tree)` to call the function, so you might think that you can find the help page by using `help(plot)`. However, this brings up the generic plot function which doesn't say anything about the one you want (the tree plotting function in `ape`.

What is going on is that `ape` has a method set for plotting objects of the class `phylo`, so that you don't have to remember the specific function name. This is actually a wonderful feature of object-oriented programming, otherwise you would have to remember thousands of functions, all uniquely named.

So how do we find the one we want? You could try:

```
> help(plot, package="ape")
```

But you will see that this doesn't return anything. This means that the actual plotting function in `ape` is named something else (R requires all named functions in packages to be documented).

You have several options:

help.search('plot') does a "fuzzy" (i.e., not exact) search for plot in any documentation. It will return a huge list, you can look through them by package.

help.start() brings up an html browser, click on "Packages", then "ape", then look through the p's and you will see "plot.phylo". Browsing through the help is very useful for beginners

help(package="ape") will return the package's main help page, where you can see a list of functions, but they are not clickable. Once you locate the name of the function you can follow up with a help(plot.phylo).

methods(plot) will return all of the methods written for the generic plot call. Looking through it, you might guess that `plot.phylo` is the one you want. NOTE: this only works for S3 methods.

Chapter 3

Simple Comparative Analyses in R

Chapter Topics:

- Why use comparative methods?
- Learn how to run independent contrasts, phylogenetic GLS, Ancestral Reconstruction methods using R package ape

  Run modified examples from help files

  Simulate some comparative data and run analyses

Skills: Loading and using packages, using functions, practice with `ape`, plotting, creating data objects, accessing help, traversing your file directory.

3.1 Why use comparative methods (and a bit about how they work)

Comparative methods are one of the oldest means for studying adaptation and evolutionary processes in general. Comparative methods were in use prior to Darwin.

See lecture slides.

3.2 Running simple comparative analyses using `ape`: or a tour through R using phylogenetic examples

In this section, we will run some comparative analyses using `ape`. See `ape's` homepage at http://ape.mpl.ird.fr/.

Go to your R console and load the ape package into active memory. Type at the prompt ("require" is nicer than "load" because "load" will just load the package. "require" checks if the package is already loaded first):

```
> require(ape)
```

3.2.1 Getting help

R has great built-in help facilities. Once you get used to R's syntax (the form of R functions and data), you will find them incredibly useful. But first to access the help for a specific function, you need to know what it is called. Access the main help page for the package `ape`:

```
> help(package="ape")
```

Notice that as you type `help(` you start to see the function definition on the bottom of the console window. It shows you how to call the function (what variables it expects).

Packages are generally a set of functions that are loaded from some (hidden) directory on your computer into active memory, so that you can use them by name. Now that you know the names of the functions, you can access specific help pages directly. Try the help page for independent contrasts:

```
> help(pic)
```

Looking at the help page, notice that there are sections (these are common to most help pages):

Description what it does

Usage the format for calling the function (making it run)

Arguments explanation for each of the arguments, their type, and what they represent

Details more explanation

Value what is returned from calling the function

Author

References

See Also other functions to check out

Examples Often the most valuable section, with examples that actually work. You can test them out by cutting and pasting into the R console.

This is modified from the sample code given in the pic documentation and Paradis (2006). First use the `read.tree` function to create a phylogenetic tree in R for the primates, save it in an object (a variable) called "tree.primates". Some basic ways to get information about R objects is to just type the name of the object on the command line (it will return some info or the value itself), or using a function called `summary`:

```
> tree.primates <- read.tree(text="((((Homo:0.21,Pongo:0.21):0.28,Macaca:0.49):
+ 0.13,Ateles:0.62):0.38,Galago:1.00);")
> tree.primates

Phylogenetic tree with 5 tips and 4 internal nodes.

Tip labels:
[1] "Homo"   "Pongo"  "Macaca" "Ateles" "Galago"

Rooted; includes branch lengths.

> summary(tree.primates)

Phylogenetic tree: tree.primates

  Number of tips: 5
  Number of nodes: 4
  Branch lengths:
    mean: 0.415
    variance: 0.08208571
    distribution summary:
   Min. 1st Qu.  Median 3rd Qu.    Max.
 0.1300  0.2100  0.3300  0.5225  1.0000
  No root edge.
  Tip labels: Homo
              Pongo
              Macaca
              Ateles
              Galago
  No node labels.
```

For trees, perhaps the best thing to do is to plot it:

```
> plot(tree.primates)
```

You can resize the pdf window by grabbing the corners. You can also save the plot to a pdf file using the `pdf` function, which opens a pdf graphics device driver. Just give your pdf file a name, then replot the tree, then turn the pdf device off again:

```
> pdf(file="primatetree.pdf")
> plot(tree.primates)
> dev.off()
```

Now, where did the file go? It is saved in your working directory, which on a Mac is wherever you started the R program, or your home user directory by default. You don't want all your work going there. So let's take a moment to set up some nice directories.

3.2.2 Directories and File organization

In order for R to interact with the files on your computer (i.e., for INPUT/OUTPUT), R needs to know the path to your working directory. This is where R is "parked" on your computer, and will look here for external files, or will write output files to here.

Mac the default working directory (on your computer) is your User directory. For example: "/Users/marguerite". Or where you opened your .R file (more on this later).

PC default is "C:/Program Files/R/R-2.7.1" (your installed R version number).

Linux/UNIX or running R in a terminal default is where you started R.

Course Directory Organization

So let's create a working directory. For the purposes of this course, at the top level of your user directory, create a folder called `Rcourse`. You will have to do this outside of R. Either create the `Rcourse` folder through the Finder or open a terminal, change to your user directory if you're not there via "`cd` " then "`mkdir Rcourse`."

As you go along, you will want to keep things tidy. Two ways (of probably many) to organize yourself is either by having a `data` and `Rdata` folder or to have separate folders for each project/assignment:

`Rcourse` the main project file for the course. It will contain all source code (scripts) and direct output. If this folder gets too big, we can make subfolders.

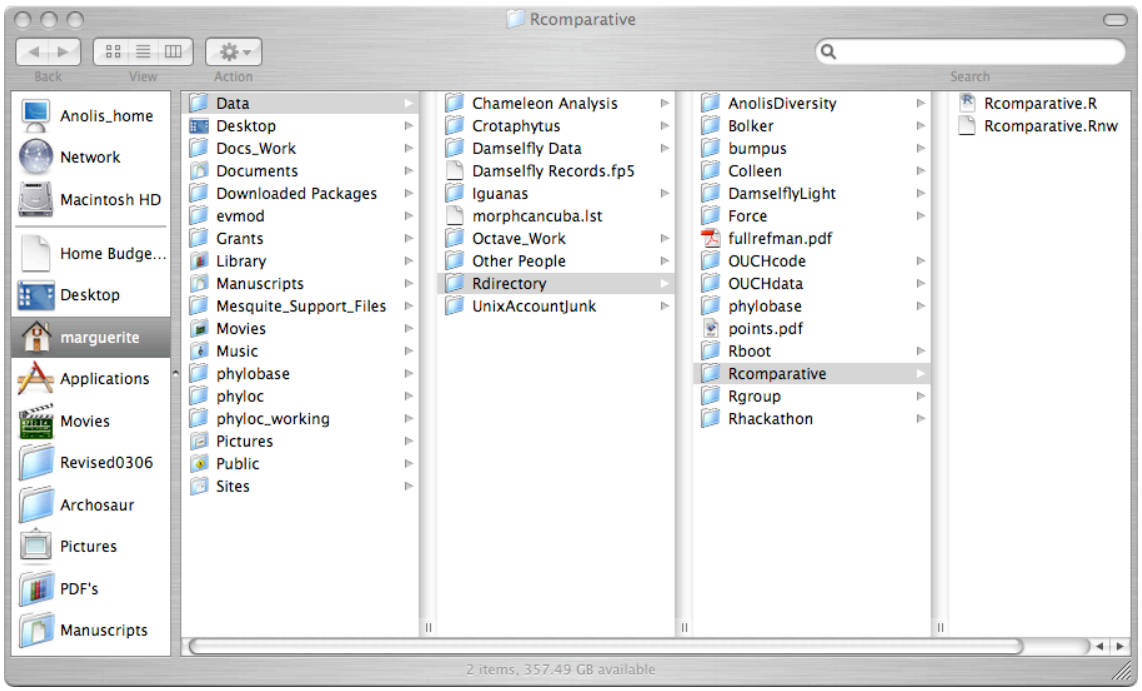


Figure 3.1: An example of directory organization to keep your R programming projects organized. Here the Rcomparative folder is moved to its final location. When you are actively working on a project, it may be more convenient to have it at the top level of your user directory (here "marguerite").

`data` to store our raw data input files (spreadsheets and text files). It should be within `Rcourse`.

`Rdata` for writing processed R data files. After setting this up, future analyses or scripts can access these files directly, rather than working from the raw data files.

The idea behind this method is that you want all of your "input" to be pristine and unaltered. The Rdata, on the other hand is semi-processed data, which you may want for further analysis. It is roughly input and output (but here the output may be used further).

When you are done with the course, you can move it to an appropriate place in your file heirarchy. As an example, this is the way I organize my personal computer (Fig. 3.1): I like to have all of my Data and analyses in a folder called "Data" at the top level of my User directory. Within it, I save all of my R code and analyses in an folder called "Rdirectory". Inside that, I have separate folders for each project. This one is called "Rcourse".

Moving through the directories

You can "get" your current working directory from within R. You can also set your working directory (after creating the directories first). This is the filepath from the root directory of my computer:

```
> getwd()
> setwd("/Users/marguerite/Rcourse")
```

And this is the filepath from my "home" directory which is "marguerite":

```
> setwd("~/Rcourse")
```

As you may have guessed, the "filepath" is the path to your files. In the Unix file system, the "root" is signified by starting the filepath with "/". You can't go up any more folders from the "root". Anything to the right are names of the folders within the root, and within that folder, etc. The first example above is called an *absolute filepath*. You can also use *relative filepaths*, which navigate relative to where you are. In this case, start with a name rather than ". Some useful special characters are:

`~` a special character for "my user directory"

`..` which means to go up one level

`.` which means the current directory (here)

`/` separator between folders or levels. If you begin your filepath with `/` with nothing preceeding it, this indicates an absolute file path starting from the root.

For example, if you wanted to back up to your user directory and change to a project called "`MyFirstAnalysis`", you would have to go up one directory and then specify the folder name, so filepath would be "`../MyFirstAnalysis`". To go up two directories and then into a new directory, use "`../../MyFirstAnalysis`."

Now try rerunning the code (you can get the lines you typed or cut and pasted by hitting the up arrow, or by clicking on the history icon (the blue and yellow striped box), and voila! You will see the pdf appear in your Rcomparative folder.

```
> setwd("~/Rcourse")
> pdf(file="primatetree.pdf")
> plot(tree.primates)
> dev.off()
```

3.2.3 Running Independent Contrasts using ape

Now back to our example using ape. Let's create two continuous characters. We also assign "names" to the entries so we know which species are associated with which datapoint.

```
> X <- c(4.09434, 3.61092, 2.37024, 2.02815, -1.46968)
> Y <- c(4.74493, 3.33220, 3.36730, 2.89037, 2.30259)
> X

[1]  4.09434  3.61092  2.37024  2.02815 -1.46968

> Y

[1] 4.74493 3.33220 3.36730 2.89037 2.30259

> names(X) <- names(Y) <- c("Homo", "Pongo", "Macaca", "Ateles", "Galago")
> X

    Homo    Pongo   Macaca   Ateles   Galago
 4.09434  3.61092  2.37024  2.02815 -1.46968

> Y

   Homo   Pongo  Macaca  Ateles  Galago
4.74493 3.33220 3.36730 2.89037 2.30259
```

Compute phylogenetically independent contrasts using the ape function pic and save them as new objects called "pic.X" and "pic.Y". The names attribute we assigned to the X and Y values above are very important here, as they will be used to match our comparative data to the species on the tips of the tree. If we have no names, pic assumes that they are in the same order as the tree (**be careful!!**).

We can now apply ordinary statistics to these PIC values. R has a huge number of statistical functions, including tests of correlation and linear models (regression):

```
> pic.X <- pic(X, tree.primates)
> pic.Y <- pic(Y, tree.primates)
> pic.X
```

```
        6         7         8         9
3.3583189 1.1929263 1.5847416 0.7459333

> pic.Y

        6         7         8         9
0.8970604 0.8678969 0.7176125 2.1798897

> cor.test(pic.X, pic.Y)

        Pearson's product-moment correlation

data:  pic.X and pic.Y
t = -0.8562, df = 2, p-value = 0.4821
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.9874751  0.8823934
sample estimates:
       cor
-0.5179156

> lm.YX <- lm(pic.Y ~ pic.X - 1) # this is a regression of Y "as a function" of X,
>                                # with -1 meaning no intercept (through origin)
> summary(lm.YX)    # this shows us the p-values and summary statistics

Call:
lm(formula = pic.Y ~ pic.X - 1)

Residuals:
       6        7        8        9
-0.55351  0.35263  0.03311  1.85770

Coefficients:
      Estimate Std. Error t value Pr(>|t|)
pic.X   0.4319     0.2865   1.508    0.229

Residual standard error: 1.138 on 3 degrees of freedom
Multiple R-squared: 0.4311,        Adjusted R-squared: 0.2414
F-statistic: 2.273 on 1 and 3 DF,  p-value: 0.2288
```

Great! We ran our first comparative analysis. But what happened? Why did we get what we did? Do we believe it? Let's take a step back and first look at the raw data (Fig. 3.2):

```
> plot(X, Y)   # same as plot(Y ~ X)
```

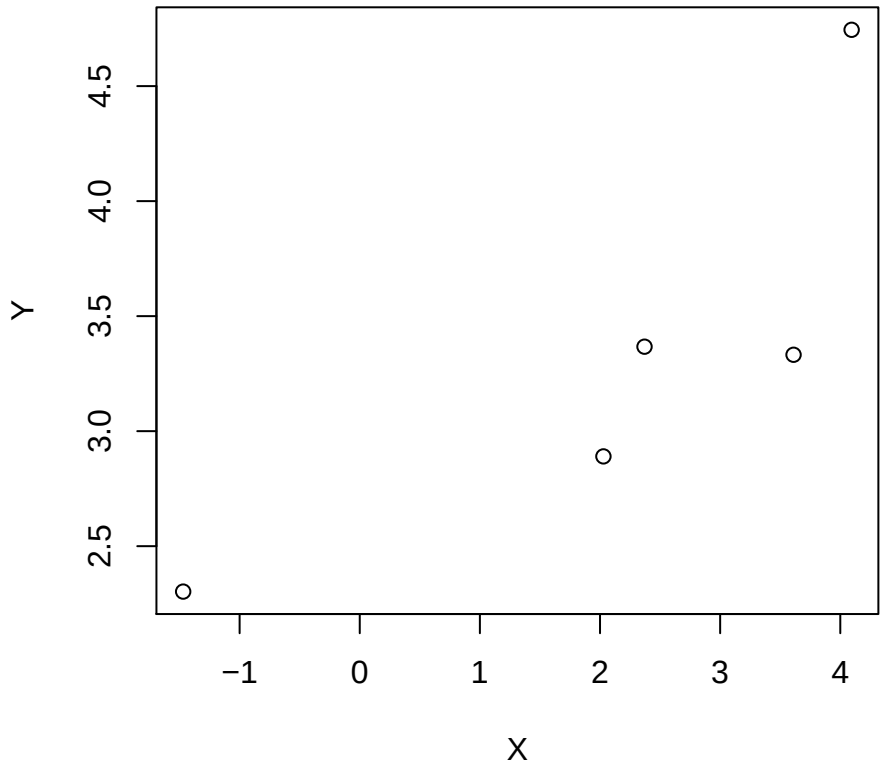


Figure 3.2: A plot of our raw data.


```
> plot(X, Y)   # same as plot(Y ~ X)
```


```
> summary(lm(Y ~ X -1))

Call:
lm(formula = Y ~ X - 1)

Residuals:
   Homo   Pongo  Macaca  Ateles  Galago
 0.6285 -0.2982  0.9843  0.8513  3.7802
```

```
Coefficients:
  Estimate Std. Error t value Pr(>|t|)
X   1.0054     0.3142     3.2   0.0329 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.029 on 4 degrees of freedom
Multiple R-squared: 0.719,        Adjusted R-squared: 0.6488
F-statistic: 10.24 on 1 and 4 DF,  p-value: 0.03291
```

So the regression of Y on X, not taking account of phylogeny is significant. Let's look at the phylogeny, the data, and the independent contrasts (Fig. 3.3). First open a new quartz device so we can keep more than one plot at a time. And let's add a title so we know what it is.

```
> quartz()
> plot(pic.X, pic.Y)
> title("Independent Contrast Plot of Primate Data")
```

So we see that the variation in the data is perfectly aligned with phylogenetic distance. When we take account of the expected covariane due to phylogeny, we get very different PIC values from the raw data, and we lose the statistical association. So now we have a better feeling for what is going on in this example.

3.2.4 The Brownian Motion Model of Evolution

The Brownian motion (hereafter BM) process was the first model of evolution applied to comparative data Felsenstein (1985). It is a very simple stochastic model for continuous data (data which can take on any value fractional value, such as size, or mass or metabolic rate). BM assumes that at any point in time, the trait has some probability of increasing or decreasing in value (the probability is from a normal distribution, so there is equal probability of going up or down).

Written as a stochastic differential equation,

$$dX(t) = \sigma\, dB(t). \tag{3.1}$$

Eq. 3.1 expresses the amount of change in character X over the course of a small increment in time: specifically, $dX(t)$ is the infinitesimal change in the character X over the infinitesimal interval from time t to time $t + dt$. The term $dB(t)$ is "white noise"; that is, the random variables $dB(t)$ are independent and identically-distributed normal random

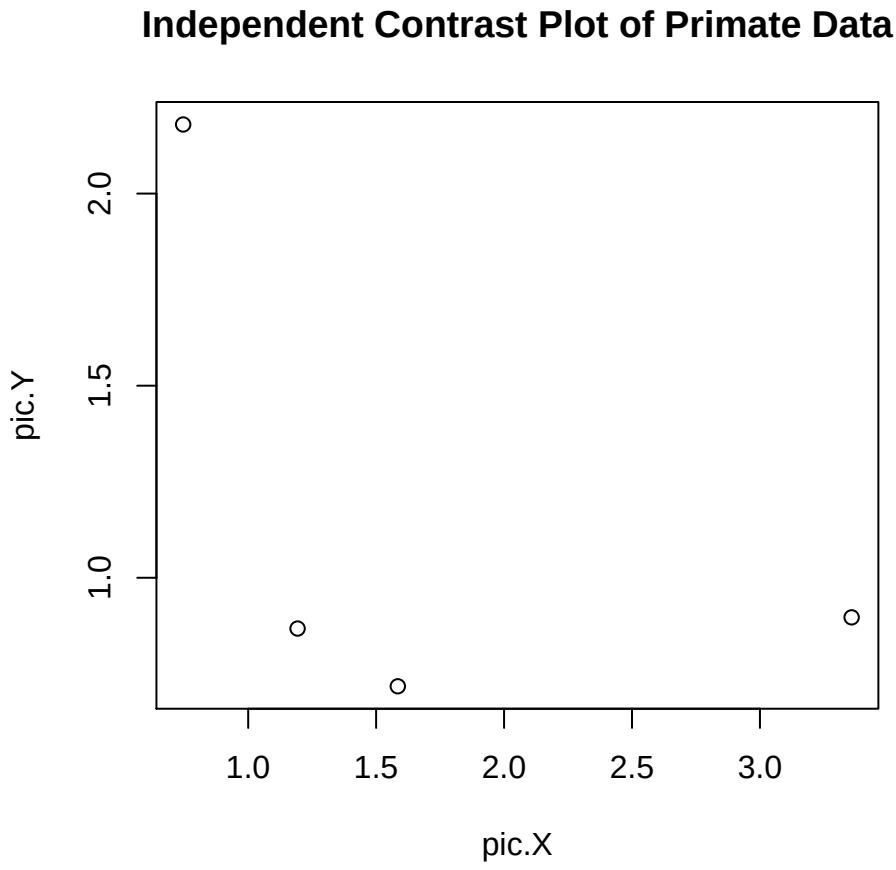


Figure 3.3: A plot of the independent contrasts on X, Y with phylogeny tree.primate.

variables, each with mean zero and variance proportional to σ. The parameter σ thus measures the intensity of the random fluctuations in the evolutionary process.

Applied to a phylogeny, the species are expected to covary in proportion to the amount of time they share in evolutionary history. That is, they have only been evolving independently since they diverged from their most recent common ancestor. It is this covariance that methods such as independent contrasts, phylogenetic GLS, and other methods which assume BM seek to correct for.

3.2.5 Phylogenetic GLS

A closely related approach to independent contrasts is the phylogenetic Generalized Least Squares technique. If we view the phylogenetic data as data containing "correlated errors". In statistical terms, this means that there are correlations among the observations, in this case as a result of shared phylogenetic history. We can transform this data (getting rid of the "correlated errors" by the phylogenetic GLS transformation:

$$\begin{aligned} \mathbf{Z} &= \mathbf{G^{-1/2}Y} \\ \mathbf{U} &= \mathbf{G^{-1/2}X} \end{aligned} \tag{3.2}$$

Where X, Y are the original data, G is the correlation matrix resulting from phylogenetic relationship, and Z and U are the transformed data. Using BM, the G matrix is simply the amount of time from the root of the phylogeny to the mrca of the pair of taxa (t_{bm}):

$$\mathbf{G} = \mathbf{t_{bm}} \tag{3.3}$$

It is important to note that in this case, if you are doing a regression, you must include the intercept term in the phylogenetic transformation. That is because we did not mean-center the data to begin with.

Recall that a regression is of the form:

$$y = mx + b \tag{3.4}$$

Where y is the dependent variable, x is the independent variable, with parameters for slope m and intercept b. A regression procedure takes the data (x and y), and finds the best filling values for m and b. Thus, we are *estimating* these parameters.

$$y = m * x + b * 1 \tag{3.5}$$

When we apply "phylogenetic corrections" on y and x, we have to remember that we must also apply it to the 1 next to the intercept term, b. It is maybe more clear if we

think about it as x being the coefficient of m, then the 1 is the coefficient of the intercept b.

In any case, when we apply phylogenetic correction, we must apply it to entire relationship, which includes both the slope and intercept.

Later, when we take a linear model of the data, we will exclude the intercept term because we've essentially bundled the intercept with the X to compute the phylogenetic correction, so we shouldn't double the intercept (more on this later).

In order to calculate this using ape codes, we first compute the correlation matrix assuming Brownian motion:

```
> tree <- tree.primates
> bm.prim <- corBrownian(phy=tree)
```

We then take this and use a standard statistical technique called generalized least squares, in which you can specify a matrix of correlated errors (in other words, it doesn't assume that correlations among observations are zero):

```
> XY <- data.frame(Y, X)
> summary(gls(Y ~ X, correlation=corBrownian(phy=tree), data=XY))

Generalized least squares fit by REML
  Model: Y ~ X
  Data: XY
       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
X           0.4319328 0.2864904 1.507669  0.2288

 Correlation:
  (Intr)
X -0.437

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
```

We can see that the p-values (and parameter estimates) are the same using either phylogenetic GLS or independent contrasts.

Note that we could also have written the model as `Y ~ X` without specifying the dataframe in the `gls()` call, but we would get a warning that the rownames of the dataframe don't match the tree. This is because this code was written to expect a data frame and not a vector (vectors don't have "rownames" or "columnames" because they have only one dimension. Instead, they only have "names" for each vector element.)

Other correlation structures can be specified (see help documentation for explanation of the parameters). `corGrafen` is a scaled Brownian motion, whereas `corMartins` specifies an OU model with a global optimum:

```
> corGrafen(value, tree, fixed=FALSE)
> corMartins(value, tree, fixed=FALSE)
```

3.2.6 Ancestral Reconstruction Methods

`ape` also has a function for reconstructing ancestral states called `ace`. Currently, there are two main models that `ace` uses to do the reconstruction, "ML" for Maximum Likelihood, and "pic" for Phylogenetically Independent Contrasts. Note that both use a Brownian motion model, but the statistical method to fit differs. "pic" is using a least-squares fitting method, whereas "ML" is using likelihood.

Using our primate data from above, try:

```
> ancstatesML <- ace(X, tree, type="continuous")
> ancstatesPIC <- ace(X, tree, type="continuous", method="pic")
> ancstatesML

$loglik
[1] -6.714469
```

```
$ace
       6        7        8        9
1.183725 2.192018 2.571320 3.503182

$sigma2
[1] 1.9711502 0.6970463

$CI95
           [,1]     [,2]
[1,] -0.5058591 2.873308
[2,]  0.9868737 3.397163
[3,]  1.4844055 3.658235
[4,]  2.6858445 4.320519

$call
ace(x = X, phy = tree, type = "continuous")

attr(,"class")
[1] "ace"
```

```
> ancstatesPIC
```

```
$ace
       6        7        8        9
1.183725 2.780824 3.200378 3.852630

$CI95
          [,1]     [,2]
[1,] -1.296931 3.664381
[2,]  0.854866 4.706781
[3,]  1.367000 5.033757
[4,]  2.582428 5.122832

$call
ace(x = X, phy = tree, type = "continuous", method = "pic")

attr(,"class")
[1] "ace"
```

For continouous data, `ace` returns a list with elements:

ace the estimates of the ancestral character values.

CI95 the estimated 95% confidence intervals.

sigma2 if model = "BM", and method = "ML", the maximum likelihood estimate of the Brownian parameter.

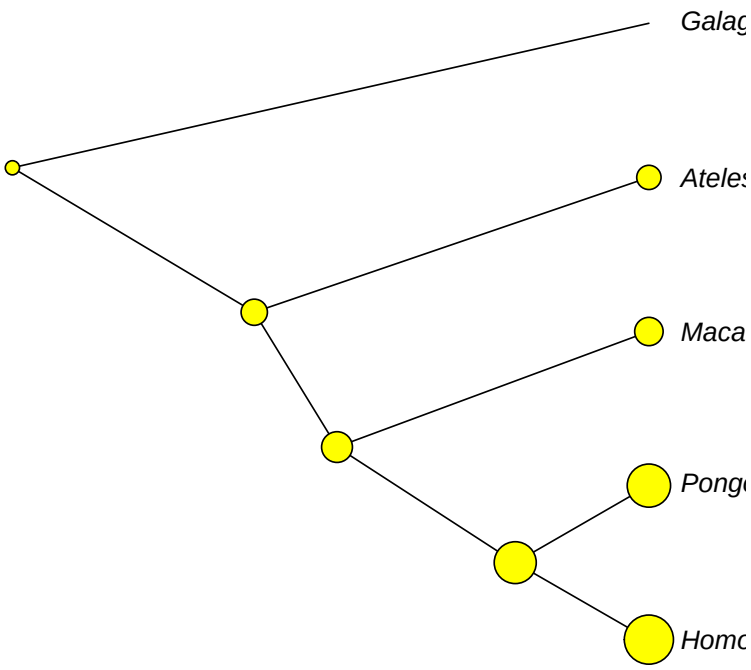
loglik if method = "ML", the maximum log-likelihood.

call the function call.

NOTE: In general, the 95% CI for ancestral states is very large, and increase as you go further back in time. There is simply less information way back near the root to draw any strong statistical conclusions.

Let's try to visualize the ancestral states on the tree. We will use two handy functions from `ape`: `tiplabels()` and `nodelabels()`.

```
> plot(tree, type="cladogram", label.offset=.05)
> tiplabels(pch=21, cex= X, bg="yellow")
> nodelabels(pch=21, cex= ancstatesML$ace, bg="yellow")
```



The "label.offset" parameters simply plots the species names a little bit away from the phylogeny. For other plots or plotting styles, you'll just have to play with this a little to get it right. tiplabels and nodelabels are plotting graphical symbols on the phylogeny, pch=21 designates a two-tone plotting symbol and we set the background or internal color to yellow using bg="yellow". You can also set the outside color of this point by setting col="red". The information is in the size of the symbol, which is set by the cex parameter. Luckily, our character values were in a nice range for plotting (roughly 1 to 5), if the numeric values were not so, you would simply scale them by multiplying by a constant (e.g., for doubling the size, cex=X*2 and cex=ancstatesML$ace*2). If you think having the branch length info is ugly, you can turn this off by using:

```
> plot(tree, type="cladogram", use.edge.length=FALSE, label.offset=.05)
```

Let's save the tree and data in an Rdata file in the "Rdata" folder:

```
> save(tree.primates, X, Y, file="Rdata/tree.primates.rda")
```

HINT: Make a new folder for each R project/analysis, and keep them tidy

Chapter 4

Saving your work as R scripts

Chapter Topics:

- Building good scripts
- Running source code
- Debugging scripts
- Clearing your workspace

Skills: writing clean source code, verifying, using `print` and `cat`, using history files.

Because R is interactive, it is tempting to simply play with code until you get the results you want. The problem with this is that you may not be able to reproduce it. Also, you may have made many manipulations of your data, some of which you've lost track of, and so your data objects may not really be what you think they are. This makes it impossible to double-check your analysis.

A key part of any analysis is verification:

1. Did you do what you really think you did?
2. Was the input free of error?
3. Did the steps of your analysis work without error?

You can save a complete record of your work from input to output in a script (`.R`) file. The mac interface has a very nice text editor. From the R menu, choose File > New Document (or command-N). Simply type or cut and paste your code into here. Let's make a script for the analyses we've done thus far.

First, make sure that you are in the directory that you want the script to execute from (`Rcomparative`). Start off with any packages that you wish to load, then begin to cut

and paste your code. Make sure to add comments indicated by the # symbol so that you know what the code does:

```
> require(ape)
> tree.primates <- read.tree(text="((((Homo:0.21,Pongo:0.21):0.28,Macaca:0.49):
+ 0.13,Ateles:0.62):0.38,Galago:1.00);")      # this is our primate tree
> summary(tree.primates)
> pdf(file="primatetree.pdf")      # turn on pdf device for output
> plot(tree.primates)            # plot the tree to the pdf file
> dev.off()               # turn off pdf device to return output to the default
> save(tree.primates, file="Rdata/tree.primates.rda")  # save tree in Rdata format
> ###
> # please insert your other lines of code here
> ###
>
> plot(tree, type="cladogram", label.offset=.05)     # make fancy plot of tree
> tiplabels(pch=21, cex= X, bg="yellow")           # add yellow balls at tips
> nodelabels(pch=21, cex= ancstatesML$ace, bg="yellow")  # add balls at nodes
```

Save the script file as "`simpleCA.R`" in your `Rcomparative` folder. Now if you want to run the code, you simply type at the R console (from within your Rcomparative directory):

```
> source("simpleCA.R")
```

When I am trying to develop a script, I often work by having the script window open next to the R console, and once a bit of code is working, I cut and paste it directly into the script. Save the script and source it. Once you have a good amount of code, you can work by making changes to the script, saving, and sourcing, over and over again.

Another handy feature of R is that it automatically saves a history file. That is, a file that has a list of every command you've executed in your sessions. It is saved by default as `.history` in your working directory. Because the file name begins with a period, it is not visible normally (although it is there). To save it explicitly with your own filename, either click on the history button on the R gui (box with yellow and blue lines), and click on "save history" at the bottom of the side window, or type the code:

```
> savehistory(file = "date_today.Rhistory")
```

This is an ordinary text file, which you can open up and edit (removing all the mistakes), and save as a `scriptname.R` file.

Another helpful tip when writing source code is to use `print` and `cat` functions to print out your output to the console. When you are using R in interactive mode, when you

type the name of a variable, you get a print of its contents. However, when you source the same code, the variable does not print to the screen. You have to explicitly put a `print` or `cat` function around it. Make a test script file and save it as `test.R`:

```
> require(ape)
> tree.primates <- read.tree(text="((((Homo:0.21,Pongo:0.21):0.28,Macaca:0.49):
+ 0.13,Ateles:0.62):0.38,Galago:1.00);")
> print('Species labels')

[1] "Species labels"

> print(tree.primates$tip.label)

[1] "Homo"   "Pongo"  "Macaca" "Ateles" "Galago"

> cat("Species labels =", tree.primates$tip.label, sep=" ")

Species labels = Homo Pongo Macaca Ateles Galago

> summary(tree.primates)

Phylogenetic tree: tree.primates

  Number of tips: 5
  Number of nodes: 4
  Branch lengths:
    mean: 0.415
    variance: 0.08208571
    distribution summary:
   Min. 1st Qu.  Median 3rd Qu.    Max.
 0.1300  0.2100  0.3300  0.5225  1.0000
  No root edge.
  Tip labels: Homo
              Pongo
              Macaca
              Ateles
              Galago
  No node labels.
```

You can see that `print` just makes a rough dump of the variables onto the screen. I added a character string so that we would know what variable was being printed to screen. `cat` makes a nicer, more customized display (it turns everything into a character vector, then pastes them together [i.e., concatenates them] before printing). They both do the same basic job, however. Notice also that `summary` does print to screen. Usually you only need to use these explicit print statements to see the contents of your variables as you are debugging.

Finally, remember that R is interactive, and the objects you create during a session are still around even after you've run your source code and forgotten about them. So to really check that your script is complete, you should shut down R (don't save the workspace), double click on the name of your script to restart R in the correct directory, and then source the program again. Does it work? Great!!

You could also try clearing all the objects from your workspace using the command:

```
> rm(list=ls())     # remove a list of objects consisting of the entire workspace
```

But this doesn't unload your packages, and there is still a danger that the script won't run in a fresh session. It's OK for minor incremental changes, but the best thing for a real test is to quit R and retry with a blank slate.

Try to create a script file for all the analyses we've done so far (and for every session throughout the course).

4.1 Exercises

1. Create a script of the work we've done so far.
2. Plot your reconstructed ancestral states.
3. Save output to a file.
4. Explore other datasets in R. At the R command prompt type `data()` to see what is available, especially those for other phylogeny programs.

Chapter 5

The Workhorse Functions of Data Manipulation

Chapter Topics/Skills:

Indexing/Subsetting accessing particular elements of your dataframe

String Matching using grep, sub

Sorting ordering data

Matching using logical comparisons to index

Merging matching two data frames or matrices by a common column and merging into a new object

Reshaping R Objects changing the shape of matrices and dataframes, long-thin to short-fat formats

Attributes, Classes the characteristics of data objects and how to manipulate them

As a biologist, these data manipulation topics may seem dry, but they are really powerful and will allow you do to much more sophisticated analyses, and to do them with confidence. So it is well worth taking some time to learn how to use them well.

5.1 Indexing and subsetting

R has powerful database functionality. Subsetting (picking particular observations out of an R object) is something that you will have to do all the time.

Let's work with a dataframe that is provided with the `phylobase` package called `geospiza_raw`. It contains five morphological measurements for 13 species. First, let's clear the workspace (or clear and start a new R session):

```
> rm(list=ls())
> require(phylobase)
> data(geospiza_raw)   # load the dataset into the workspace
> ls()               # list the objects in the workspace
```

```
[1] "geospiza_raw"
```

The object was named `geospiza_raw`. Let's find out some basic information about this object:

```
> class(geospiza_raw)
```

```
[1] "list"
```

```
> attributes(geospiza_raw)
```

```
$names
[1] "tree" "data"
```

It is a list with two elements. Here we want the data

```
> geo <- geospiza_raw$data
> dim(geo)
```

```
[1] 13  5
```

It is a dataframe with 13 rows and 5 columns. If we want to know all the attributes of geo:

```
> attributes(geo)
```

```
$names
[1] "wingL"   "tarsusL" "culmenL" "beakD"   "gonysW"

$row.names
 [1] "fuliginosa"   "fortis"       "magnirostris" "conirostris"  "scandens"
```

```
 [6] "difficilis"   "pallida"      "parvulus"     "psittacula"   "pauper"
[11] "Platyspiza"   "fusca"        "Pinaroloxias"

$class
[1] "data.frame"
```

We see that it has a "names" attribute, which refers to column names in a dataframe. Typically, the columns of a dataframe are the variables in the dataset. It also has "rownames" which contains the species names (so it does not have a separate column for species names).

In general, accessing elements of vectors, matrices, or dataframes is achieved through *indexing* by:

inclusion a vector of positive integers indicating which elements of the vector to include

exclusion a vector of negative integers

logical values a vector of TRUE / FALSE values indicating which elements to include / exclude

by name a character vector of names of columns (only) or columns and rows

blank index take the entire column, row, or object

Dataframes have two dimensions which we can use to index with: dataframe[row, column].

```
> geo     # the entire object, same as geo[] or geo[,]
> geo[c(1, 3), ]   # select the first and third rows, all columns
> geo[, 3:5]   # all rows, third through fifth columns
> geo[1, 5]  # first row, fifth column (a single number)
> geo[1:2, c(3, 1)]  # first and second row, third and first column (2x2 matrix)
> geo[-c(1:3, 10:13), ]  # everything but the first three and last three rows
> geo[ 1:3, 5:1]  # first three species, but variables in reverse order
```

Indexing a data frame by a single vector (meaning, no comma separating) selects an entire column. This can be done by name or by number:

```
> geo[3]   # third column
> geo["culmenL"]  # same
> geo[c(3,5)]  # third and fifth column
> geo[c("culmenL", "gonysW")]  # same
```

We can also use the names (or rownames) attribute if we are lazy. Suppose we wanted all the species which began with "pa". we could find which position they hold in the dataframe by looking at the rownames, saving them to a vector, and then indexing by them:

```
> sp <- rownames(geo)
> sp                              # a vector of the species names

 [1] "fuliginosa"   "fortis"       "magnirostris" "conirostris"  "scandens"
 [6] "difficilis"   "pallida"      "parvulus"     "psittacula"   "pauper"
[11] "Platyspiza"   "fusca"        "Pinaroloxias"

> sp[c(7,8,10)]     # the ones we want are #7,8, and 10

[1] "pallida"  "parvulus" "pauper"

> geo[ sp[c(7,8,10)], ]  # rows 7,8 and 10, same as geo[c(7, 8, 10)]

            wingL tarsusL culmenL   beakD   gonysW
pallida  4.265425 3.08945 2.43025 2.01635 1.949125
parvulus 4.131600 2.97306 1.97442 1.87354 1.813340
pauper   4.232500 3.03590 2.18700 2.07340 1.962100
```

An equivalent way to index is by using the `subset` function. Some people prefer it because you have explicit parameters for what to select and which variables to include. See help page `?subset`.

5.2 String Matching

A more useful feature is string matching. R has grep facilities, which can do partial matching of character strings. For example, we could directly search for species (the object or "x") names which contain "p" (the pattern):

```
> sp <- rownames(geo)
> grep(pattern = "p", x = sp)  # returns indices

[1]  7  8  9 10 11

> grep("p", sp, value=T)  # returns the species names which match
```

```
[1] "pallida"    "parvulus"   "psittacula" "pauper"     "Platyspiza"
```

```
> grep("p", sp, ignore.case=T, value=T)   # case-sensitive by default
```

```
[1] "pallida"      "parvulus"     "psittacula"   "pauper"       "Platyspiza"
[6] "Pinaroloxias"
```

```
> grep("^P", sp, value=T)  # only those which start with (^) capital P
```

```
[1] "Platyspiza"   "Pinaroloxias"
```

It is possible to use perl-type regular expressions, and the sub function is also available. Sub is related to grep, but substitutes a replacement value to the matched pattern. Notice that there are two species which have upper case letters. We can fix this with:

```
> sp <- rownames(geo)
> sub(pattern = "^P", replacement = "p", sp)
```

```
 [1] "fuliginosa"   "fortis"       "magnirostris" "conirostris"  "scandens"
 [6] "difficilis"   "pallida"      "parvulus"     "psittacula"   "pauper"
[11] "platyspiza"   "fusca"        "pinaroloxias"
```

```
> rownames(geo) <- sub(pattern = "^P", replacement = "p", sp)   # to save changes
```

5.3 Ordering Data

Suppose we now want geo in alphabetical order. We can use the `sort` function to sort the rownames vector, then use it to index the dataframe:

```
> sort(rownames(geo))
> geo[ sort(rownames(geo)), ]
```

A better option for dataframes, though, is `order`:

```
> order(rownames(geo))   # the order that the species should take to be
```

```
 [1]  4  6  2  1 12  3  7  8 10 13 11  9  5
```

```
>                   #  sorted from a-z
> rbind(rownames(geo), order(rownames(geo)))  # to illustrate

     [,1]         [,2]     [,3]           [,4]          [,5]       [,6]
[1,] "fuliginosa" "fortis" "magnirostris" "conirostris" "scandens" "difficilis"
[2,] "4"          "6"      "2"            "1"           "12"       "3"
     [,7]      [,8]       [,9]         [,10]    [,11]        [,12]
[1,] "pallida" "parvulus" "psittacula" "pauper" "platyspiza" "fusca"
[2,] "7"       "8"        "10"         "13"     "11"         "9"
     [,13]
[1,] "pinaroloxias"
[2,] "5"

> oo <- order(rownames(geo))
> geo[oo,]   # sorted in alpha order

               wingL  tarsusL  culmenL    beakD   gonysW
conirostris  4.349867 2.984200 2.654400 2.513800 2.360167
difficilis   4.224067 2.898917 2.277183 2.011100 1.929983
fortis       4.244008 2.894717 2.407025 2.362658 2.221867
fuliginosa   4.132957 2.806514 2.094971 1.941157 1.845379
fusca        3.975393 2.936536 2.051843 1.191264 1.401186
magnirostris 4.404200 3.038950 2.724667 2.823767 2.675983
pallida      4.265425 3.089450 2.430250 2.016350 1.949125
parvulus     4.131600 2.973060 1.974420 1.873540 1.813340
pauper       4.232500 3.035900 2.187000 2.073400 1.962100
pinaroloxias 4.188600 2.980200 2.311100 1.547500 1.630100
platyspiza   4.419686 3.270543 2.331471 2.347471 2.282443
psittacula   4.235020 3.049120 2.259640 2.230040 2.073940
scandens     4.261222 2.929033 2.621789 2.144700 2.036944
```

Order can sort on multiple arguments, which means that you can use other columns to break ties. Let's trim the species names to the first letter using the substring function, then sort using the first letter of the species name and breaking ties by tarsusL:

```
> sp <- substring(rownames(geo), first=1, last=1)
> oo <- order(sp , geo$tarsusL) # order by first letter species, then tarsusL
> geot <- geo[oo,]["tarsusL"]   # ordered geo dataframe, take only the wingL column
> geo <- geo[oo,]
```

Note: using `geo["tarsusL"]` as a second index for order doesn't work, because it is a one column dataframe, as opposed to geo$tarsus which is a vector. It must match sp, which is a vector. Check the `dim` and `length` of each. vectors have length only, dataframes have dimension 2.

5.4 Matching

Matching is very easy in R, and is often used to create a logical vector to subset objects. Greater than and less than are as usual, but logical equal is "==" to differentiate from the assignment operator. Also >= and <=.

```
> geot > 3    # a logical index
```

```
             tarsusL
conirostris    FALSE
difficilis     FALSE
fuliginosa     FALSE
fortis         FALSE
fusca          FALSE
magnirostris    TRUE
parvulus       FALSE
pinaroloxias   FALSE
pauper          TRUE
psittacula      TRUE
pallida         TRUE
platyspiza      TRUE
scandens       FALSE
```

```
> geot == 3  # must match exactly 3, none do
```

```
             tarsusL
conirostris    FALSE
difficilis     FALSE
fuliginosa     FALSE
fortis         FALSE
fusca          FALSE
magnirostris   FALSE
parvulus       FALSE
pinaroloxias   FALSE
pauper         FALSE
psittacula     FALSE
pallida        FALSE
platyspiza     FALSE
scandens       FALSE
```

```
> geot[ geot > 3 ]   # use to get observations which have tarsus > 3
```

```
[1] 3.038950 3.035900 3.049120 3.089450 3.270543
```

```
> #  ii <- geot > 3    # these two lines of code accomplish the same
> #  geot[ii]
> cbind(geo["tarsusL"], geot > 3)  # check
```

```
              tarsusL tarsusL
conirostris  2.984200   FALSE
difficilis   2.898917   FALSE
fuliginosa   2.806514   FALSE
fortis       2.894717   FALSE
fusca        2.936536   FALSE
magnirostris 3.038950    TRUE
parvulus     2.973060   FALSE
pinaroloxias 2.980200   FALSE
pauper       3.035900    TRUE
psittacula   3.049120    TRUE
pallida      3.089450    TRUE
platyspiza   3.270543    TRUE
scandens     2.929033   FALSE
```

```
> geo[geot>3, ]["tarsusL"]  # what does this do?
```

```
              tarsusL
magnirostris 3.038950
pauper       3.035900
psittacula   3.049120
pallida      3.089450
platyspiza   3.270543
```

Matching and subsetting works really well for replacing values. Suppose we thought that every measurement that was less than 2.0 was actually a mistake. We can remove them from the data:

```
> geo [ geo<2 ] <- NA
```

Missing values compared to anything else will return a missing value (so NA == NA returns NA, which is usually not what you want). You must test it with `is.na` function. You can also test multiple conditions with and (&) and or (|)

```
> !is.na(geo$gonysW)
```

```
 [1]  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE
[13]  TRUE
```

```
> geo[!is.na(geo$gonysW) & geo$wingL > 4, ]  # element by element "and"
```

```
               wingL  tarsusL  culmenL    beakD   gonysW
conirostris  4.349867 2.984200 2.654400 2.513800 2.360167
fortis       4.244008 2.894717 2.407025 2.362658 2.221867
magnirostris 4.404200 3.038950 2.724667 2.823767 2.675983
psittacula   4.235020 3.049120 2.259640 2.230040 2.073940
platyspiza   4.419686 3.270543 2.331471 2.347471 2.282443
scandens     4.261222 2.929033 2.621789 2.144700 2.036944
```

```
> geo[!is.na(geo$gonysW) | geo$wingL > 4, ]   # element by element "or"
```

```
               wingL  tarsusL  culmenL    beakD   gonysW
conirostris  4.349867 2.984200 2.654400 2.513800 2.360167
difficilis   4.224067 2.898917 2.277183 2.011100       NA
fuliginosa   4.132957 2.806514 2.094971       NA       NA
fortis       4.244008 2.894717 2.407025 2.362658 2.221867
magnirostris 4.404200 3.038950 2.724667 2.823767 2.675983
parvulus     4.131600 2.973060       NA       NA       NA
pinaroloxias 4.188600 2.980200 2.311100       NA       NA
pauper       4.232500 3.035900 2.187000 2.073400       NA
psittacula   4.235020 3.049120 2.259640 2.230040 2.073940
pallida      4.265425 3.089450 2.430250 2.016350       NA
platyspiza   4.419686 3.270543 2.331471 2.347471 2.282443
scandens     4.261222 2.929033 2.621789 2.144700 2.036944
```

```
> !is.na(geo$gonysW) && geo$wingL > 4   # vectorwise "and"
```

```
[1] TRUE
```

Matching works on strings also:

```
> geo[rownames(geo) == "pauper",]   # same as   geo["pauper", ]
> geo[rownames(geo) < "pauper",]
```

There are even better functions for strings, though. In the expression `A %in% B`, the `%in%` operator compares two vectors of strings, and tells us which elements of `A` are present in `B`.

```
> newsp <- c("clarkii", "pauper", "garmani")
> newsp[newsp  %in% rownames(geo)]     # which new species are in geo?
```

We can define the "without" operator:

```
> "%w/o%" <- function(x, y) x[!x %in% y]
> newsp  %w/o% rownames(geo)   # which new species are not in geo?
```

5.5 Merging

Merging is another powerful database function. The concept is simple. Given two objects with a common matching key, can we merge them together into one object? Usually, the matching key in comparative data is the species name.

A common task is to match a morphology dataset with an ecology dataset, or a tree file with a data file. Continuing our example, let's make an ecology field and add it to geot:

```
> geot$ecology <- LETTERS[1:nrow(geot)]   # A:M
```

Now, let's merge geo["tarsusL"] with the first five rows of geot:

```
>                     # only maches to both datasets are included
> merge(x=geo["tarsusL"], y=geot[1:5, ], by= "row.names")
```

```
    Row.names tarsusL.x tarsusL.y ecology
1 conirostris  2.984200  2.984200       A
2  difficilis  2.898917  2.898917       B
3      fortis  2.894717  2.894717       D
4  fuliginosa  2.806514  2.806514       C
5       fusca  2.936536  2.936536       E
```

```
>                     # all species in both datasets are included
> merge(x=geo["tarsusL"], y=geot[1:5,], by= "row.names", all=T)
```

```
      Row.names tarsusL.x tarsusL.y ecology
1   conirostris  2.984200  2.984200       A
2    difficilis  2.898917  2.898917       B
3        fortis  2.894717  2.894717       D
4    fuliginosa  2.806514  2.806514       C
5         fusca  2.936536  2.936536       E
```

```
6  magnirostris  3.038950        NA    <NA>
7        pallida  3.089450        NA    <NA>
8       parvulus  2.973060        NA    <NA>
9         pauper  3.035900        NA    <NA>
10 pinaroloxias  2.980200        NA    <NA>
11   platyspiza  3.270543        NA    <NA>
12   psittacula  3.049120        NA    <NA>
13     scandens  2.929033        NA    <NA>
```

The results of merge are sorted by default on the sort key. To turn it off:

```
> geo <- geo[rev(rownames(geo)), ]   # reverse the species order of geo
>                       # merge on geo first, then geot
> merge(x=geo["tarsusL"], y=geot[1:5, ], by= "row.names", sort=F)

    Row.names tarsusL.x tarsusL.y ecology
1       fusca  2.936536  2.936536       E
2      fortis  2.894717  2.894717       D
3  fuliginosa  2.806514  2.806514       C
4  difficilis  2.898917  2.898917       B
5 conirostris  2.984200  2.984200       A

>                       # geot first, then geo
> merge(x=geot[1:5,], y=geo["tarsusL"], by= "row.names", sort=F)

    Row.names tarsusL.x ecology tarsusL.y
1 conirostris  2.984200       A  2.984200
2  difficilis  2.898917       B  2.898917
3  fuliginosa  2.806514       C  2.806514
4      fortis  2.894717       D  2.894717
5       fusca  2.936536       E  2.936536
```

5.6 Reshaping R Objects

Internally, R objects are stored as one huge vector. The various shapes of objects are simply created by R knowing where to break the vector into rows and columns. So it is very easy to reshape matrices:

```
> vv <- 1:10  # a vector
> mm <- matrix( vv, nrow=2)  # a matrix
> mm
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

```
> dim(mm) <- NULL
> mm <- matrix( vv, nrow=2, byrow=T)  # a matrix, but cells are now filled by row
> mm
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
```

```
> dim(mm) <- NULL
> mm  # vector is now n a different order because the collapse occurred by column
```

```
 [1]  1  6  2  7  3  8  4  9  5 10
```

Other means of "collapsing" dataframes are:

```
> unlist(geo)   # produces a vector from the dataframe
>             # the atomic type of a dataframe is a list
> unclass(geo)  # removes the class attribute, turning the dataframe into a
>             # series of vectors  plus any names attributes, same as setting
>             # class(geo) <- NULL
> c(geo)  # similar to unclass but without the attributes
```

Chapter 6

All About Data

Goals:

- Handling raw data and preprocessing to R data files.
- Programming:

  Identify what do you want to do? (you can't do it if you can't articulate it exactly)

  Am I sure the raw data is free of error? (check data entry)

  Do I need to reshape my data? (check data processing)

  Am I sure that all of the programming steps are doing what I want them to do and free of error?

  Do the results make sense? Do I trust it?

Concepts:

- Reading in external files
- Saving r data files
- Flowcharting

6.1 Raw data to "curated" data

An important part of data analysis is checking for accuracy in your data transfer from your field notebook to the raw data files, to your processed data files. Here are some best practices:

- Set up a "data" and an "Rdata" directory for your raw data files and processed data files, respectively.

- Raw data is data that is the most original source that is on the computer. Whether it is manually entered it, or obtained from the literature. It is usually a text file or a spreadsheet. The most convenient format for R is ".csv" or comma separated format, but R can handle any delimiter or fixed-width file. Excel files should be saved as .csv if at all possible (it is possible to read in Excel, but it is a pain).

- Data entry is inherently error-prone. Keeping this in mind, take some steps to make it as easy as possible to enter accurately. Organize your field notebook in rows and columns, then type the data into your file exactly in the same order as in your notebook.

- It is important to check raw data for accurate data entry. Print out your text file, and check against your notebook line by line and number by number.

- Raw data should ONLY be corrected for typos on data entry. Numbers should not be altered because they are "outliers". Filling in missing data is not recommended.

- Protect the sanctity of your raw data. Keep your raw data files pristine and unaltered. Any changes, fixes, or cleaning up that you should be done in the analysis (i.e., via your R script), and not to your raw data file itself. If you permanently change your raw data file, and later have questions, you will never be able to reconstruct what you did. If it is just too difficult to fix the data with code, at least keep a copy of your original data file, and save a version as "edited" and keep notes on what was modified.

- Save your cleaned up, reogranized data as an R data file (.rda or .Rdata) in your Rdata folder.

- Save a script of all your data processing and analysis. If you send someone this script and the raw data file, they should be able to run your code.

- As with all programming projects, plan your steps from input (raw data and what shape is it in) to output ("curated data" and what shape do you want for our analyses).

R has great database (merging and matching), string manipulation, sorting, and data reshaping facilities. We'll illustrate some of these in a typical data "curation" step using a morphological dataset of *Anolis* lizards. We will start with the raw data, entered into the computer from a field notebook and save the "curated" product as an R data file.

6.1.1 Reading in fixed width format

Although `read.csv` is much easier, the data was intended for use with SAS and saved as a fixed-width format. R has a fixed-width reading function called `read.fwf` which requires as arguments the filename and a vector of widths. Note that if you have blank columns that you don't want read in, indicate the width of these with negative integers. For example, we are reading in the first 13 characters as the first variable, character 14 as the second variable, skipping character 15, etc.. So let's read in the files, assign names to the columns, and then check that the complete file was read in using the `head()` and `tail()` functions (which are especially useful for large datasets):

```
> read.fwf('data/94morphja.dat', widths=c(13, 1, -1, 5, 5, 5, -1, 1, 5, 5, 5,
+  3, -5, -5, -1, 1), as.is=T, strip.white=T) ->datja
> read.fwf('data/94morphpr.dat', widths=c(13, 1, -1, 5, 5, 5, -1, 1, 5, 5, 5,
+  3, -5, -5, -1, 1), as.is=T, strip.white=T) ->datpr
> names(datpr) <- names(datja) <- c('species', 'sex', 'svl', 'mass', 'tail', 'regen',
+ 'forel', 'hindl', 'headl', 'lamn', 'food')
> head(datja)
```

```
     species sex   svl mass  tail regen forel hindl headl lamn food
1    garmani   m 103.0 23.5 151.0     r  41.0  71.0     .   33    n
2     sagrei   m  54.0  3.8  93.5     r  23.0  39.0  13.5   18    n
3 lineatopus   m  62.5    .  82.0     r  28.0  47.0  19.0   20    y
4     sagrei   m  51.0  3.9 104.0     .  22.0  36.0  14.0   17    y
5     sagrei   m  50.0  3.2  65.0     r  21.5  37.0  13.5   18    n
6     sagrei   m  45.5  3.1     .     .     .     .     .    .    .
```

```
> tail(datja)
```

```
       species sex svl mass  tail regen forel hindl headl lamn food
52     garmani   f  75 10.5 156.0     .  30.0  53.0  19.5   28    y
53 valencienni   m  61  3.6  77.0     .  21.0  30.5  16.5   24    .
54     grahami   f  46    .     .     .     .     .     .    .    y
55     grahami   m  69    .     .     .     .     .     .    .    y
56      sagrei   m  51    .     .     .     .     .     .    .    y
57      sagrei   m  48    .     .     .     .     .     .    .    n
```

```
> head(datpr)
```

```
     species sex  svl mass  tail regen forel hindl headl lamn food
1  stratulus   m 43.5  1.9  44.0     r  21.0  32.0  11.5   19    y
```

```
2  evermanni   m 65.5  5.7  81.0     r  31.0  49.0  18.0   28    n
3      krugi   j 49.0  1.2 105.0     .  16.0  32.0  10.5   18    .
4      krugi   m 48.0  2.6 126.0     .  21.0  39.0  14.0   17    .
5 pulchellus   m 42.0  1.6 110.0     .  17.5  31.5  12.3   18    .
6  stratulus   j 32.0  1.3  58.5     r  18.0  28.5  10.5   21    .
```

```
> tail(datja)
```

```
       species sex svl mass  tail regen forel hindl headl lamn food
52     garmani   f  75 10.5 156.0     .  30.0  53.0  19.5   28    y
53 valencienni   m  61  3.6  77.0     .  21.0  30.5  16.5   24    .
54     grahami   f  46    .     .     .     .     .     .    .    y
55     grahami   m  69    .     .     .     .     .     .    .    y
56      sagrei   m  51    .     .     .     .     .     .    .    y
57      sagrei   m  48    .     .     .     .     .     .    .    n
```

These files are not too big, so display the datasets and take a look to make sure that all the data are in the proper columns of the dataframe. The data were saved as separate files for each island, so we want to add in island to each dataset. If we like, we can add year, etc.

```
> datpr$island <- "Puerto Rico"
> datja$island <- "Jamaica"
```

6.1.2 Combining the data into one file

We want to have one merged file to work with. A very simple thing to do would be to simply "row-bind" the data frames using `rbind()`. However, a danger here is that if the columns of one file is not in the same order as the columns of another file, we will have the wrong columns stacked on top of one another. However, we can check that the columns are identical before doing a row bind operation.

```
> names(datja)
```

```
 [1] "species" "sex"     "svl"     "mass"    "tail"    "regen"
 [7] "forel"   "hindl"   "headl"   "lamn"    "food"    "island"
```

```
> names(datpr)
```

```
 [1] "species" "sex"     "svl"     "mass"    "tail"    "regen"
 [7] "forel"   "hindl"   "headl"   "lamn"    "food"    "island"
```

We can see that they look good. If we had a huge number of columns, though, trusting your eye is sometimes risky. You can check that the names vectors match element-by-element by doing this:

```
> names(datja) == names(datpr)
```

```
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Or even adding up all the cases where the elements DO NOT match. Logical values in arithmetic operations have value `TRUE=1`, `FALSE=0`.

```
> sum(names(datja) != names(datpr))
```

```
[1] 0
```

OK, now we're sure, so let's combine the datasets:

```
> rbind(datja, datpr) -> dat
```

6.1.3 Adding variables to the data

Caribbean *Anolis* lizards have evolved convergently into distinct microhabitat specialists termed "ecomorphs". So we would like to add in the ecomorph designation, which is missing from the data files. To do that, we can make use of some very nice subsetting and string matching functions.

The `%in%` operator is a matching function for vectors of character strings. It returns which of those character strings to the left match the list of character strings on the right.

First we need to create vectors for each of the ecomorphs, containing the species names which belong to the ecomorph:

```
> spp <- unique(dat$species)
> spp                          # list of species in our sample
```

```
 [1] "garmani"      "sagrei"       "lineatopus"   "grahami"
 [5] "valencienni"  "stratulus"    "evermanni"    "krugi"
 [9] "pulchellus"   "cristatellus" "gundlachi"    "occultus"
[13] "cuvieri"
```

```
> tgspp <- c("cristatellus", "gundlachi", "sagrei", "lineatopus")
> tcspp <- c("stratulus", "evermanni", "grahami")
> cgspp <- c("cuvieri", "garmani")
> gbspp <- c("krugi", "pulchellus")
> twspp <- c("occultus", "valencienni")
```

Make sure that all of the species have been assigned to one and only one of the ecomorph groups. First combine all of the ecomorph species groups, into a vector. Test for duplicates:

```
> ecospp <- c(tgspp, tcspp, cgspp, gbspp, twspp)
> length(ecospp)

[1] 13

> length(spp)

[1] 13

> ecospp == unique(ecospp)  # if any are duplicated or missing, will get some FALSE

 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Make a new function called "%w/o%" which tells us what is in string A that is NOT in string B (A without B). There should be no spp which are not in ecospp and vice versa.

```
> "%w/o%" <- function(x, y) x[!x %in% y]
> ecospp %w/o% spp

character(0)

> spp %w/o% ecospp

character(0)
```

Now we are sure that we assigned everything to one and only one group.

(If you want to see what happens when we make a mistake go back and change the "4" in the list for "tcspp" to a "3" and see what happens – don't forget to reverse it and rerun the code correctly though). Now we can create an index vector for where to fill the (newly created) "ecomorph" variable with the correct ecomorph by searching for those rows which belong to the correct species.

```
> tgi <- dat$species %in% tgspp
> tci <- dat$species %in% tcspp
> cgi <- dat$species %in% cgspp
> gbi <- dat$species %in% gbspp
> twi <- dat$species %in% twspp
```

Now use these ecomorph indicies to add a variable called "ecomorph" to the dataset.

```
> dat$ecomorph[twi] <- "twig"
```

The above code fills the variable "ecomorph" with "twig" if the species name is one of "valencienni" or "occultus". You can check by looking at the following (I've only printed the first 10 lines here to save paper). You should have a "TRUE" everywhere that the species is "valencienni" or "occultus", and FALSE for all others. And for each of these the ecomorph should be "twig".:

```
> cbind(twi, dat[, c('species', 'ecomorph')])
```

```
      twi      species ecomorph
151 FALSE cristatellus     <NA>
152 FALSE        krugi     <NA>
153 FALSE        krugi     <NA>
154 FALSE    stratulus     <NA>
155 FALSE        krugi     <NA>
156 FALSE cristatellus     <NA>
157 FALSE    stratulus     <NA>
```

Let's do the rest:

```
> dat$ecomorph[tgi] <- "trunk-ground"
> dat$ecomorph[tci] <- "trunk-crown"
> dat$ecomorph[cgi] <- "crown-giant"
> dat$ecomorph[gbi] <- "grass-bush"
```

Check that everything looks OK (make sure that there are no "NA"s in the ecomorph field, etc.

```
> head(dat)
```

```
     species sex  svl mass  tail regen forel hindl headl lamn food
1    garmani   m  103 23.5 151.0     r  41.0  71.0     .   33    n
2     sagrei   m   54  3.8  93.5     r  23.0  39.0  13.5   18    n
3 lineatopus   m 62.5    .  82.0     r  28.0  47.0  19.0   20    y
4     sagrei   m   51  3.9 104.0     .  22.0  36.0  14.0   17    y
5     sagrei   m   50  3.2  65.0     r  21.5  37.0  13.5   18    n
6     sagrei   m 45.5  3.1     .     .     .     .     .    .    .
   island     ecomorph
1 Jamaica  crown-giant
2 Jamaica trunk-ground
3 Jamaica trunk-ground
4 Jamaica trunk-ground
5 Jamaica trunk-ground
6 Jamaica trunk-ground
```

```
> head(dat$ecomorph)
```

```
[1] "crown-giant"  "trunk-ground" "trunk-ground" "trunk-ground"
[5] "trunk-ground" "trunk-ground"
```

6.1.4 Sort by species and sex

First we create and order index "o" using the function `order`. Then we reorder the data frame rows by "o" and resave:

```
> o <- order(dat$species, dat$sex)
> dat <- dat[o, ]
```

6.1.5 Editing data into R format

Notice that missing values in this dataset were encoded with a ".". This is not recognized by R, so we should replace that with "NA"s. In fact, we actually read in the whole dataframe as "character". The `apply` repeats the `mode` function across the columns of dat:

```
> apply(dat, 2, mode)
```

```
    species         sex         svl        mass        tail
"character" "character" "character" "character" "character"
      regen       forel       hindl       headl        lamn
```

```
"character" "character" "character" "character" "character"
       food      island    ecomorph
"character" "character" "character"
```

Now we need to replace the "." and convert the the appropriate columns to mode numeric.

```
> dat[dat=='.'] <- NA
```

Also, we want to use only adult males and females, and not juveniles, so exclude anything that is not sex "m" or "f":

```
> dat <- dat[dat$sex %in% c('m', 'f'), ]
```

The "`regen`" column is an indicator variable for regenerated tails, which are not a good indication of adult tail size. So we want to look at `regen`, and if it is "r", we want to remove those `tail` measurements, replacing them with NA:

```
> dat$tail[ dat$regen == "r" ] <- NA
```

Check that we did what we wanted to do (only printing a few rows on paper):

```
> dat[c("regen", "tail")]
```

```
    regen tail
72      r <NA>
96   <NA> 77.0
105     r <NA>
113  <NA> 84.0
114     r <NA>
115  <NA> 82.0
```

We want the numeric data to have mode numeric so let's use:

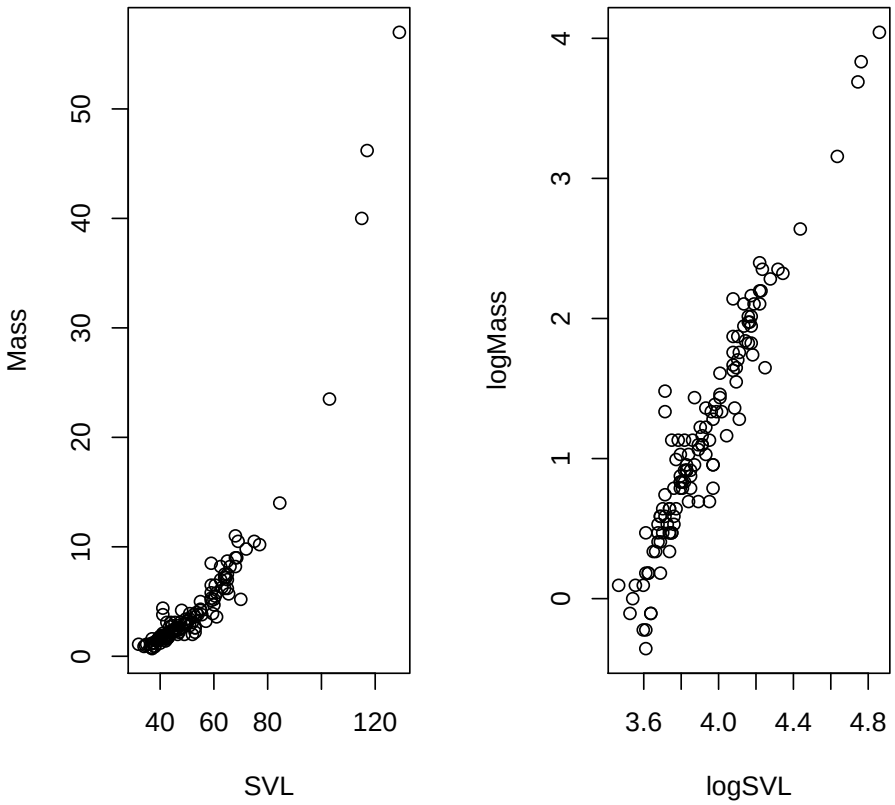
```
> dat.num <- dat[c('svl', 'mass', 'tail', 'forel', 'hindl', 'headl', 'lamn')]
> dat.num <- as.data.frame( apply(dat.num, 2, as.numeric) )
> apply(dat.num, 2, mode)
```

```
      svl      mass      tail     forel     hindl     headl      lamn
"numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
```

6.1.6 Getting statistics by species and sex

It is standard to use log-transformation in morphometrics. There are several reasons: (1) the data are usually not normally distributed (or, rather, the "errors" from any statistical model are not normally distributed), and log-transformation improves this. (2) Organisms tend to follow logistic growth curves, and species tend to follow non-allometric scaling patterns. Log transformation tends to linearize these non-linear trends, reducing the size of the very large and increase the size of the very small. (3) Some morphometric features scale multiplicatively with one another. For example, mass scales with the cube of length. Log-transformation makes these multiplicative relationships additive. Take a look at the plot of SVL versus MASS:

```
> op <- par(no.readonly = TRUE)
> par(mfrow=c(1,2))
> with(dat.num, plot(svl, mass, xlab="SVL", ylab="Mass"))
> with(dat.num, plot(log(svl), log(mass), xlab="logSVL", ylab="logMass"))
> par(op)
```



Let's log-transform the data, and then get the means, sample size, and standard deviations by species and sex. The `aggregate` function is an apply method, that is it operates on the matrix dat.num all at once, but it groups observations by some grouping variable first. In this case, species and sex.

```
> dat.mean <- aggregate( log(dat.num), list(species=dat$species, sex=dat$sex),
+ mean, na.rm=T)
> dat.sd <- aggregate( log(dat.num), list(species=dat$species, sex=dat$sex),
+ function(x) {if (all(is.na(x))) return(NA) else return(sd(x, na.rm=T))})
```

If we are interested in sexual dimorphism, then we will need to reshape our data. We want male and female observations to be matched by secies, so that we can take the ratio of male size to female size, for example.

```
> dd <- split(dat.mean, dat.mean$sex)
> fems <- dd[[1]]
> mals <- dd[[2]]
> sexes_sf <- merge(fems, mals, by="species", suffixes=c(".f", ".m"))
```

`sexes_sf` is a short-fat representation of the male and female data. We could then go through and subtract svl.m - svl.f, and so on, variable by variable. Alternatively, since R by default performs performs matrix operations element by element, and we know that the two matrices have exactly the same structure, we can simply subtract:

```
> sexdim <- mals[-(1:2)] - fems[-(1:2)]   # computes male larger dimorphism value
```

Limit to the five main variables.

Workarounds for broken code

- write work-around code and place it before every instance of the broken function
- `fix()`
- write a work-around function in the global environment by the same name

In R version 2.6.2, the `na.rm` option in `sd` used to return an `NA` if there were no observations in a particular group (since sd is not defined for zero observations). However, in 2.7.1, the behavior has changed and we get an error that halts execution. Go ahead and try:

```
> dat.sd <- aggregate(dat.num, list(species=dat$species, sex=dat$sex), sd, na.rm=T)
```

Now try:

```
> dat.sd <- aggregate(dat.num, list(species=dat$species, sex=dat$sex),
+ function(x) {if (all(is.na(x))) return(NA) else return(sd(x, na.rm=T))})
```

The function that I wrote above is a work-around. If all observations are missing, the function simply returns an NA without sending it to `sd`. This is an example of placing a fix right in front of the function call. Not very convenient if we need to use `sd` again. We could also use a handy function called `fix`, which allows us to edit code within a function, but it is only present for the current session, and disappears after.

```
> fix(sd)
```

When a source code window pops up, make the following edits, then close the window again:

Try the line of code should now work:

```
> dat.sd <- aggregate(dat.num, list(species=dat$species, sex=dat$sex), sd, na.rm=T)
```

Another option is to take the square root of the variance, and use a feature of the variance function to eliminate NA's:

```
> sd <- function (x, na.rm=FALSE)  sqrt(var(x,na.rm=na.rm,
+ use='pairwise.complete.obs'))
> dat.sd <- aggregate(dat.num, list(species=dat$species, sex=dat$sex),
+  sd, na.rm=T)
```

Whether you manually remove NA observations or use the square root and variance option, re-writing the `sd` function is probably the best option. It will reside in your global environment, so that will take precedence over the one in the `stats` package. And you can source this code (run it) without any manual inputs. `fix` is useful though, because it brings up a copy of the code you need to fix and allows you to make edits to it.

While `mean()` and `sd()` have options to remove NA's (although they may not work perfectly!), length does not. So we have to write a little function to do this manually before applying length.

```
> dat.N <- aggregate(dat.num, list(species=dat$species, sex=dat$sex),
+ function(x) { x<- x[!is.na(x)]; length(x)})
```

Or more compactly:

```
> dat.N <- aggregate(dat.num, list(species=dat$species, sex=dat$sex),
+  function(x) { length(x[!is.na(x)])})
```

Notice that with apply-like functions, which expect the name of a function to apply as a parameter, if you need to modify the input in any way or to do more than one step, you must write a little function.

Save all of our cleaned up data files in an external R dataset:

```
> save(dat, dat.mean, dat.sd, dat.N, file="Rdata/anolis_dat.rda")
```

Chapter 7

All about trees by Brian O'Meara

Goals:

- Study the information content of phylogenetically structured data
- Learn about particular tree formats in ape, phylobase, and ouch
- Learn how to hand-make trees
- Learn how to import trees from nexus and newick formats
- Learn tree conversion from one format to another

Concepts:

- file access
- classes
- coercion

7.1 Tree objects

In nature, a tree is a large perennial woody plant. It has roots, a main trunk, branches, and leaves. In graph theory, a tree is a network where there is only one path between any two nodes (in other words, a network with no cycles). In phylogenetics, we use ideas and terminology from both graph theory and nature. **Terminal taxa** are also known as **leaves**, **terminals**, **OTU**s ("**Operational Taxonomic Units**"), **tips**, or simply **taxa**. **Branches** are also called **edges**. **Internal** nodes (places where two or more branches connect) are also known as vertices and sometimes simply **nodes** (technically, leaves are

also nodes). A **rooted** tree has one node designated as the **root**, and all other nodes are descended from this root. An **unrooted** tree has no root designated. Traditionally, the root node has at least two descendants; it may also have a subtending branch. A tree where every internal node has two and only two descendants is known as a **binary** or **bifurcating** tree. A tree where at least one internal node has more than two descendants is said to be **multifurcating**; such a node is a **polytomy**. Trees in phylogenetics generally represent either **species trees** (a history of the splitting of interbreeding populations) or **gene trees** (a history of the coalescence of gene copies). In both cases, it is generally believed that the true process is bifurcating, so that each split results in two descendants. Thus, polytomies on trees are generally taken as representing uncertainty in the relationships. Branches may have **lengths**; these lengths may correspond to time, amount of change in some set of characters, number of speciation events, or some other measure. A tree where all branch lengths from root to tips are equal is known as an **ultrametric** tree. A tree without branch lengths is known as a **topology**. A **clade** is an ancestor and all its descendants. Any **edge** corresponds to a **bipartition**: a division of the tree into two parts connected by that edge (if a root were inserted on the edge, then each of those parts would be a clade).

7.1.1 Newick

A very basic tree description is Newick (named after the seafood restaurant in New Hampshire where it was formalized; it is also called New Hampshire format for that reason). It is simply a string. Each nesting on the tree corresponds to a parenthetical statement. For example, for this tree: Taxa G and F form a clade, as do G, F, and E, as do A and B, and so forth. Thus, to create a Newick string, just go down the tree, nesting as you go:

(G,F)

((G,F),E)

other side:

(A,B)

(C,D)

((A,B),(C,D))

all together:

(((G,F),E),((A,B),(C,D)))

And that's it (it will be clearer in the lecture) If a tree has branch lengths, these are entered following the descendant clade. For example, if the branch leading to G has length 1.0, we would write G:1.0 rather than just G. If the tree is ultrametric, and the branch below the common ancestor of G and F is of length 1.1, and the branch below

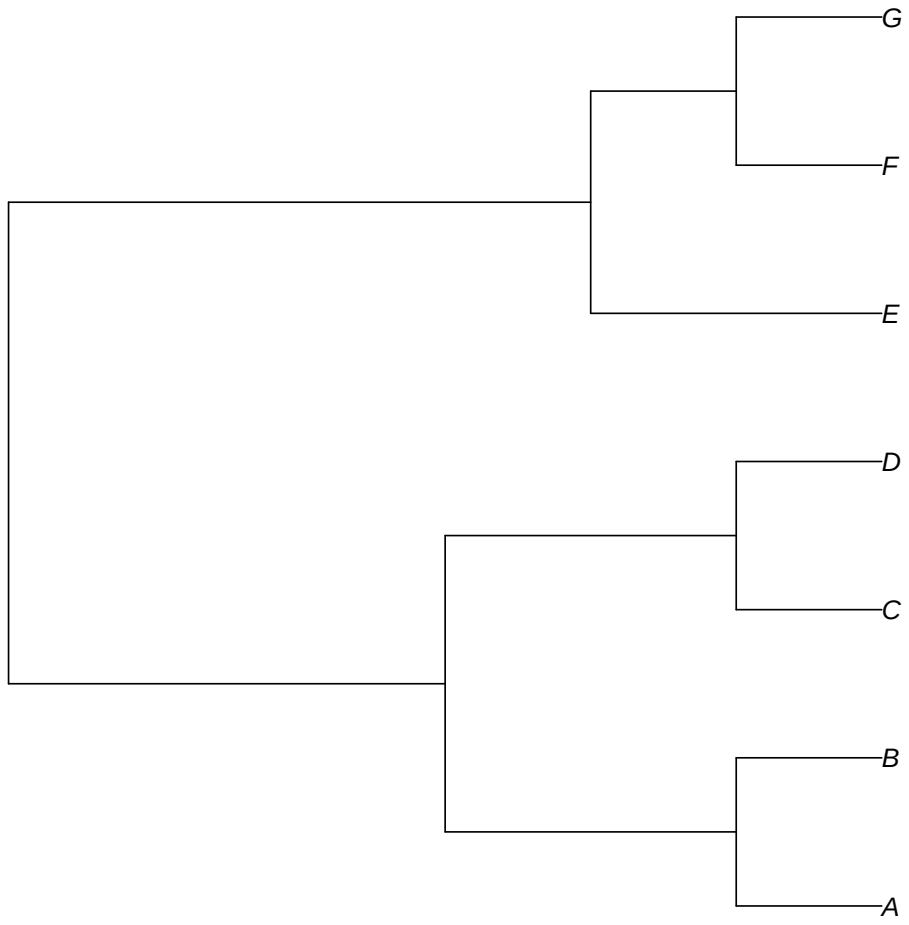


Figure 7.1: A simple tree

that of length 3.5, we could write

((G:1.0,F:1.0):1.1,E:2.1):3.5

And so forth. One problem with Newick representation is that there are many ways of representing the same tree. At every node, one can rotate the descendant branches (switching the left and right positions) and get the same tree (for example, imagine switching the G and E labels). Thus, the Newick strings

((G,F),E)

and

((F,G),E)

describe the same trees, though it might not be easy to tell at first glance. This is generally an issue for any tree representation. Newick strings also don't lend themselves to easy

tree traversal (moving up or down the tree). In most software, some other representation is used.

7.1.2 phylo (ape 1.9 or above)

The `ape` package (Paradis et al., 2004) uses a different representation of trees. It uses R `structures`, `lists`, `matrices`, and `vectors` to store a tree. Each node in the tree receives a number. For example, here is the tree from before in `ape` format. First let's clear any old workspace, load our libraries, and load our tree called "`simpletree`" from "`Rdata/simpletree.rda`".

```
> rm(ls=list())
> require(ape)
> require(ouch)
> require(phylobase)
> load("Rdata/simpletree.rda")
```

Here is `simpletree` with the node numbers printed. It is printed with the following commands:

```
> plot(simpletree,no.margin=TRUE)
> nodelabels()
> tiplabels()
```

For a tree with N tips, the tips have numbers 1...N and the nodes have numbers greater than N (this is in contrast to how this was done in earlier (<1.9) versions of `ape`). These numbers are used to store information about the tree's structure. To do this, a matrix is created, with height corresponding to the number of internal and terminal nodes and width 2. The first column of the matrix has the node at the beginning of the branch, the second has the node at the end of the branch. For example, for our simple tree, this matrix is

```
> simpletree$edge
```

```
     [,1] [,2]
[1,]    8    9
[2,]    9   10
[3,]   10    1
[4,]   10    2
[5,]    9   11
[6,]   11    3
[7,]   11    4
```

```
 [8,]    8   12
 [9,]   12    5
[10,]   12   13
[11,]   13    6
[12,]   13    7
```

This alone is enough for a basic topology. However, it might be nice to know what the taxa actually are, rather than just numbers. To do this, a character vector with as many entries as the number of tips is used. In the example tree, this is

```
> simpletree$tip.label
```

```
[1] "A" "B" "C" "D" "E" "F" "G"
```

It's possible that internal nodes have labels, too (for example, the most recent common ancestor of a set of birds might be labeled "Aves"). If so, an optional `node.label` is used. If branch lengths are known, they are included as the numeric vector `edge.length`.

```
> simpletree$edge.length
```

```
 [1] 1.5 1.0 0.5 0.5 1.0 0.5 0.5 2.0 1.0 0.5 0.5 0.5
```

Finally, there are a few other elements (`Nnode`, the number of internal nodes; `class=phylo`) to give more information. To see what the internal representation of a tree is, you can use `unclass` (the S4 analog is `attributes`):

```
> unclass(simpletree)
```

```
$edge
      [,1] [,2]
 [1,]    8    9
 [2,]    9   10
 [3,]   10    1
 [4,]   10    2
 [5,]    9   11
 [6,]   11    3
 [7,]   11    4
 [8,]    8   12
 [9,]   12    5
[10,]   12   13
[11,]   13    6
```

```
[12,]   13    7

$tip.label
[1] "A" "B" "C" "D" "E" "F" "G"

$Nnode
[1] 6

$edge.length
 [1] 1.5 1.0 0.5 0.5 1.0 0.5 0.5 2.0 1.0 0.5 0.5 0.5
```

`phylo` trees are S3 objects. We'll be learning more about them later in the week, but an important thing to know is that you directly access any element of them by using the `$` operator (as was done above). Optional elements, or even elements of your own devising, can be added to them, too, using the same operator.

7.1.3 ouchtree

`OUCH` (the most recent version) uses a different tree structure than does `ape`. First, `OUCH`'s is an S4 class, rather than S3. There are several differences between them, which you'll learn later. There are two main distinctions that will be important now. It helps to have a metaphor: think of a car. The S3 representation of a car is all the parts, neatly disassembled and laid out. The S4 representation of a car is a closed box. With S3, you can look at and manipulate any part of the car directly and manipulate it (using the `$` operator). You could check the amount of gas in the tank by directly accessing the gas. With S4, you should use a method, if one exists, to access and manipulate elements. For example, you could check the gas in the tank using the fuel gauge, if the fuel gauge method exists and works properly. S3 objects can be built up piecemeal, and there aren't built-in checks to make sure that everything is correct: if you forget to add a wheel element to the S3 char, you won't know there's a problem until some function tries to access it and fails. S4 objects are instantiated once, when you pass them all the initialization info they need (they often have defaults, and often have internal consistency checks). `OUCH` uses the `ouchtree` class as a basic tree class, then derives other classes from this for storing information on analyses. The `ouchtree` class is:

```
setClass(
  'ouchtree',
  representation=representation(
    nnodes = 'integer',
    nodes = 'character',
    ancestors = 'character',
```

```
    nodelabels = 'character',
    times = 'numeric',
    root = 'integer',
    nterm = 'integer',
    term = 'integer',
    anc.numbers = 'integer',
    lineages = 'list',
    epochs = 'list',
    branch.times = 'matrix',
    depth = 'numeric'
  )
)
```

At first glance, it looks like creating a new `ouchtree` object will be a lot of work: there are 13 different elements, some of them vectors, built in the class. However, with S4 objects, the beauty of constructors comes into play. The constructor for a new `ouchtree` is just the function

```
ouchtree(nodes, ancestors, times, labels = as.character(nodes))
```

This function only has four arguments, one of them optional. Using the function and these elements, all the other elements of the class are initialized. The first element is `nodes`, a character vector of node ids (including internal nodes). Unlike `ape`, the leaves do not need to have smaller ids than internal nodes. The second argument is `ancestors`, a character vector of node ids of the ancestors for the nodes in the `nodes` vector. The `nodes` and `ancestors` vectors almost correspond to the second and first columns of the `ape edge` matrix, respectively, with the exception that `ouchtree` includes the root node with an ancestor of `NA`. The third element, `times`, represents the height of each node from the root. Remember that `ape`'s `edge.length` vector has the length of the branch subtending each node; instead, `ouchtree` has the sum of the lengths of all branches connecting a given node to the root. Again, the root node is included in `ouchtree` (with height 0) but not in ape. The fourth argument, `labels`, is a vector of labels for both tips and internal nodes. If internal nodes do not have names, they get a label of `<NA>`. For example, our example tree, when converted to ouchtree format, is

```
> attributes(simpletreeouch)$nodes

 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13"

> attributes(simpletreeouch)$ancestors
```

```
 [1] NA  "3" "1" "6" "6" "1" "2" "2" "3" "4" "4" "5" "5"
```

```
> attributes(simpletreeouch)$times
```

```
 [1] 0.0000000 0.8333333 0.6666667 0.8333333 0.8333333 0.5000000 1.0000000
 [8] 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
```

```
> attributes(simpletreeouch)$nodelabels
```

```
 [1] ""  ""  ""  ""  ""  ""  "G" "F" "E" "D" "C" "B" "A"
```

One other element of `ouchtree`, created on initialization, is a matrix showing shared amount of time on a tree between two tips (which may be the same tip). This, multiplied by a rate parameter, becomes a variance-covariance matrix under a Brownian motion model, which we'll be discussing in the course.

```
> attributes(simpletreeouch)$branch.times
```

```
          [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
[1,] 1.0000000 0.8333333 0.6666667 0.0000000 0.0000000 0.0000000 0.0000000
[2,] 0.8333333 1.0000000 0.6666667 0.0000000 0.0000000 0.0000000 0.0000000
[3,] 0.6666667 0.6666667 1.0000000 0.0000000 0.0000000 0.0000000 0.0000000
[4,] 0.0000000 0.0000000 0.0000000 1.0000000 0.8333333 0.5000000 0.5000000
[5,] 0.0000000 0.0000000 0.0000000 0.8333333 1.0000000 0.5000000 0.5000000
[6,] 0.0000000 0.0000000 0.0000000 0.5000000 0.5000000 1.0000000 0.8333333
[7,] 0.0000000 0.0000000 0.0000000 0.5000000 0.5000000 0.8333333 1.0000000
```

The entire content of the `simpletreeouch` object can be dumped to screen using the following command (not executed here to save paper):

```
> attributes(simpletreeouch)
```

7.1.4 phylo4 (phylobase)

`Phylobase` is a new package for phylogenetic trees and datasets, started in December 2007 at a NESCent-sponsored hackathon. Its development is ongoing, so some of its function names and class elements may change. We'll be discussing it more later in the course. It has two main classes: `phylo4` and `phylo4d`. The first is just a tree class, the second includes a tree and data. Its tree class is closely (and intentionally) based on `ape`'s `phylo` object: it has an `edge` matrix, `edge.length` vector, `tip.label` vector,

node.label vector (not optional), and Nnode variable. It also has an edge.label vector, which is distinct from the node.label and tip.label vectors (i.e., may have different names) and an element, root.edge that can specify where the root is (or NA if the tree is unrooted). phylo4d is derived from the phylo4 class (a concept common in S4 and in object-oriented languages, like C++) and thus has all the elements of phylo4, plus elements tipdata, nodedata, edgedata for storing data.frames of data (typically, this will just be data at tips, such as nucleotide sequences, but internal nodes could have reconstructed sequences and their might be data for edges, too, such as estimated changes on the tree). The constructor for a phylo4 object is

```
phylo4(edge, edge.length = NULL, tip.label = NULL, node.label = NULL, edge.label
= NULL, root.edge = NULL, ...)
```

The only required argument is edge (a phylo-style edge matrix). For everything else, there are default constructors that will create names and other needed information.

The entire content of the simpletree object in phylo4 are

```
> attributes(as(simpletree,"phylo4"))

$edge
      ancestor descendant
 [1,]        8          9
 [2,]        9         10
 [3,]       10          1
 [4,]       10          2
 [5,]        9         11
 [6,]       11          3
 [7,]       11          4
 [8,]        8         12
 [9,]       12          5
[10,]       12         13
[11,]       13          6
[12,]       13          7

$edge.length
 [1] 1.5 1.0 0.5 0.5 1.0 0.5 0.5 2.0 1.0 0.5 0.5 0.5

$Nnode
[1] 6

$tip.label
[1] "A" "B" "C" "D" "E" "F" "G"

$node.label
```

```
[1] "N1" "N2" "N3" "N4" "N5" "N6"

$edge.label
 [1] "E9"  "E10" "E1"  "E2"  "E11" "E3"  "E4"  "E12" "E5"  "E13" "E6"  "E7"

$root.edge
[1] NA

$class
[1] "phylo4"
attr(,"package")
[1] "phylobase"
```

7.2 Getting trees into R

There are ways to use R to estimate phylogenetic trees given a set of taxa with characters. For more information on this, see Paradis (2006). In many cases, however, there will be trees saved in an existing file, saved as a result of an analysis in programs such as PAUP or MrBayes. This section will discuss getting those trees into R. Note there may be additional ways to load trees not discussed here. For example `apTreeshape` can load trees directly from `http://www.treebase.org` if you know the appropriate study number; as these trees lack branch lengths, they are generally unsuitable for the sort of analyses we will be doing in this course.

7.2.1 Using ape

`Ape` can read Newick trees by using the function:

```
read.tree(file = "", text = NULL, tree.names = NULL, skip = 0, comment.char
= "#", ...)
```

There are three main ways to use this function:

`read.tree()`: Gets interactive input of a Newick string from the user

`read.tree(text="((A,B),C);")`: Inputs a Newick tree string directly. Note that the tree string needs to end with a semicolon

`read.tree(file="mytree.txt")`: Inputs one or more Newick strings from a file.

See the documentation for the (little-used) other arguments.

The other way ape can get trees is from NEXUS files. NEXUS is a standard (see Maddison et al. 1997) used in many phylogenetics programs like PAUP, MrBayes, MacClade, and

Mesquite and can contain blocks with trees, characters, batch commands for programs, and more. `Ape` can get trees (and only trees) from such files using the command

```
read.nexus(file, tree.names = NULL)
```

Ape treats all trees as rooted and ignores tree weights (which can be output by PAUP and MrBayes). One gotcha associated with `ape`'s tree input functions is that if the file has one tree, the returned item is a tree object, but if it contains more than one tree, a list of trees is returned. These two kinds of objects must be used differently in R.

7.2.2 Using phylobase

One of the key features of `phylobase` is the ability to load trees and data directly from NEXUS files. To get trees, the function is

```
NexusToPhylo4(fileToRead, multi = FALSE)
```

This works as you'd expect. If `multi=FALSE`, the default, it works as `ape`'s input functions do, returning a single object if there is one tree in the file and a list of objects if there are multiple trees. If `multi=TRUE`, it always returns a list, even if there is just one element. This way, the type of the returned item is constant regardless of the number of trees. For data alone,

```
NexusToDataFrame(fileToRead, allchar = FALSE, polymorphictomissing = TRUE,
 levelsall = TRUE)
```

can be used (it has features to convert categorical data to factors, DNA data to strings, continuous data to an appropriate data.frame, and more – see documentation).

```
NexusToPhylo4D(fileToRead, multi=FALSE, allchar=FALSE,  polymorphictomiss-
ing=TRUE, levelsall=TRUE)
```

loads data and trees into one `phylo4d` object. Note that the names of these functions may change in the future to make capitalization more consistent with the rest of `phylobase`.

7.3 Going from one format to another

The R packages `ape`, `phylobase`, `ouch`, `geiger`, `apTreeshape`, `picante`, `laser`, `phangorn`, `PhySim`, `ade4`, `PHYLOGR`, and others all use trees (see http://www.r-phylo.org). Unfortunately, they often use different tree formats, sometimes within the same package! [For example, if the format has changed and not all functions have been updated to use the new version]. There are ways to convert between formats. Some use functions of the type you've come to expect: `output<-in2out(input)`. Others use coercion: `output<-as(input, "output class name")`. This has the advantage of being much more standardized (no need to wonder whether the function is 'in2out' or 'in.to.out'

or 'convertIn2Out') and generally simpler to use. One thing to note is that there are

Table 7.1: The input object is always called "object"

from	to	command	package
phylo	phylo4	`as(object, "phylo4")`	phylobase
phylo4	phylo	`as(object, "phylo")`	phylobase
phylo4d	phylo	`as(object, "phylo")`	phylobase
phylo4	phylo4d	`as(object, "phylo4d")`	phylobase
phylo4d	phylo4	`as(object, "phylo4d")`	phylobase
phylo4d	phylog (ade4)	`as(object, "phylog")`	phylobase
hclust	phylo	`as.phylo(object, ...)`	ape
phylo	hclust	`as.hclust(object, ...)`	ape
phylog (ade4)	phylo	`as.phylo(object, ...)`	ape
old phylo	phylo	`old2new.phylo(object)`	ape
phylo	old phylo	`new2old.phylo(object)`	ape
phylo	treeshape	`as.treeshape(object, model, p, ...)`	apTreeshape
treeshape	phylo	`as.phylo(object, ...)`	apTreeshape
phylo	ouchtree	`ape2ouch(object, ...)`	ouch
phylo	earlier ouch format	`ape2ouch(object, data, ...)`	geiger

two `ape2ouch` functions. The one in `geiger` converts a `phylo` object and data to the `data.frame` that earlier versions of `ouch` used. The one in the lastest version of `ouch` converts just a `phylo` object to an `ouchtree` object.

7.4 Exercises

1. Take `simpletree` in ape's phylo format and set all branch lengths equal to 1.0, plot to verify.

2. Convert `simpletree` to phylobase's phylo4 format. Which packages do you need in order to do this? Plot with the display nodes option (show.node=T).

3. Convert to ouch. Plot with displaying nodes (node.names=T). Convert to data frame representation.

4. Assign node labels to your tree object. What is the easiest way to do that?

   Try assigning the same value for the node label to more than one node in the tree (not the tips).

   Try assigning unique node labels. Do both work? Are there problems?

```
> plot(simpletree,no.margin=TRUE)
> nodelabels()
> tiplabels()
```

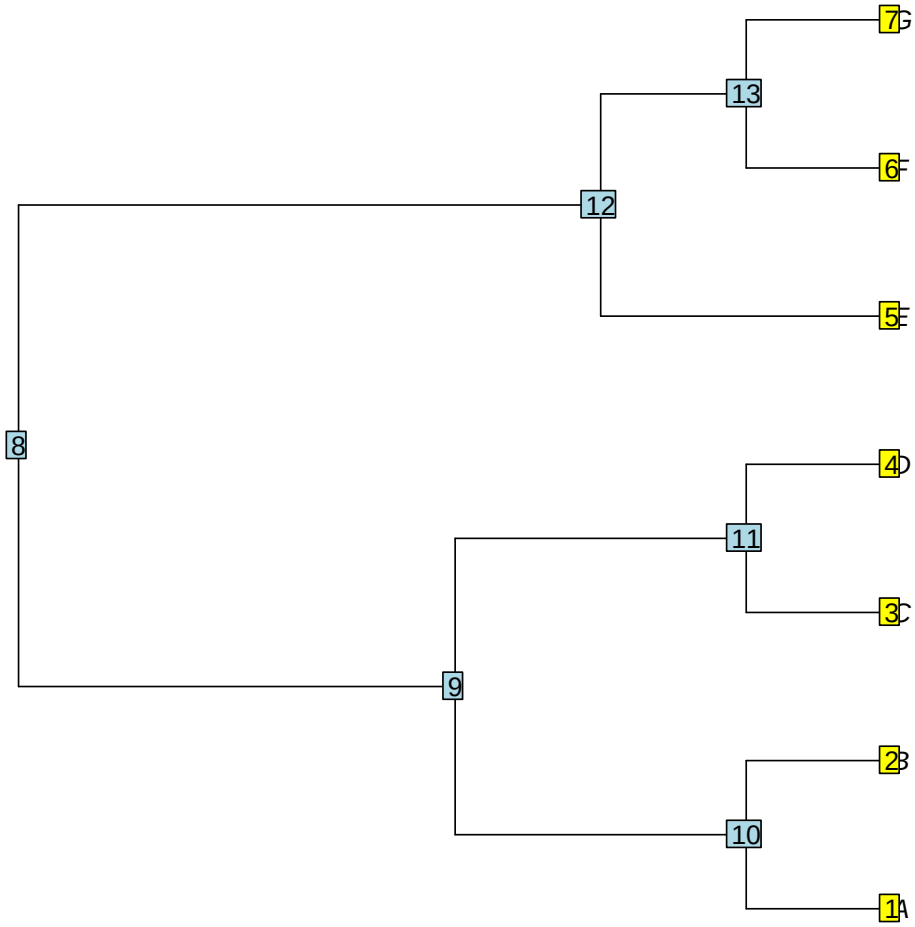


Figure 7.2: A simple tree with ape's numbering of nodes included

Chapter 8

Verification: Computing Phylogenetic GLS "by hand"

Chapter Topics:

- Checking your computations by other means
- Deconstructing `ape` objects

Skills: accessing object elements, constructing similarity matrices, using R matrix math functions, using R linear model functions.

A good practice is to try to verify the software you are using by doing things another way. Phylo GLS is a good one because it is fairly simple mathematically. Recall the equation (3.2). All we need to do is take our data and, using matrix math, divide by the square root of the phylogenetic covariance matrix. Thus we need to do the following steps:

1. Compute the phylogenetic covariance matrix expected under BM. This is a "similarity" matrix based on the amount of time they share along the phylogeny. Let's call this `tbm`
2. Take the inverse of the matrix, `tbmi`.
3. Find the square root of `tbmi`, a good way to do this is by cholesky decomposition.
4. Multiply our data vectors by `roottbmi` to get our phylogenetically transformed data.
5. Run regression analysis on the transformed data.

Let's load our primate tree that we saved earlier:

```
> require(ape)
> load("Rdata/tree.primates.rda")
> tree <- tree.primates
> names(tree)
```

```
[1] "edge"        "Nnode"       "tip.label"   "edge.length"
```

```
> tree$edge
```

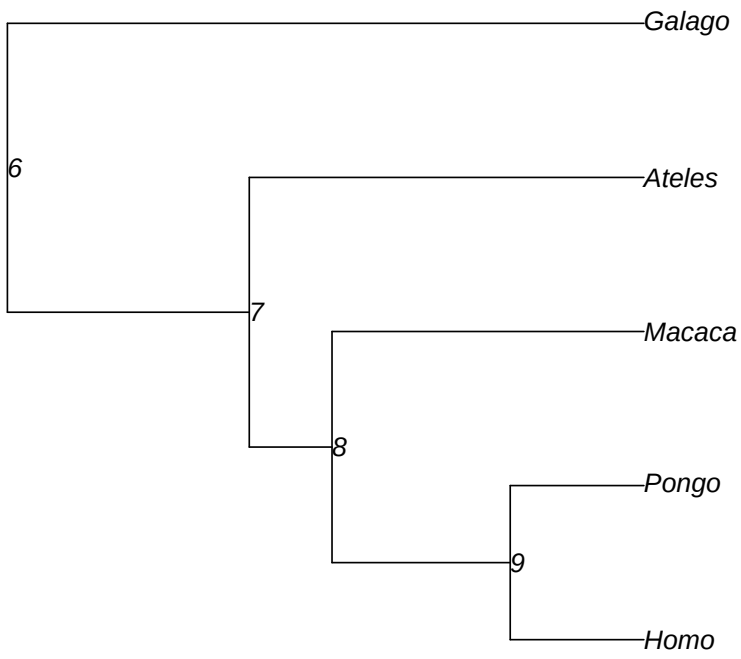
```
     [,1] [,2]
[1,]    6    7
[2,]    7    8
[3,]    8    9
[4,]    9    1
[5,]    9    2
[6,]    8    3
[7,]    7    4
[8,]    6    5
```

```
> tree$tip.label
```

```
[1] "Homo"   "Pongo"  "Macaca" "Ateles" "Galago"
```

Remember that the first column in the edge matrix is the ancestral node, and the second column is the descendant node. If you think of the descendant node as the reference point, then the tips are nodes 1-5, and the internal nodes are therefore 6-9. We can assign the node.labels and plot them on the tree:

```
> tree$node.label <- c(6:9)
> plot(tree, show.node.label = TRUE)
```



Let's make a dataframe so that we can see the tree structure and associated metadata more clearly. We're using the function `with`, which defines a small local environment where we are using the object `tree`.

Let's `cbind` together the `edge` matrix, which describes the ancestor-descendant pairs, with the appropriate branch lengths, and species and tip labels. Note that the edge matrix doesn't have a row for the most basal node in the tree as a descendant, so we have to drop the first node label from our vector (otherwise the label vector will be too long by one).

```
> with(tree, cbind(tree.primates$edge, edge.length, labels = c(node.label[-1],
+     tip.label)))
```

```
             edge.length labels
[1,] "6" "7" "0.38"      "7"
[2,] "7" "8" "0.13"      "8"
[3,] "8" "9" "0.28"      "9"
[4,] "9" "1" "0.21"      "Homo"
```

```
[5,] "9" "2" "0.21"      "Pongo"
[6,] "8" "3" "0.49"      "Macaca"
[7,] "7" "4" "0.62"      "Ateles"
[8,] "6" "5" "1"         "Galago"
```

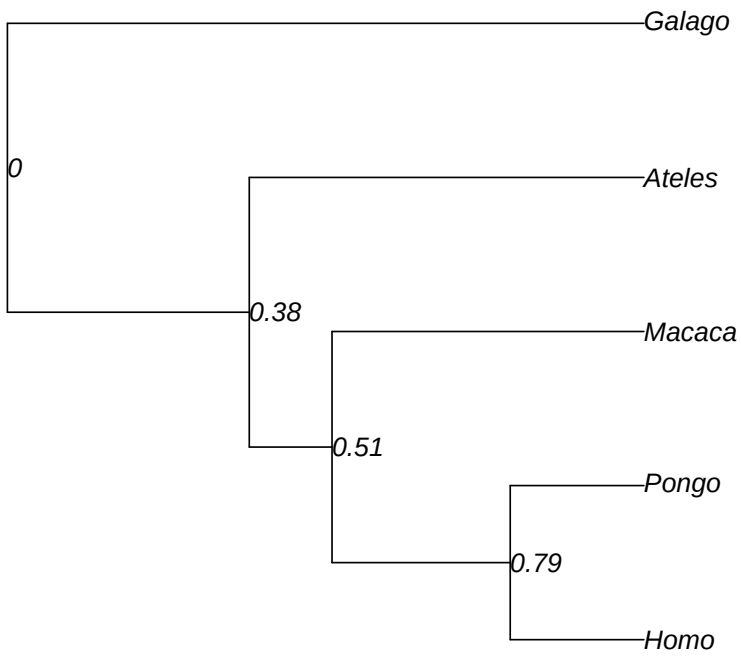
Since we need to calculate a similarity matrix based on time in shared evolutionary history, it would actually be much more convenient to have for each node, the time from the root to the node. Let's call this "times". For a small and simple phylogeny, we can do this by "hand". Looking at the dataframe we just made, we can add up the branches to each node, with the base of the tree at time zero and the tips at time 1, and then remake the matrix (again, we drop the basal node from this vector):

```
> times <- c(0, 0.38, 0.38 + 0.13, 0.38 + 0.13 + 0.28, 1, 1, 1,
+     1, 1)
> with(tree, cbind(tree.primates$edge, edge.length, labels = c(node.label[-1],
+     tip.label), times = times[-1]))
```

```
             edge.length labels   times
[1,] "6" "7" "0.38"      "7"      "0.38"
[2,] "7" "8" "0.13"      "8"      "0.51"
[3,] "8" "9" "0.28"      "9"      "0.79"
[4,] "9" "1" "0.21"      "Homo"   "1"
[5,] "9" "2" "0.21"      "Pongo"  "1"
[6,] "8" "3" "0.49"      "Macaca" "1"
[7,] "7" "4" "0.62"      "Ateles" "1"
[8,] "6" "5" "1"         "Galago" "1"
```

Looks good so far. Let's plot these "times" on the tree. Replace the node.label with times values and replot:

```
> tree$node.label <- as.character(times[1:4])
> plot(tree, show.node.label = TRUE)
```



Now, looking at the tree, let's make our tbm matrix. First, make an empty matrix with just species names to help us with a mental image of what we're doing:

```
> tbm <- matrix(nrow = 5, ncol = 5)
> rownames(tbm) <- colnames(tbm) <- tree$tip.label
> tbm
```

```
       Homo Pongo Macaca Ateles Galago
Homo     NA    NA     NA     NA     NA
Pongo    NA    NA     NA     NA     NA
Macaca   NA    NA     NA     NA     NA
Ateles   NA    NA     NA     NA     NA
Galago   NA    NA     NA     NA     NA
```

Now look at the phylogeny and fill in the pairwise similarities by the amount of evolutionary history they share:

```
> tbm[1, ] <- c(1, 0.79, 0.51, 0.38, 0)
> tbm[2, ] <- c(0.79, 1, 0.51, 0.38, 0)
> tbm[3, ] <- c(0.51, 0.51, 1, 0.38, 0)
> tbm[4, ] <- c(0.38, 0.38, 0.38, 1, 0)
> tbm[5, ] <- c(rep(0, 4), 1)
> tbm
```

```
       Homo Pongo Macaca Ateles Galago
Homo   1.00  0.79   0.51   0.38      0
Pongo  0.79  1.00   0.51   0.38      0
Macaca 0.51  0.51   1.00   0.38      0
Ateles 0.38  0.38   0.38   1.00      0
Galago 0.00  0.00   0.00   0.00      1
```

Of course, one could program an automated way to do this, but that would take a lot of time and skill. But this is a verification, so we compute it by hand. Now we are ready to begin transforming our data. First do the inversion and root of tbm using the `solve()` and `chol()` functions, respectively. These are both part of the `base` package.

```
> tbmi <- solve(tbm)
> roottbmi <- chol(tbmi)
```

Now transform our data. For the regression model, we will also have to transform the intercept, so we bind a column of 1's with the X variable (independent variable in this example) and make transformed variables Z and U. Note that matrix multiplication is designated by %*% (Otherwise multiplication is element-by-element, the default in R) :

```
> Z <- roottbmi %*% Y
> U <- roottbmi %*% cbind(1, X)
```

(Feel free to look at any of these matrices we're creating). Now we just have to run the regression. Since the intercept term is inside the X matrix now, we do a regression without an intercept (one is added automatically unless you say no). You can use either lm or gls in this case because we don't have to specify a correlation structure. Let's compare that with the phylogenetic GLS, and with the regression from PIC:

```
> summary(lm(Z ~ U - 1))
```

```
Call:
lm(formula = Z ~ U - 1)
```

```
Residuals:
    Homo    Pongo   Macaca   Ateles   Galago
 1.73621 -0.66358  0.03030 -0.48572  0.43733

Coefficients:
   Estimate Std. Error t value Pr(>|t|)
U    2.5001     0.7755   3.224   0.0484 *
UX   0.4319     0.2865   1.508   0.2288
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.138 on 3 degrees of freedom
Multiple R-squared: 0.8746,        Adjusted R-squared: 0.791
F-statistic: 10.46 on 2 and 3 DF,  p-value: 0.04442


> summary(gls(Y ~ X, correlation = corBrownian(phy = tree.primates),
+     data = XY))


Generalized least squares fit by REML
  Model: Y ~ X
  Data: XY
       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
X           0.4319328 0.2864904 1.507669  0.2288

 Correlation:
  (Intr)
X -0.437

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
```

```
> summary(lm(pic.Y ~ pic.X - 1))
```

```
Call:
lm(formula = pic.Y ~ pic.X - 1)

Residuals:
       6        7        8        9
-0.55351  0.35263  0.03311  1.85770

Coefficients:
      Estimate Std. Error t value Pr(>|t|)
pic.X   0.4319     0.2865   1.508    0.229

Residual standard error: 1.138 on 3 degrees of freedom
Multiple R-squared: 0.4311,        Adjusted R-squared: 0.2414
F-statistic: 2.273 on 1 and 3 DF,  p-value: 0.2288
```

Chapter 9

Sweave

We are going to take a moment or two to learn a little LaTeX and an R function called `Sweave`. This may seem like a really painful idea, BUT, the payoff is really big. You may have guessed, this is how Todd, Brian, and I produced this tutorial. Package developers are now using `Sweave` to produce vingettes, small package tutorials that are actually compiled with the package (you can see a list of all available vingettes on you computer by typing `vignette()`.

I use it routinely in data analysis, which evolves directly into manuscripts. Some publishers accept manuscripts in LaTeX and combined with BibTeX the reference management tool, adding Sweave gives so much functionality that it is becoming increasingly popular. The analysis *becomes* part of the document. Let's take a little look.

9.1 The Notion of Reproducible Results

`Sweave` provides "literate programming" for R. This new idea (or movement) is really well explained at Charlie Geyer's website: http://www.stat.umn.edu/c̃harlie/Sweave

Let's take a look at it now.

9.2 A bit about LaTeX

At first, LaTeXlooks a little scary. There are curly braces around everything. But all you have to do is learn the bare basics. In TeXShop, take a look at the file in the `Rcomparative`
`misc` folder called "small2e.tex". It is one of the "official" sample documents for LaTeX.

Two elements are required in all LaTeXdocuments:

1. A document class. This can be article (most common), book, report, etc.

2. A begin document and end document line, which opens and closes the text that will be displayed in the final document.

Additionally, these basics are very helpful:

1. Remember that some characters are special characters in LaTeX. You can use them in your document, but you must set them off with an escape character — the backslash. `\& \$ \# \% \_ \{ \} \^ \~`

   Just knowing these characters can save you a lot of aggravation.

2. Many formatting commands have the syntax backslash commandname{} surrounding the words

   ```
   \section{ PUT SECTION NAME HERE }
   ```

   Italics are indicated by

   ```
   \emph{ WORD }
   ```

   Bold:

   ```
   \textbf{ WORD }
   ```

   Many of these are in the Macros menu in TeXShop

3. Heirarchical arrangements are very natural. Just put a "sub" in front of section, subsubsection for the next level, and so on. Works on emph as well.

4. Another common syntax is having begin and end tags for environments. For example:

   ```
   \begin{center}
   ...  Figure, Table, Text, etc.
   \end{center}
   ```

   Will center whatever is between the tags.

   ```
   \begin{itemize}
       \item FIRST ITEM
       \item SECOND ITEM
   \end{itemize}
   ```

   Will generate a bullet point list.

When you want to get fancier, there is an amazing wealth of possiblities for formatting figures, tables, equations, etc. A very helpful source is the LaTeXmanual on Wikibooks: http://en.wikibooks.org/wiki/LaTeX.

9.3 Simple Sweave

I wrote two Sweave demo files, both in the `Rcomparative misc` folder: "`SweaveSample.pdf`" and a pared-down example "`SweaveMinimalist.Rnw`". Then I found Charlie Guyer's exaple, which is much better "`foo.Rnw`".

9.4 Sweave -> LaTeX

After looking at these Rnw documents, convert these examples to LaTeXby running the command from R:

```
> Sweave("foo.Rnw")
```

Alternatively, you can run Sweave from the terminal. Make sure you are in the same directory as the document, then type

```
R CMD Sweave foo
```

9.5 LaTeX -> pdf

If the LaTeXis generated free of error, you can generate a pdf from it by opening up the `.tex` files in TeXShop (just double click on the `.tex` file). Click on "Typeset". If you get errors, clicking on the "Goto Error" button can be very helpful.

You can also do this step at the command line (Terminal):

```
latex foo
xdvi foo
```

Either way, if you have figures generated by R, you will see a bunch of pdfs and eps files appear. These are the individual figures that get inserted into the document. There are also auxiliary files.

9.6 Stangle

Another very convenient function is `Stangle`, which is a sort of opposite of `Sweave`. It strips out all of the text content of the document, and produces a `.R` file from the code chunks in the `.Rnw` document:

```
> Stangle("foo.Rnw")
```

The code chunks are numbered, which is very helpful when debugging your R code, as R will halt and tell you which code chunk it failed at. It is also helpful for distributing the code examples.

```
R CMD Stangle foo
```

9.7 Best Practices

Don't write the entire `Sweave` *document at once!* Write one code chunk at a time and check that it compiles cleanly. Go slow, especially at first. Once you get confident, you can write a bit more.

DIvide and Conquer If you are finding debugging the R code frustrating when combined with the LaTeX, then try writing the R script first, then pasting it into the `.Rnw` document.

Include all your steps in the .Rnw file The point is to be able to go from start to finish, reproducing the results. Make sure the starting point is clear. Sometimes, you may want one .Rnw file to go from raw data to an R data file, then another one for the analysis starting from the R data file. This is fine, of course. Just think it through and organize in a logical and most simple manner.

Write clean code This will really sharpen your coding skills. Nothing will motivate you like the prospect of releasing it to the world.

Write each bit of code once and only once If you want to run it again, label the code chunk and call the code chunk next time. As your code evolves, you may end up changing the code chunk, but you could easily forget to change all the copies. If it occurs only once in your code, the single change will be propagated throughout the document:

```
<<label=twotwo>>=
2+2
@

<<>>=
<<twotwo>>=
@
```

9.8 Exercises

Obviously... go write some `Sweave` documents! Start of slow and simple. It's like riding a bike. Shaky at first, but then it becomes second nature.

Chapter 10

S3 vs. S4 Objects

References:

- Freidrich Leisch's lecture on S4 classes and methods
- Programming with Data: A guide to the S language, by John M. Chambers, 1998, ISBN is 0-387-98503-4

`phylobase` and `ouch` are written in the newer S4 class system (the one which you've been using and learning is the S3 class system). The main difference between these two systems is in the degree to which they follow the object-oriented programming model.

10.1 What is an object?

R works on objects:

- Objects are ways of bundling parts of programs into small, manageable pieces.
- Objects are simply a definition for a type of data to be stored

  e.g., data vector, matrix, array, data frame, list, function
- An object is a component of a program that knows how to perform certain actions and to interact with other pieces of the program.
- Functions can be described as "black boxes" that take an input and spit out an output. Objects can be thought of as "smart" black boxes. That is, objects can know how to do more than one specific task (method or behavior), and they can store their own set of data.

Table 10.1: Attributes of Hero characters.

Attribute	Value
Health	16
Strength	12
Agility	14
WeaponType	”mace”
ArmorType	”leather”

Table 10.2: Behaviors or methods of Hero characters.

Methods
move through the maze
attack monsters
pick up treasure

- *It is an abstraction:* Objects are something that have **attributes** (values) and **behaviors** (actions). These are sometimes called **states** and **methods**. These are formally defined in the object definition.

10.2 Object example: A Medieval Video Game (remember Dungeons and dragons?)

We have two types of players: **Monsters** and **Heros**. For the hero character, we need to store the values of certain **attributes** (Table **??**).:

Heros must also be able to certain behaviors (which we will call **methods**:

These **attributes** and **behaviors** completely define the Hero. Modules may be written that know how to interpret (interact with) heros.

10.3 S3 Classes

Similarly, S3 classes have attributes and methods. If we create a data.frame, we can ask R what its attributes and methods available are:

```
> flies <- data.frame(species=c("melanogaster", "silvestris", "heteroneura"),
+ headW=c(3.5, 5, 12))
> attributes(flies)

$names
[1] "species" "headW"
```

```
$row.names
[1] 1 2 3

$class
[1] "data.frame"
```

```
> methods(class="data.frame")
```

```
 [1] $<-.data.frame           Math.data.frame
 [3] Ops.data.frame           Summary.data.frame
 [5] [.data.frame             [<-.data.frame
 [7] [[.data.frame            [[<-.data.frame
 [9] aggregate.data.frame     as.data.frame.data.frame
[11] as.list.data.frame       as.matrix.data.frame
[13] by.data.frame            cbind.data.frame
[15] dim.data.frame           dimnames.data.frame
[17] dimnames<-.data.frame    duplicated.data.frame
[19] edit.data.frame*         format.data.frame
[21] formula.data.frame*      getCovariate.data.frame*
[23] getGroups.data.frame*    getResponse.data.frame*
[25] head.data.frame*         is.na.data.frame
[27] mean.data.frame          merge.data.frame
[29] na.exclude.data.frame*   na.omit.data.frame*
[31] plot.data.frame*         print.data.frame
[33] prompt.data.frame*       rbind.data.frame
[35] row.names.data.frame     row.names<-.data.frame
[37] rowsum.data.frame        split.data.frame
[39] split<-.data.frame       stack.data.frame*
[41] str.data.frame*          subset.data.frame
[43] summary.data.frame       t.data.frame
[45] tail.data.frame*         transform.data.frame
[47] unique.data.frame        unstack.data.frame*
[49] within.data.frame

   Non-visible functions are asterisked
```

Programmers can write methods specifically for the classes that they define. For example, let's see what methods are available for class `phylo` objects from the `ape` package:

```
> require(ape)
> methods(class="phylo")
```

```
 [1] AIC.phylo                   all.equal.phylo
 [3] as.hclust.phylo             as.matching.phylo
 [5] coalescent.intervals.phylo cophenetic.phylo
 [7] deviance.phylo              identify.phylo
 [9] logLik.phylo                makeLabel.phylo
[11] plot.phylo                  print.phylo
[13] reorder.phylo               skyline.phylo
[15] summary.phylo
```

10.3.1 No Validation

However, the S3 system also lacks features that object oriented languages often have, mostly related to a less structured language definition. For users, problems can arise when they accidentally make bad objects and crash their code. Under the S3 system, the class of an object is simply assigned via the class attribute.

```
> tree <- "This is not a phylogenetic tree"
> class(tree) <- "phylo"
> attributes(tree)
```

```
$class
[1] "phylo"
```

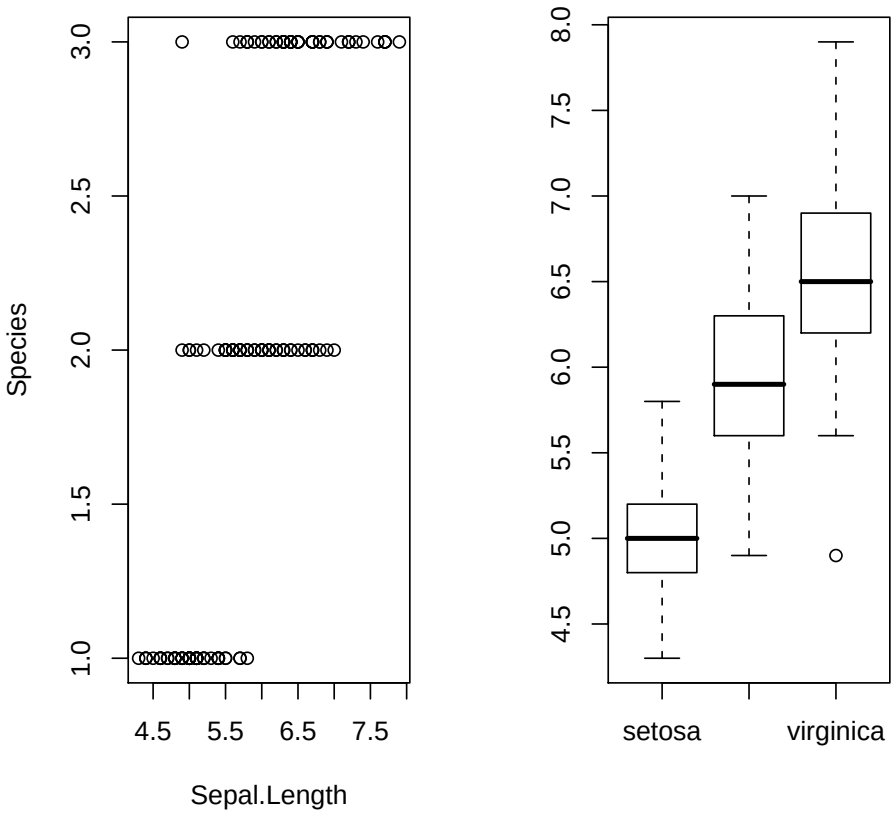
There is no validation or checking that the objects have appropriate contents. R just tries to do what it can with the crazy object. Worse things will happen if you try to plot this fake tree.

10.3.2 Methods dispatch

One of the beauties of methods is that the user doesn't have to worry about remembering the package-specific name of the function they want to use. They just use familiar generics such as `plot`, `summary`, and so on. The reason why this works is because of **methods dispatch**. The function looks at the class of the object that is given as its argument, and then it calls the correct function.

With S3 methods dispatch, however, it is rather clunky for programmers to write methods for every variation of parameters. As a consequence, sometimes you get the wrong method called. For example, here is something you may have experienced yourself. Take Fisher's famous iris dataset. You may have wanted a boxplot of some continuous character by species, but instead gotten a scatterplot:

```
> data(iris)
> op <- par(no.readonly = TRUE)
> par(mfrow=c(1,2))
> with(iris, plot(Sepal.Length, Species))
> with(iris, plot(Species, Sepal.Length))
> par(op)
```



The plot method dispatched a scatter plot when the first argument was a continuous variable, but dispatched a boxplot when the first argument was a factor. It is because R is trying to figure out what you want based on the supplied arguments. In this case, it is the first argument that determines the method, not the two arguments together.

10.4 S4 Classes

The S4 class system is considered to follow the principles of object oriented programming because:

1. New objects of any class must be specifically created using the constructor function `new()`. Programmers write a function to create new objects so that users never have to make calls to `new()`. Part of this process is **validation**, a series of checks that programmers write to make sure that object supplied by users are actually valid for that class. For users, this is a very nice feature that will prevent errors down the line which can be very confusing and difficult to trace.

2. Classes and methods can be inherited. This is mostly of benefit to programmers, but it is of benefit to users as well because there is a good chance that methods for one class of object can be used for related classes.

3. Programmers using S4 classes must write more consistent code, which makes it easier to extend packages and build a more functional family of packages. This consistency and predictability in the code makes it much easier for users to learn new packages.

Currently, `ouch` and `phylobase` are written in S4, and `ade4` will move to S4 in the next revision. More packages will soon follow.

10.4.1 What are the differences for users?

There are three main differences for users. We will illustrate these with examples in the next chapter on `phylobase`

1. Accessing help

2. Creating objects

3. Accessing internal elements of objects

Chapter 11

Phylobase

Some good starting points:

- `vignette('phylobase')`
- `?phylo4`

`phylobase` is a package whose development was started at the recent NESCent Hackathon on Comparative Methods in R. The hackathon was an event which brought programmers and users together to discuss integration of packages, including methods for data exchange, interoperability, and usability.

One clear need for the growing number of comparative methods packages was a common data format and utilities which would be useful for any comparative analysis.

11.1 Some Useful Features

tree input from Newick, Nexus, and other popular formats

coercion or translating from one package-specific format to another

combining trees with data functions to combine trees with data, matching on node labels (species and node names), or node numbers.

treewalking functions to get the user-specified label for a node or the internally generated node number for ancestors, descendants, parents (all ancestors), or children (all descendants), siblings (sister nodes), or MRCA (most recent common ancestor).

subsetting a very convenient function to select a subset of a phylogenetic object by specifying tips, or a clade, or the MRCA.

tree plotting nice tree and tree+data plotting facilities.

11.2 Accessing help

S4 help pages can still be called using the `?topic` syntax. In addition, there are some new features. Let's use a built-in example from `phylobase`:

```
> require(phylobase)
> data(geospiza)
> class(geospiza)

[1] "phylo4d"
attr(,"package")
[1] "phylobase"
```

We see that it is a `phylo4d` object. We can find out more about this class and the methods available for it:

```
> class ? phylo4d              # returns information on the class
> showMethods(class="phylo4d")  # lists all methods available for phylo4d
> method ? plot("phylo4d")   # the specific plot method for class phylo4d
> ? plot(geospiza)          # also returns plot method for phylo4d
```

The last one, the question mark in front of the entire plot call, is particularly nice because you don't have to specify (or know) the class of the object. The help function will figure it out.

11.3 Creating Objects

The basic objects in `phylobase` are the tree object called `phylo4` and the tree+data object called `phylo4d`. We can create new objects by calls to the `phylo4()` and `phylo4d()` constructors.

The other way if you have an existing tree in ape format `phylo`, is to coerce it to `phylo4d`. The standard way of doing this in S4 is to use the `as(object, "newclass")` function.

```
> load("Rdata/tree.primates.rda")
> tree4 <- as( tree.primates, "phylo4")
```

There are currently coercion functions to convert between phylobase `phylo4` to `phylo4d`, from phylobase to ape, from ape to phylobase, from phylobase to ade4, and from phylobase to data.frame.

11.4 Tree and Data Formats

11.4.1 phylo4

We can see the structure of a phylo4 object simply by:

```
> attributes(tree4)


$edge
     ancestor descendant
[1,]        6          7
[2,]        7          8
[3,]        8          9
[4,]        9          1
[5,]        9          2
[6,]        8          3
[7,]        7          4
[8,]        6          5

$edge.length
[1] 0.38 0.13 0.28 0.21 0.21 0.49 0.62 1.00

$Nnode
[1] 4

$tip.label
[1] "Homo"   "Pongo"  "Macaca" "Ateles" "Galago"

$node.label
[1] "N1" "N2" "N3" "N4"

$edge.label
[1] "E7" "E8" "E9" "E1" "E2" "E3" "E4" "E5"

$root.edge
[1] NA

$class
[1] "phylo4"
attr(,"package")
[1] "phylobase"
```

The structure is very similar to ape's `phylo` format, upon which it was modeled. The main features of importance to users is the edge matrix, the edge lengths, and the tip labels. The edge matrix contains a column of ancestor nodes and a column of descendant nodes. Together these define an "edge", and "edge.length" is the branch length associated with that particular edge. `tip.label` contains the species or taxon names for the terminal taxa. The other labels are optional, but are generated automatically if none are user-supplied.

The print and show methods for phylo objects show the tree as it would appear in data frame format, to make it easier for users to verify the tree since all of the nodes, branch lengths, and taxon names are lined up by row.

```
> tree4
```

```
   label node ancestor branch.length node.type
1     N1    6       NA            NA      root
2     N2    7        6          0.38  internal
3     N3    8        7          0.13  internal
4     N4    9        8          0.28  internal
5   Homo    1        9          0.21       tip
6  Pongo    2        9          0.21       tip
7 Macaca    3        8          0.49       tip
8 Ateles    4        7          0.62       tip
9 Galago    5        6          1.00       tip
```

11.4.2 phylo4d

The phylo4d format adds a data frame which contains the phenotypic data. The data are actually stored as two data frames, one for tip.data and one for node.data (typically NULL, but put in place for future use as a means for incorporating fossil data). Let's do an example:

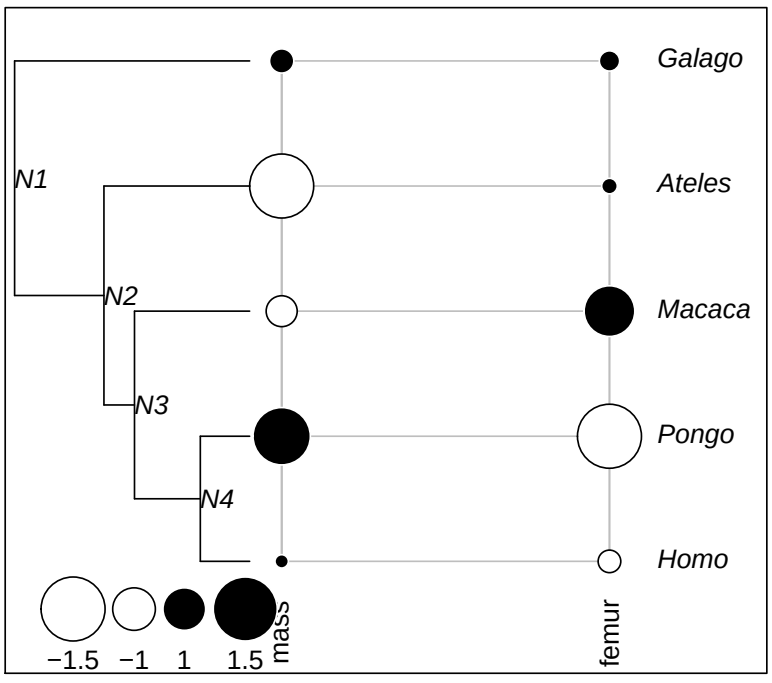
1. Make a dataframe of morphological data to match the primate tree
2. Name the rows with the tip.label (species names)
3. Use the phylo4d constructor to bind the tree and data together

```
> morph <- data.frame(row.names=labels(tree4), mass=rnorm(nTips(tree4),
+ mean=30, sd=10), femur=rnorm(nTips(tree4), mean=10, sd=5))
> tree4d <- phylo4d( tree4, tip.data=morph)
> tree4d
```

```
   label node ancestor branch.length node.type     mass      femur
1     N1    6       NA            NA      root       NA         NA
2     N2    7        6          0.38  internal       NA         NA
3     N3    8        7          0.13  internal       NA         NA
4     N4    9        8          0.28  internal       NA         NA
5   Homo    1        9          0.21       tip 29.36089  4.3744618
6  Pongo    2        9          0.21       tip 33.60660 -0.1852925
7 Macaca    3        8          0.49       tip 25.06429 12.4575003
8 Ateles    4        7          0.62       tip 21.87775  8.5941291
9 Galago    5        6          1.00       tip 30.40994  9.1082989
```

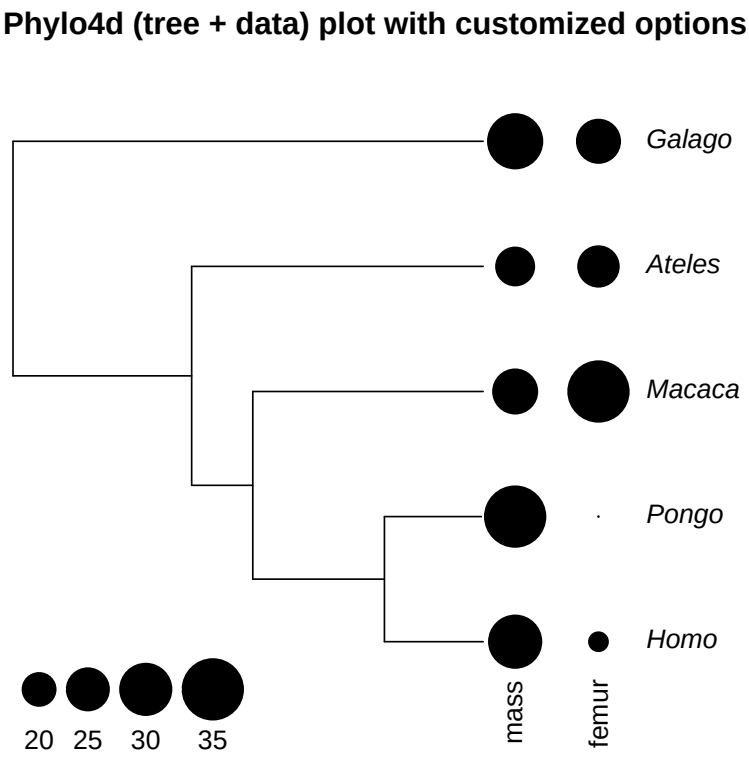
`phylo4d` objects are plotted in the same way as `phylo4`, but we can also add a bubble plot to indicate quantitative data.

```
> plot(tree4d)
> title("Phylo4d (tree + data) plot with default options")
```



There are many options which you can change to customize your plot. Type `?plot(tree4d)` to see the help page.

```
> plot(tree4d, center=F, scale=F, show.node.label=F, grid=F, ratio.tree=2/3, box=F)
> title("Phylo4d (tree + data) plot with customized options")
```



The `phylo4d` constructor also works directly from ape's `phylo` trees:

```
> phylo4d(tree.primates, tip.data=morph)
```

```
   label node ancestor branch.length node.type     mass      femur
1     N1    6       NA            NA      root       NA         NA
2     N2    7        6          0.38  internal       NA         NA
3     N3    8        7          0.13  internal       NA         NA
4     N4    9        8          0.28  internal       NA         NA
5   Homo    1        9          0.21       tip 29.36089  4.3744618
6  Pongo    2        9          0.21       tip 33.60660 -0.1852925
```

```
7 Macaca    3        8          0.49       tip 25.06429 12.4575003
8 Ateles    4        7          0.62       tip 21.87775  8.5941291
9 Galago    5        6          1.00       tip 30.40994  9.1082989
```

We can extract the data back from the `phylo4d` object using the `tdata` function:

```
> tdata(tree4d)
```

```
           mass      femur
Homo   29.36089  4.3744618
Pongo  33.60660 -0.1852925
Macaca 25.06429 12.4575003
Ateles 21.87775  8.5941291
Galago 30.40994  9.1082989
```

11.5 Accessing Internal Elements of S4 Objects

S4 classes are actually intended so that users do not need to know the internal structure of S4 objects. Rather, the programmer provides "accessor" functions to get at the data that users want. This is part of the concept of "abstraction". The reason behind distancing the user from the actual data object is so that developers can be free to change or modify the data structures without destroying the entire package, and breaking all the code that users have developed for their personal analyses. It is actually quite liberating and allows for greater flexibility for continued improvement after the initial design.

Accessor functions for phylobase include the following. A complete list is available by accessing the help for any of these individual functions (i.e., `?nTips`).

```
> nTips(tree4)    # the number of terminal taxa
```

```
[1] 5
```

```
> labels(tree4)           # tip (species) labels
```

```
[1] "Homo"   "Pongo"  "Macaca" "Ateles" "Galago"
```

```
> nNodes(tree4)  # number of internal nodes
```

```
[1] 4
```

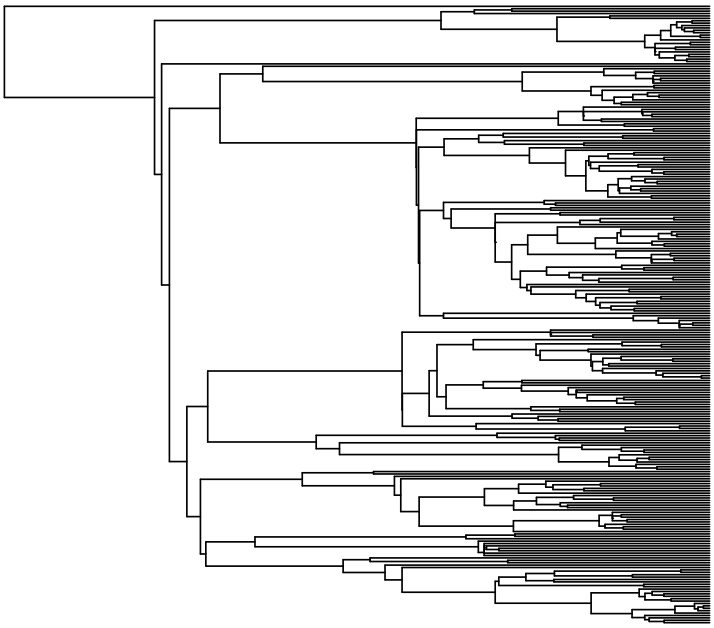
However, if you really want to access an internal element directly, the `$` operator which works for data frames and lists doesn't generally work for S4 objects (it actually works for `phylo4` objects because the developers wrote translation functions for them). Instead, internal elements are accessed using the `@` symbol. Usually, though, this is reserved for "internal" programming, such as coding the accessor functions.

11.6 Subsetting

Phylobase has a number of nice subsetting features. They extract portions of phylogenetic trees and their associated data. The subset can be specified by a vector of tips to include or exclude, or the most recent common ancestor of a group of nodes. We can also extract a subtree of a given node.

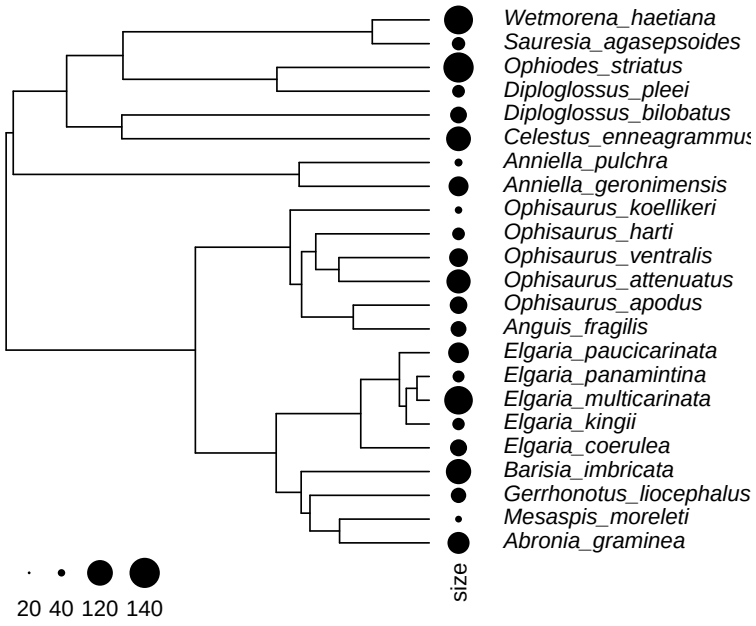
For this example, let's use a larger phylogeny of squamates (lizards). We plot without tip labels because there are just too many taxa.

```
> tree4 <- NexusToPhylo4("Data/squamatetree1.nex")
> plot(tree4, show.tip.label=F)
```



Let's prune the tree to the MRCA of *Wetmorena haetiana* and *Abronia graminea*. To show test the subsetting on `phylo4d` objects, let's make up some data and bind it to the tree:

```
> smalltree4 <- subset(tree4, mrca=c( "Wetmorena_haetiana", "Abronia_graminea"))
> smalltree4d <- phylo4d(smalltree4, tip.data=data.frame(
+ size=rnorm(nTips(smalltree4), mean=85, sd=25), row.names=labels(smalltree4)))
> plot(smalltree4d, center=F, scale=F, show.node.label=F, grid=F, ratio.tree=.6,
+ box=F, cex.symbol=.5, cex.label=.8)
```



We had to reduce the bubble `cex.symbol` and text `cex.label` sizes to fit the space for the plot.

Suppose we were only interested in plotting the genera *Diploglossus*, *Ophisaurus*, and *Elgaria*. We can combine this with the `grep` function which matches patterns in strings (i.e., it does partial matching).

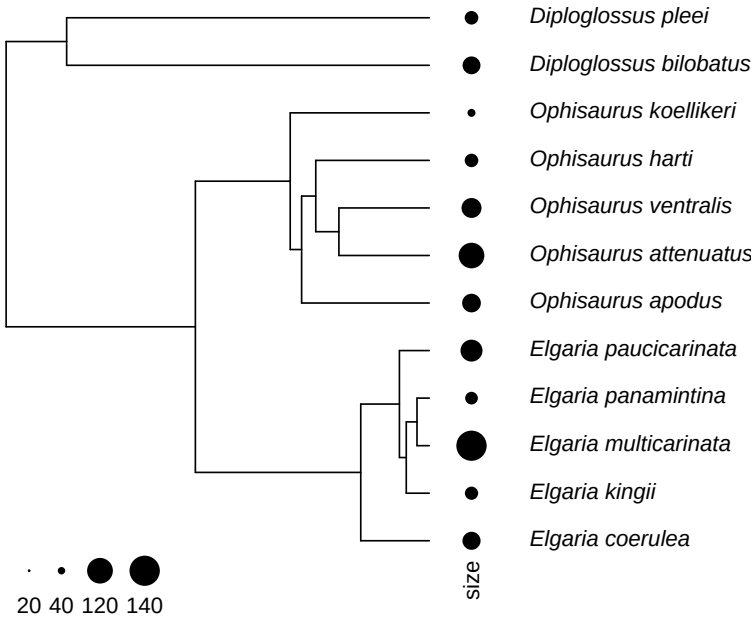
```
> include <- c( grep("Ophisaurus", labels(smalltree4), value=T),
+                 grep("Diploglossus", labels(smalltree4), value=T),
+                 grep("Elgaria", labels(smalltree4), value=T))
> include
```

```
 [1] "Ophisaurus_apodus"      "Ophisaurus_attenuatus"
 [3] "Ophisaurus_ventralis"   "Ophisaurus_harti"
 [5] "Ophisaurus_koellikeri"  "Diploglossus_bilobatus"
 [7] "Diploglossus_pleei"     "Elgaria_coerulea"
 [9] "Elgaria_kingii"         "Elgaria_multicarinata"
[11] "Elgaria_panamintina"    "Elgaria_paucicarinata"
```

```
> smalltree4d <- subset(smalltree4d, tips.include=include)
```

This time, let's also remove the underscore from the names by using the `sub` function (related to grep):

```
> labels(smalltree4d) <- sub("_", " ", labels(smalltree4d))
> labels(smalltree4) <- sub("_", " ", labels(smalltree4))
> plot(smalltree4d, center=F, scale=F, show.node.label=F, grid=F, ratio.tree=.6,
+ box=F, cex.symbol=.5, cex.label=.8)
```



We can also subset by using node numbers or species names (in quotes) with the square bracket operator. These are all equivalent:

```
> smalltree4d[1:3]    # tips to include
> smalltree4d[-(4:23)]   # tips to exclude
> smalltree4d[c("Elgaria coerulea", "Elgaria kingii", "Elgaria multicarinata")]
> inc <- c("Elgaria coerulea", "Elgaria kingii", "Elgaria multicarinata")
> smalltree4d[inc]
```

Note: Trees often have taxon labels with underscores. `ape` plots underscores as blank in text fields by default.

Ape does not currently have tree+data objects, and phylobase does not have actual comparative methods functions such as independent contrasts. But it can still be useful to bind the data to the tree, perform tree manipulations, and export the tree and data formats (in the proper order) to ape objects. Future developments should bring increased functionality.

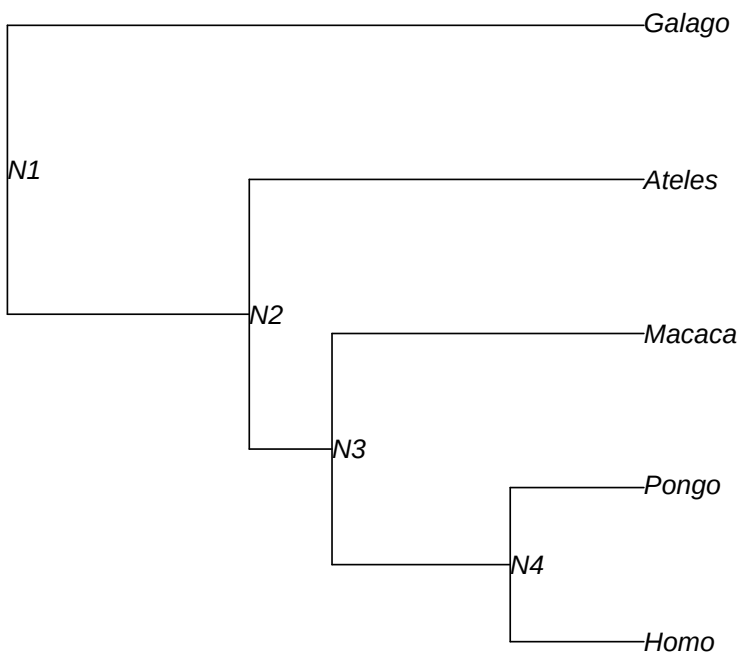
```
> tree.ape <- as(smalltree4d, "phylo")
> tree.ape.dat <- tdata(smalltree4d)
```

11.7 Treewalking

Other ways to get descendants (immediate or all) or ancestors and subtrees is by using the treewalking functions. Find help by typing `?ancestors`.

The `which` parameter in `ancestors` specifies whether to return just direct ancestor ("parent") or "all" ancestor nodes; in `descendants`, specify whether to return just direct descendants ("children"), all extant descendants ("tips"), or all descendant nodes ("all").

```
> tree4 <- as(tree.primates, "phylo4")
> plot(tree4, show.node.label=T)
```



Suppose we wanted to label the node leading to Pongo and Homo with the word "apes". There are several ways we could access the node in question:

```
> mm <- MRCA(tree4, c("Homo", "Pongo"))
> mm

N4
 9


> ancestor(tree4, "Homo")

N4
 9
```

Now we find which node label we want, replace that label, erase the rest, and plot:

```
> ii <- which(nodeLabels(tree4)==names(mm))
> nodeLabels(tree4)[-ii] <- ""
> nodeLabels(tree4)[ii] <- "Great Apes"
> plot(tree4, show.node.label=T)
```

Finally, `getnodes` is a handy function to identify the number of the node if you have the label, or vice versa.

```
> getnodes(tree4, "Great Apes")
> getnodes(tree4, 9)
```

11.8 Example: Generating a set of trees with simulated branch lengths

Suppose we wanted to test the robustness of our conclusions on error in the branch length estimates for our tree. But how to generate the branch lengths? We come up with two ideas: (1) Draw branch lengths from a probability distribution, say the normal distribution with mean and standard deviation from the observed branch lengths. What this procedure implies is that each branch length is an identical draw from the same distribution (as if the branching events result from a single uniform process, although with noise). This is not particularly biologically reasonable, but it is something we can do. (2) Assume that the branch lengths are reasonable estimates of the times between branching events, but they are sampled with some error. This implies that if the process were repeated many times, you each particular branch would have its own mean. All we have to estimate this mean is the observed branch itself, so let's take it as our estimate. We have no information on the standard deviation, so let's try a common standard deviation of 1/4 of the grand mean. (3) Of course, if we have a series of branch length estimates (say from PAUP or other phylogeny estimation program), we could simply try that. But for demonstration purposes, let's try (1) and (2)

11.8.1 Branch lengths drawn from a common distribution

Using the primate tree from above, we generate a set of branch lengths, using the `phylo4` accessors to get the information from the tree

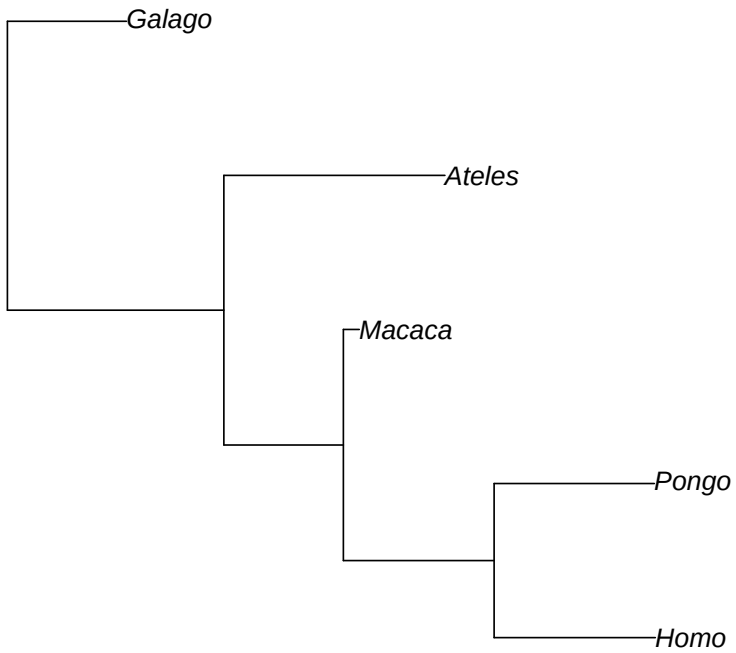
```
> bl <- rnorm(nEdges(tree4), mean=mean(edgeLength(tree4)), sd=sd(edgeLength(tree4)))
> bl

[1] 0.6463380 0.3569145 0.4500820 0.4814310 0.4783944 0.0474164
[7] 0.6601589 0.3555261
```

One problem that you may see is that it is very possible (in fact easy) to get negative branch lengths. One solution is to simply discard those sets of bl that have negatives. One could also take the absolute value (reflect the negatives) or set them to zero. Any of these solutions will change the distribution (the latter two making it either a reflective or an absorbing boundary), but the first option seems to be the least damaging.

```
> while(any(bl < 0)) bl <- rnorm(nEdges(tree4), mean=mean(edgeLength(tree4)),
+ sd=sd(edgeLength(tree4)))
```

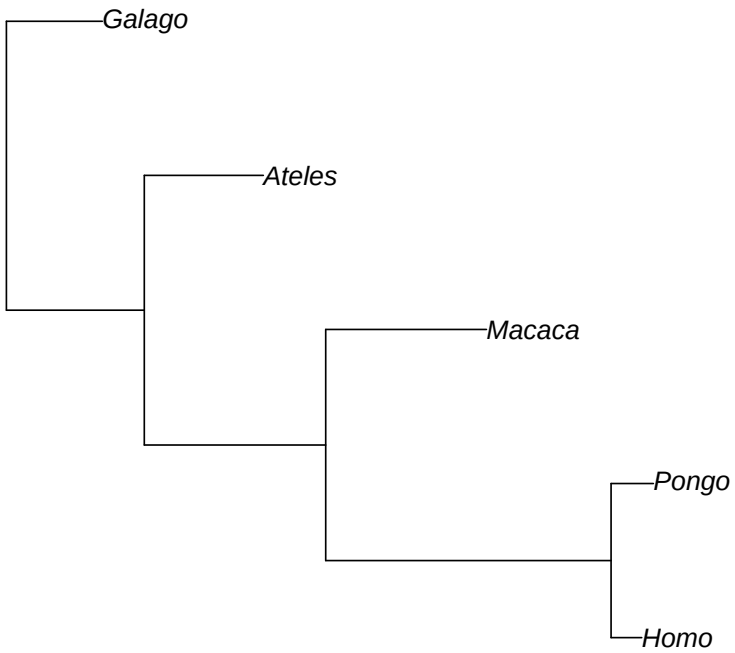
Now we need to make a new tree with the phylo4 constructor. For clarity in this example, lets first save the information from `tree4` into separate variables:

```
> nodes <- edges(tree4)
> species <- labels(tree4)
> tr <- phylo4(edge=nodes, edge.length=bl, tip.label=species)
> plot(tr)
```



We could make this into functions, especially if we were planning on using it in other code. We have two tasks here (1) generating the branch lengths, and (2) making the trees with the new branch lengths, so let's make two functions:

```
> gen.bl <- function(tt) {
+     bl <- -1
+     while(any(bl < 0)) bl <- rnorm(nEdges(tt), mean=mean(edgeLength(tt)),
+     sd=sd(edgeLength(tt)))
+ }
> change.bl <- function (tt, bl) {
+     nodes <- edges(tt)
+     species <- labels(tt)
+     return( phylo4(edge=nodes, edge.length=bl, tip.label=species))
+ }
> branchlengths <- gen.bl(tree4)
> simtree <- change.bl(tree4, branchlengths)
> plot(simtree)
```



Voila! You can now create as many trees with weird branch lengths as you like. Just rerun the calls to `gen.bl` and `change.bl`. For example, if you wanted to generate a list of 9 trees, you could use a loop. First you must initialize the output tree list (or else you will get an error when you try to save a value to an element of the list in the loop):

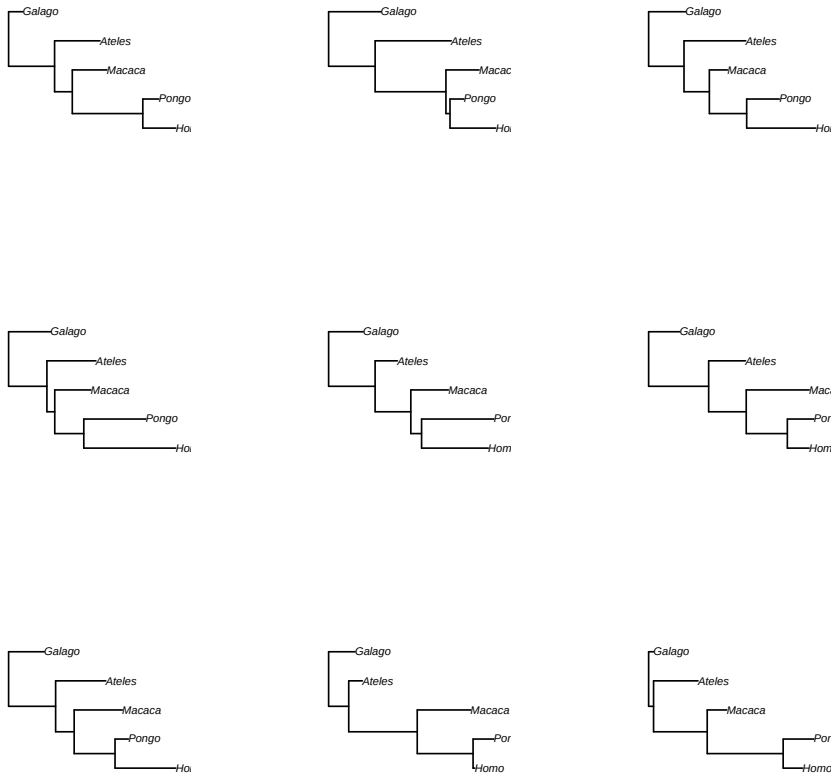
```
> simtrees <- vector(mode='list')
> for (i in 1:9) {
+   bl <- gen.bl(tree4)
+   simtrees[[i]] <- change.bl(tree4, bl)
+ }
```

Another strategy which is often very useful for data analysis is to use apply functions to generate a list of trees, each with randomized branch lengths. The function can be embedded in the apply call. Since we want to generate a list and will input a list of branch lengths, let's use `lapply`.

```
>                 # generate the list of branch lengths
> bls <- lapply(1:9, function(x) gen.bl(tree4))
>                 # generate the trees with the new branch lengths
> simtrees2 <- lapply(bls, function(x) {
+       return( phylo4(edge=edges(tree4), edge.length=x, tip.label=labels(tree4)))
+
+       })
```

We can plot the first nine of our crazy trees (the `ll <-` is just a kludge to supress output from the lapply):

```
> op <- par(no.readonly = TRUE)
> par(mfrow=c(3,3))
> ll <- lapply(simtrees2, plot)
> par(op)
```



11.8.2 Branch lengths drawn from normal distributions with separate means

The key difference between these two options is the generation of the branch lengths. So all we need to do is change the `gen.bl` function. Now instead of eight draws from the same distribution, for each tree, we want a single draw from eight distributions (one for each branch). We will assume that they are normal distributions with mean at the observed branch lengths, and standard deviations arbitrarily chosen as 25% of the mean of the observed branch lengths, which turns out to be approximately 0.1. We also change the input from the whole tree (a phylo4 object) to a vector of branch lengths:

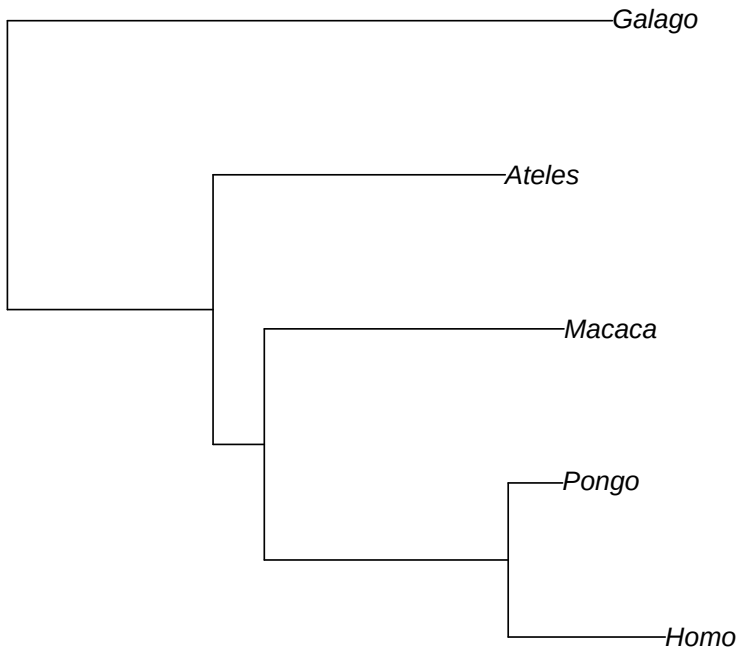
We will use the `sapply` function because we want to use `rnorm` once for each branch:

```
> bl.stan.dev <-  mean(edgeLength(tree4))*.25
> gen.bl <- function(bl, blsd) {
+     bl <- sapply( bl, function(x) rnorm(1, mean=x, sd=blsd))
+     while(any(bl < 0)) bl <- sapply( bl, function(x) rnorm(1, mean=x, sd=blsd))
```

```
+     return(bl)
+ }
```

We can reuse the function `change.bl` that we wrote previously, substitute our new branch length generating function and plot our new tree:

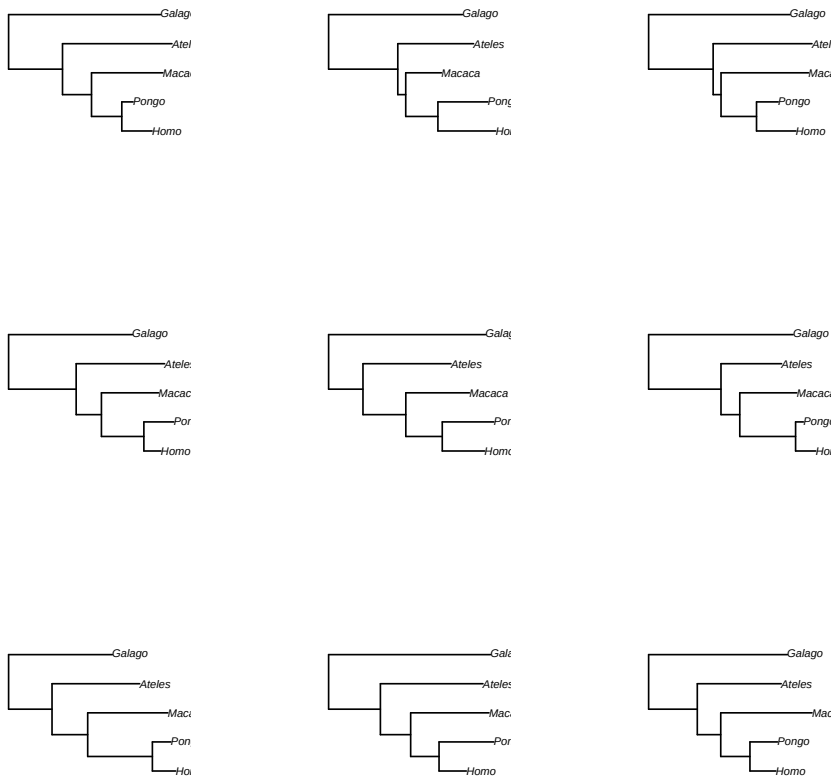
```
> plot( change.bl(tree4, gen.bl(edgeLength(tree4), bl.stan.dev)))
```



Let's redo our nine trees:

```
>                 # generate the list of branch lengths
> bls <- lapply(1:9, function(x) gen.bl(edgeLength(tree4), bl.stan.dev))
>                 # reuse our lapply code
> simtrees3 <- lapply(bls, function(x) {
+       return( phylo4(edge=edges(tree4), edge.length=x, tip.label=labels(tree4)))
+       })
> op <- par(no.readonly = TRUE)
```

```
> par(mfrow=c(3,3))
> ll <- lapply(simtrees3, plot)
> par(op)
```

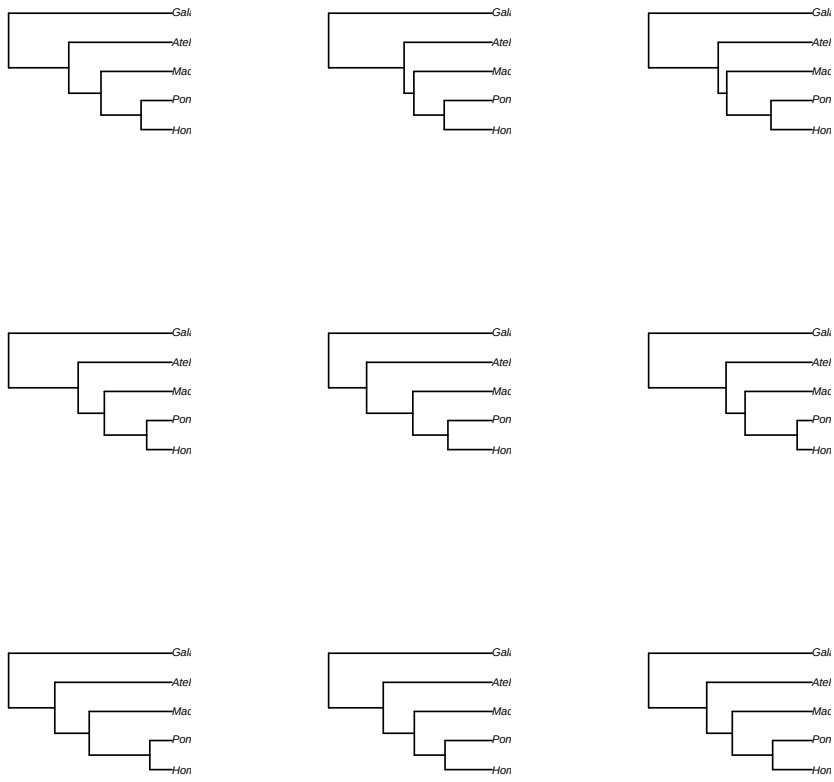


You can see that the two methods of generating branch lengths generate different shapes of trees, especially depending on the magnitude of the standard deviation you allow the second method to have. In the former case, it is difficult to relate the simulated branch lengths as reflective of time (or another way to think about this is that implicitly assumes we have no idea when branching events occurred, but the timing between them are assumed equal with some variance). In the latter case, it could be interpreted as an assumption that our observed branch lengths reflect time, but there is variance or measurement error in estimating the branch lengths.

In either case, you can now easily export these trees to `ape` to use in comparative analyses. Suppose we needed to use a comparative method that required ultrametric trees. Use the `chronogram`

```
> require(ape)
```

```
> simtrees3.ape <- lapply(simtrees3, as, "phylo")   # coerce to phylo:
>                               #same as nine calls to as(simtrees[[1]], "phylo")
> ultra3.ape <- lapply(simtrees3.ape, chronogram)  # make the trees ultrametric
> op <- par(no.readonly = TRUE)
> par(mfrow=c(3,3))
> ll <- lapply(ultra3.ape, plot)         # plot the nine trees
> par(op)                               # reset to default plot parameters
```



Chapter 12

Introduction to OU Models

Goals:

- Approaches for adaptive evolution (ouch, slouch, others)
- Model-based vs statistical approaches

Concepts:

- Model comparison tools in R
- Process-based models

12.1 The OU Model for Comparative Analysis

Recall that we described a Brownian motion process using the following equation:

$$dX(t) = \sigma\, dB(t). \tag{12.1}$$

If we imagine the phenotype X as changing through time t, this equation says that in a small increment of time, the change will be proportional to the parameter σ. Here, $dB(t)$ is a sample from a Brownian (white noise) process.

A small step towards reality is the OU Process:

$$dX(t) = \alpha\,(\theta - X(t))\, dt + \sigma\, dB(t). \tag{12.2}$$

Eq. 14.2 expresses the amount of change in character X over the course of a small increment of time: specifically, $dX(t)$ is the infinitesimal change in the character X over

the infinitesimal interval from time t to time $t + dt$. The term $dB(t)$ is "white noise"; that is, the random variables $dB(t)$ are independent and identically-distributed normal random variables, each with mean zero and variance dt. The parameter α measures the strength of selection. When $\alpha = 0$, the deterministic part of the OU model drops out and (14.2) collapses to the familiar BM model of pure drift,

12.2 Introduction to Likelihood

12.3 `ouch`

See `ouch` lecture.

Good starting points:

?bimac help page for *Bimaculatus* character displacement dataset

example(bimac) example of bimac analysis

?anolis.ssd help page for *Anolis* sexual size dimorphism dataset

`ouch` is a package designed to test adaptive hypotheses using variations of the OU process, including BM. OUCH implements a model that fits an alpha and sigma parameters to the entire phylogeny, but allows the user to specify which branches belong to different selective regimes. The location of the optima are also fit.

12.3.1 The Data

The data in OUCH are most easily assembled as a data frame. Load the built in example from ouch and then print it to the screen (I only printed the head of the dataset here):

```
> require(ouch)
> data(bimac)
> bimac
```

```
  node species size ancestor time OU.1   OU.3 OU.4  OU.LP
1    1    <NA>   NA       NA    0   ns medium  anc medium
2    2    <NA>   NA        1   12   ns medium  anc medium
3    3    <NA>   NA        2   32   ns medium  anc  small
4    4    <NA>   NA        3   34   ns medium  anc  small
5    5    <NA>   NA        4   36   ns medium  anc  small
6    6    <NA>   NA        3   36   ns medium  anc  small
```

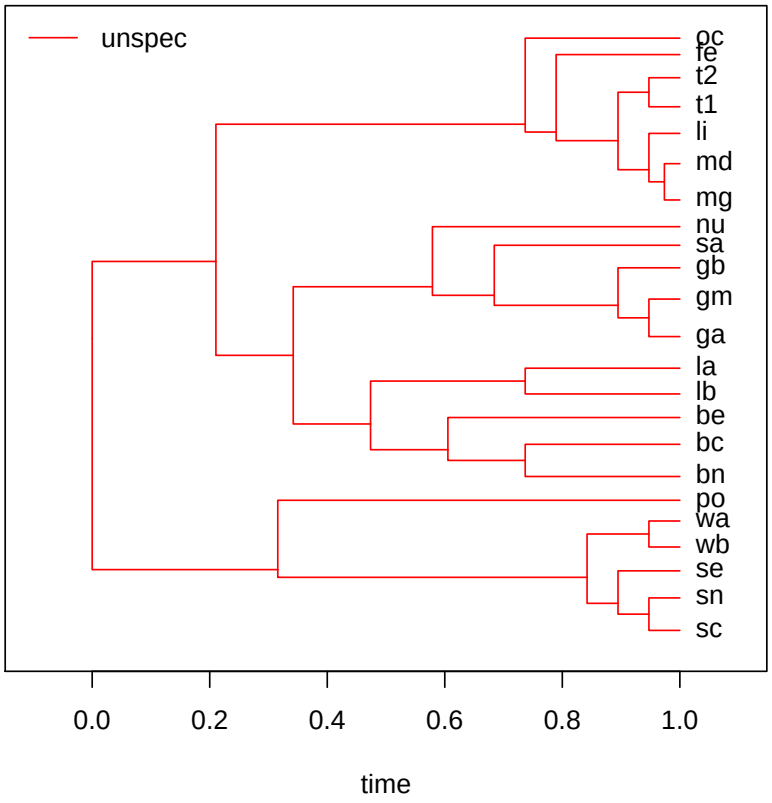
NOTE: a very important detail about `ouch` is that it matches trees with data and regimes using the node labels stored in the rownames of the objects you pass to the `ouch` functions. So it is important to make sure that your dataframes and vectors are appropriately named. The dataframe bimac already has the correct row names, but we do so here just to illustrate.

```
> rownames(bimac) <- bimac$node
```

`ouch` was designed around a rectangular data model, so although the tree object is not a dataframe internally, it still helps us to build the data as a dataframe before making the `ouchtree` objects. The central organizing element is the `node`: it has a node number (usually an integer but it is actually a unique character string), an `ancestor` to which it is joined by a branch, a `time` since the root of the tree, and optional `label` such as a species name. The hypotheses which we use are assigned by "painting" particular regimes on branches. It is convenient to represent each model or hypothesis as a column on the dataframe, with the regime assigned to the node (that is, it is assigned to the branch connecting the node to its ancestor).

Make an `ouchtree` object using the `ouchtree` constructor. `with` is a very nice function to create a small local environment so that you can use a dataframe's elements directly without using the `bimac$` prefix. It is similar to an `attach` but it is temporary – only lasting as long as the call itself. I like it much better than `attach` because I sometimes forget what I've attached and run into problems later. Also, with `attach`, you are actually working with a copy of the original dataframe object, so updating values is trickly. With `with`, it is more clear what's going on, and I don't tend to make those mistakes.

```
> tree <- with(bimac, ouchtree(node,ancestor,time/max(time),species))
> plot(tree)
```



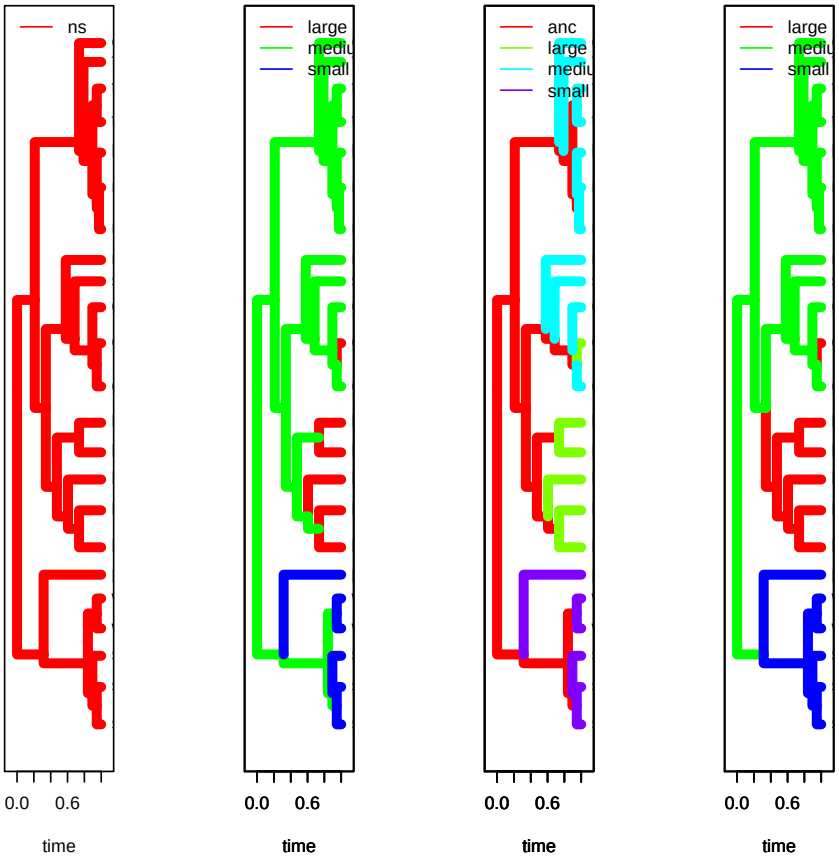
`ouch` fits the OU model Eq. 14.2 along each branch of the phylogeny. While α and σ are held constant across the entire tree, the optima along each branch θ are allowed to vary. Users can then "paint" various combinations of optima on the tree to reflect various biological scenarios.

For example, the dataset `bimac` was used to test the hypothesis of character displacement using an interspecific daaset of body sizes and sympatry/allopatry Butler and King (2004). The analysis tested several different models, which are included with `bimac`. They are: "OU.1" or global optimum, "OU.3" or small, medium, and large regimes depending on the body size of the observed species (terminal branches only, internal branches painted "medium", "OU.4" or the same as "OU.3" but with internal branches given their own unique regime called "ancestral", and "OU.LP" based on a linear parsimony reconstruction of the colonization events (i.e., that as species came into sympatry, they diverged in body size).

12.3.2 Plotting ouchtrees

You can plot the regime paintings on the tree, and set options such as line widths for prettier plots. ouch has a very nice feature which allows plotting of the alternative models on one plot.

```
> plot(tree, regimes=bimac[c("OU.1", "OU.3", "OU.4", "OU.LP")], lwd=6)
```



Remember that you can pass a single vector or a data frame to the regimes parameter, but it must have the appropriate row names or names in the case of a vector. The regimes are not part of the ouchtree object, because they represent our hypothesis of evolution along the tree, rather than the tree itself. It is part of the original dataframe from which we derived the tree, so remember to refer to bimac when passing the regimes to the plot function.

12.3.3 Fitting models

There are two main model fitting functions in `ouch`, `brown`, which fits Brownian motion models, and `hansen`, which fits OU models to comparative data. The call to `brown` is particularly simple, as it takes only the data and the tree:

```
> brown(log(bimac['size']),tree)
```

```
call:
brown(data = log(bimac["size"]), tree = tree)
   nodes ancestors     times labels     size
1      1      <NA> 0.0000000   <NA>       NA
2      2         1 0.3157895   <NA>       NA
3      3         2 0.8421053   <NA>       NA
4      4         3 0.8947368   <NA>       NA
5      5         4 0.9473684   <NA>       NA
6      6         3 0.9473684   <NA>       NA
7      7         1 0.2105263   <NA>       NA
8      8         7 0.3421053   <NA>       NA
9      9         8 0.4736842   <NA>       NA
10    10         9 0.6052632   <NA>       NA
11    11        10 0.7368421   <NA>       NA
12    12         9 0.7368421   <NA>       NA
13    13         8 0.5789474   <NA>       NA
14    14        13 0.6842105   <NA>       NA
15    15        14 0.8947368   <NA>       NA
16    16        15 0.9473684   <NA>       NA
17    17         7 0.7368421   <NA>       NA
18    18        17 0.7894737   <NA>       NA
19    19        18 0.8947368   <NA>       NA
20    20        19 0.9473684   <NA>       NA
21    21        20 0.9736842   <NA>       NA
22    22        19 0.9473684   <NA>       NA
23    23         2 1.0000000     po 2.602690
24    24         4 1.0000000     se 2.660260
25    25         5 1.0000000     sc 2.660260
26    26         5 1.0000000     sn 2.653242
27    27         6 1.0000000     wb 2.674149
28    28         6 1.0000000     wa 2.701361
29    29        10 1.0000000     be 3.161247
30    30        11 1.0000000     bn 3.299534
31    31        11 1.0000000     bc 3.328627
32    32        12 1.0000000     lb 3.353407
```

```
33    33        12 1.0000000     la 3.360375
34    34        13 1.0000000     nu 3.049273
35    35        14 1.0000000     sa 2.906901
36    36        15 1.0000000     gb 2.980619
37    37        16 1.0000000     ga 2.933857
38    38        16 1.0000000     gm 2.975530
39    39        17 1.0000000     oc 3.104587
40    40        18 1.0000000     fe 3.346389
41    41        20 1.0000000     li 2.928524
42    42        21 1.0000000     mg 2.939162
43    43        21 1.0000000     md 2.990720
44    44        22 1.0000000     t1 3.058707
45    45        22 1.0000000     t2 3.068053

sigma squared:
           [,1]
[1,] 0.04311003

theta:
NULL
   loglik  deviance       aic     aic.c       sic       dof
 17.33129 -34.66257 -30.66257 -30.06257 -28.39158   2.00000
```

What is returned is an object of class `browntree`. It contains all input including the function call, the tree and data), as well as the parameter estimate for σ and the model fit statistics including: the log-likelihood, the deviance ($-2 * log(L)$), the information criteria AIC, AIC_c (corrected for small sample size), and SIC, and the model degrees of freedom.

It is a good practice to save this, as it encapsulates the analysis. From this, we can rerun the model fit.

```
> h1 <- brown(log(bimac['size']),tree)
```

`hansen` models are slightly more complex. In addition to σ, we are now fitting α, the strength of selection, and all of the optima θ specified by our model. This maximum-likelihood search now requires an initial guesses. If you have no idea, a good starting guess is 1. If you want to be sure, you can intiate searches with different starting guesses. You can also specify alternative optimization algorithms and increase or decrease the relative tolerance, which is the stringency by which convergence is assessed. Typically, the default is roughly `reltol=1e-8`, and the limit of machine precision is in the neighborhood of `reltol=1e-15`.

```
> h2 <- hansen(log(bimac['size']),tree,bimac['OU.1'],alpha=1,sigma=1)
> h3 <- hansen(log(bimac['size']),tree,bimac['OU.3'],alpha=1,sigma=1)
```

```
> h4 <- hansen(log(bimac['size']),tree,bimac['OU.4'],alpha=1,sigma=1)
> h5 <- hansen(log(bimac['size']),tree,bimac['OU.LP'],alpha=1,sigma=1,reltol=1e-5)
```

12.3.4 hansentree and ouchtree methods

We can see the model results by typing `h5`, which will execute the `print` method for this class. You could also use the `attributes` function, but this will dump too much information. `ouchtree` objects and the classes derived from them contain information that is used in internal calculations of the algorithms, not of general interest to users.

Additional accessor functions include:

```
> coef(h5)    # the coefficients of the fitted model


$alpha
[1] 1.616580

$sigma
[1] 0.2249274

$theta
$theta$size
   large   medium    small
3.355087 3.040729 2.565249


$alpha.matrix
        [,1]
[1,] 2.61333

$sigma.sq.matrix
           [,1]
[1,] 0.05059232


> logLik(h5)   # the log-likelihood value


[1] 24.81823


> summary(h5)   # everything in the print method except the tree+data
```

We can now generate a table of our model fits:

```
> unlist(summary(h5)[c('aic', 'aic.c', 'sic', 'dof')])  # just the model fit statistics

      aic     aic.c       sic       dof
-39.63645 -36.10704 -33.95898   5.00000

>                                    # on a single line
> h <- list(h1, h2, h3, h4, h5)   # store all our fitted models in a list
> names(h) <- c("BM", "OU.1", "OU.3", "OU.4", "OU.LP")
> sapply( h, function(x) unlist(summary(x)[c('aic', 'aic.c', 'sic', 'dof')]) )

             BM      OU.1      OU.3      OU.4     OU.LP
aic   -30.66257 -25.39364 -29.15573 -35.22319 -39.63645
aic.c -30.06257 -24.13048 -25.62631 -29.97319 -36.10704
sic   -28.39158 -21.98715 -23.47826 -28.41022 -33.95898
dof     2.00000   3.00000   5.00000   6.00000   5.00000
```

By storing the model fits in a list, we can use apply methods to get the statistics from all the models at once. `sapply` returns a matrix if possible.

```
> h.ic <- sapply( h, function(x) unlist(summary(x)[c('aic', 'aic.c', 'sic', 'dof')]) )
> print( h.ic, digits = 3)

         BM  OU.1  OU.3  OU.4 OU.LP
aic   -30.7 -25.4 -29.2 -35.2 -39.6
aic.c -30.1 -24.1 -25.6 -30.0 -36.1
sic   -28.4 -22.0 -23.5 -28.4 -34.0
dof     2.0   3.0   5.0   6.0   5.0
```

Simulation and bootstrap methods: `simulate` generates random deviates or sets of simulated tip data based on the fitted model. The input is a fitted model `hansentree` or `browntree`, and the output is a list of dataframes, each comparable to the original data. These can then be used to refit the model.

```
> h5.sim <- simulate(object = h5, nsim=10)   # saves 10 sets of simulated data
```

`update` refits the model, with one or more parameters changed.

```
> summary( update( object = h5, data = h5.sim[[1]] ) )   # fit the first dataset
```

```
$call
hansen(data = data, tree = object, regimes = regimes, alpha = alpha,
    sigma = sigma)

$conv.code
[1] 0

$optimizer.message
NULL

$alpha
         [,1]
[1,] 4.723626

$sigma.squared
           [,1]
[1,] 0.07426371

$optima
$optima$size
   large   medium    small
3.314692 3.035894 2.760833


$loglik
[1] 25.15094

$deviance
[1] -50.30189

$aic
[1] -40.30189

$aic.c
[1] -36.77248

$sic
[1] -34.62442

$dof
[1] 5
```

```
> h5.sim.fit <- lapply( h5.sim, function(x) update(h5, x))  # fit all 10 simulations
```

bootstrap is a convenience function for generating parametric bootstraps of the parameter estimates. It takes the fitted model, performs the simulations, refits, and outputs a dataframe of parameter estimates.

```
> bootstrap( object = h5, nboot=10)
```

```
       alpha sigma.squared optima.size.large optima.size.medium
1   6.050536    0.11986853          3.257964           2.893158
2   4.387984    0.04847116          3.290218           3.024340
3  10.969987    0.11259006          3.290983           3.100471
4   2.460918    0.03987607          3.332102           3.092758
5  11.982402    0.21181703          3.229702           2.969486
6   3.845293    0.04486152          3.365991           3.081710
7   2.525458    0.03840842          3.428682           3.024083
8   2.899610    0.05382205          3.351515           2.998701
9   4.047742    0.05340101          3.389197           3.029283
10  7.480043    0.06112392          3.262765           3.072929
   optima.size.small   loglik       aic     aic.c       sic dof
1           2.596582 21.88124 -33.76249 -30.23308 -28.08502   5
2           2.588191 29.41874 -48.83747 -45.30806 -43.16000   5
3           2.707631 28.49128 -46.98256 -43.45315 -41.30509   5
4           2.529705 27.12678 -44.25357 -40.72415 -38.57610   5
5           2.658216 22.15156 -34.30312 -30.77371 -28.62565   5
6           2.632919 29.19929 -48.39857 -44.86916 -42.72110   5
7           2.615850 27.73924 -45.47848 -41.94906 -39.80100   5
8           2.583068 24.87212 -39.74424 -36.21483 -34.06677   5
9           2.676901 27.62122 -45.24243 -41.71302 -39.56496   5
10          2.727930 31.64728 -53.29456 -49.76515 -47.61709   5
```

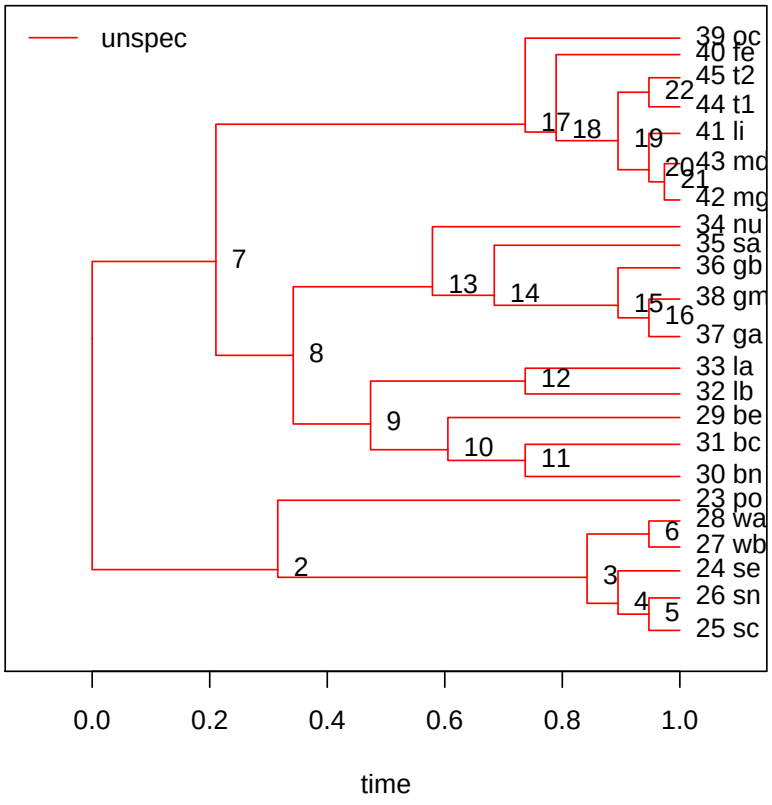
12.3.5 painting regimes on trees

A new function in ouch is paint. Previously, it was up to users to set up regimes manually by editing spreadsheets. paint helps with this task by specifying the regimes on particular species, subtrees, or particular branches.

There are two parameters to paint, subtrees, which paints the entire subtree which descends from the node, and branch, which paints the branch connecting the node to it's ancestor. For either, you specify the node label (remember it's a character and needs to be quoted), and set it equal to the name of the regime you want to specify.

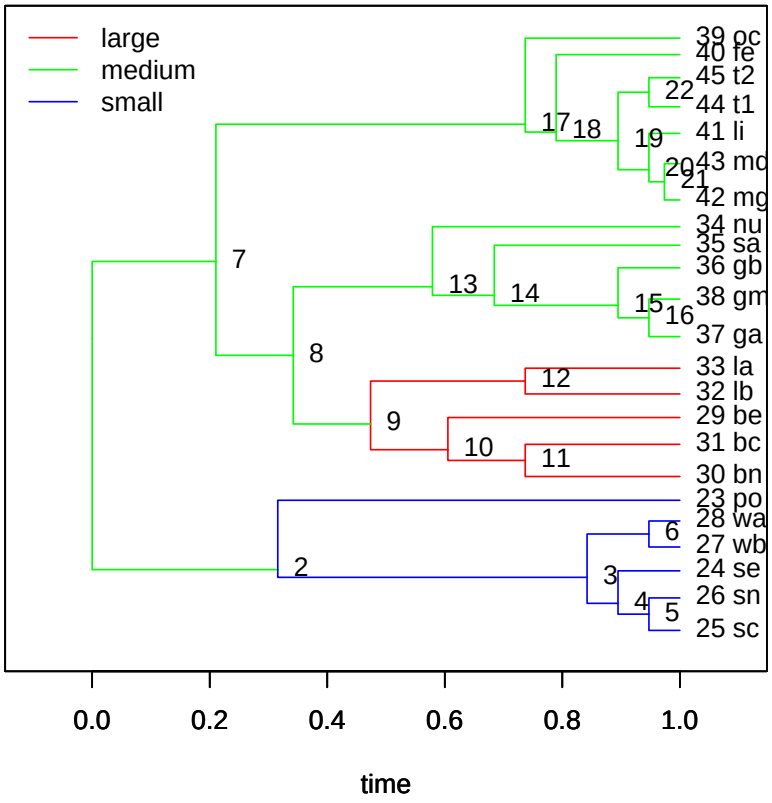
Let's try it on the bimac tree and try to recreate the OU.LP regime:

```
> plot(tree, node.names=T)
```



Paint the subtrees first, take a look:

```
> ou.lp <- paint( tree, subtree=c("1"="medium","9"="large","2"="small") )
> plot(tree, regimes=ou.lp, node.names=T)
```

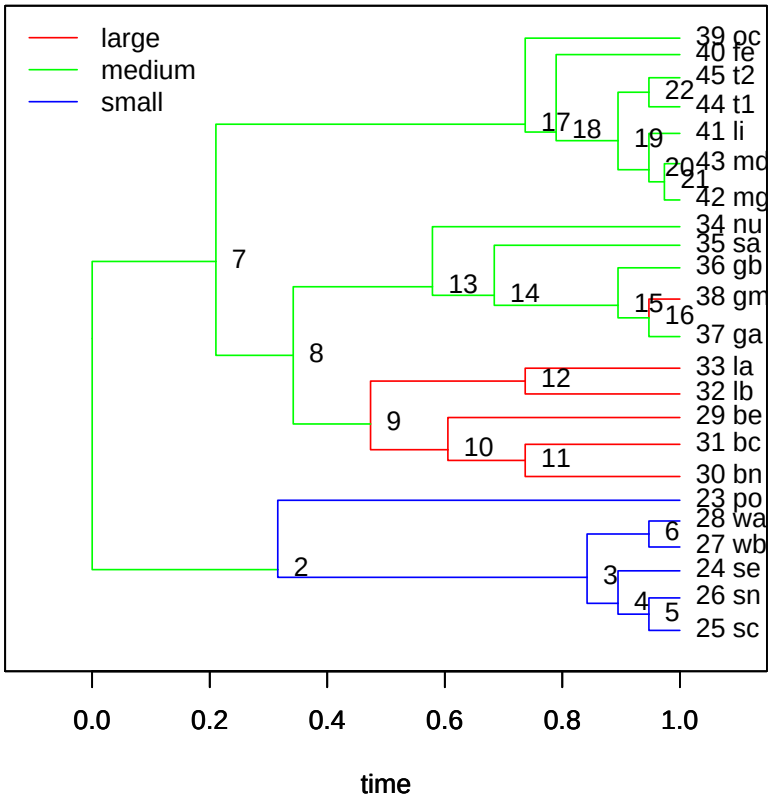


But there was an independent switch from medium to large at species "gm", or node "38", and the node connecting "9" to its ancestor:

```
> ou.lp <- paint( tree, subtree=c("1"="medium","9"="large","2"="small"),
+ branch=c("38"="large","2"="medium"))
```

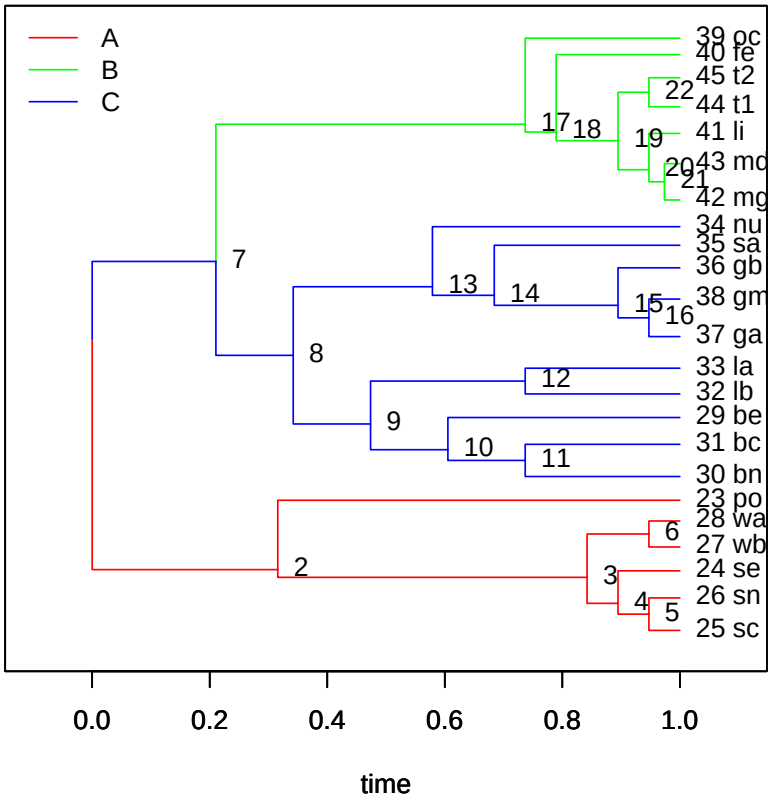
Compare it to the original "OU.LP" from above.

```
> plot(tree, regimes=ou.lp, node.names=T)
```



We can create alternative paintings of the regimes to test against the data. Suppose we wanted to add a "clade specific" hypothesis that diverged in a similar time period (this is a completely made-up hypothesis, just for example):

```
> ou.clades <- paint( tree, subtree=c("1"="A","7"="B", "8"="C"),
+ branch=c("8"="C", "7"="C", "1"="A"))
> plot(tree, regimes=ou.clades, node.names=T)
```



Run the model:

```
> h6 <- hansen(log(bimac['size']),tree, regimes=ou.clades,alpha=1,sigma=1)
```

Rebuild our table and compare models:

```
> h <- append(h, h6)          # append (add on) new model results to our list h
> names(h)[length(h)] <- "OU.clades"    # add the name of the new model
> names(h)

[1] "BM"        "OU.1"      "OU.3"      "OU.4"      "OU.LP"
[6] "OU.clades"

> h.ic <- sapply( h, function(x) unlist(summary(x)[c('aic', 'aic.c', 'sic', 'dof')]) )
> print( h.ic, digits = 3)

         BM  OU.1  OU.3  OU.4 OU.LP OU.clades
aic   -30.7 -25.4 -29.2 -35.2 -39.6     -30.7
```

```
aic.c -30.1 -24.1 -25.6 -30.0 -36.1     -27.1
sic   -28.4 -22.0 -23.5 -28.4 -34.0     -25.0
dof     2.0   3.0   5.0   6.0   5.0       5.0
```

Chapter 13

Bivariate `ouch`

The `ouch` package has been completely rewritten by Aaron King to implement a bivariate model, as well as the new S4 class system described previously.

Correlated evolution is a major feature of evolutionary theory, and of great interest among comparative biologists. However, there have been few attempts to develop a bivariate OU model for comparative analysis. NOTE: We are about to submit a paper on this, please cite us (and wait for us to publish first!).

13.0.6 The Bivariate model

The Hansen model describes the evolutionary processes operative on a single quantitative character Hansen (1997); Butler and King (2004). In the case of two characters, we will accordingly have two equations.

$$dX_1(t) = \alpha_1 \left(\theta_1 - X_1(t)\right) dt + \sigma_1 \, dB_1(t). \tag{13.1}$$
$$dX_2(t) = \alpha_2 \left(\theta_2 - X_2(t)\right) dt + \sigma_2 \, dB_2(t). \tag{13.2}$$

The above system of eqs. 13.3 can be written in matrix form with vectors in the place of dX, X, and $dB(t)$, and square matrices in place of α and σ. The θ are already vector valued in the univariate case with a single value per adaptive regime. Here we simply have separate θ vectors for each character.

13.0.7 No Correlations

When evolution is uncorrelated, the α and σ matrices are diagonal:

$$\alpha = \begin{pmatrix} \alpha_{11} & 0 \\ 0 & \alpha_{22} \end{pmatrix} \qquad \sigma = \begin{pmatrix} \sigma_{11} & 0 \\ 0 & \sigma_{22} \end{pmatrix}.$$

This form implies that neither character influences the evolution of the other (i.e., they are evolving independently of one another, and we have a simple duplication of the univariate case).

13.1 Correlated Evolution

We can readily see, however, that the matrices may have off-diagonal elements. These evolutionary correlations can enter in as off-diagonal terms in either the α or the σ terms. In particular, they will have the following form:

$$\alpha = \begin{pmatrix} \alpha_{11} & \alpha_{21} \\ \alpha_{12} & \alpha_{22} \end{pmatrix} \qquad \sigma = \begin{pmatrix} \sigma_{11} & 0 \\ \sigma_{12} & \sigma_{22} \end{pmatrix}.$$

Where $\alpha_{12} = \alpha_{21}$, and without loss of generality, σ is lower-triangular.

Written out as separate equations, the model has the following form:

$$dX_1(t) = \alpha_{11}\left(\theta_1 - X_1(t)\right)dt + \alpha_{12}\left(\theta_2 - X_2(t)\right)dt + \sigma_{11}\,dB_1(t).$$
$$dX_2(t) = \alpha_{22}\left(\theta_2 - X_2(t)\right)dt + \alpha_{12}\left(\theta_1 - X_1(t)\right)dt + \sigma_{22}\,dB_2(t) + \sigma_{12}\,dB_1(t).$$

13.2 Implementation in `ouch`

To illustrate, we reanalyzed the evolution of sexual size dimorphism in association with habitat specialization in *Anolis* lizards (Butler and King, 2004), reformulated as evolution in male and female body size.

Load the data (tree+quantitive data) and regimes:

```
> require(ouch)
> regimes <- read.csv("Rdata/regimes.csv", row.names = 1)
> ssd <- read.csv("Rdata/ssd.data.csv", row.names = 1)
> otree <- with(ssd, ouchtree(nodes, ancestors, times, labels))
> xdata <- log(ssd[c("fSVL", "mSVL")])
> names(xdata) <- paste("log", names(xdata), sep = ".")
> nreg <- length(regimes)
```

ouch now requires you to specify an initial guess for the alpha and sigma matrices. Univariate models are specified by providing a single value. **The bivariate model is specified by providing multiple values for these guesses. The three element vector will be transformed into a symmetric 2x2 matrix for alpha and a lower-triangular matrix for sigma.**

```
> alpha.guess <- c(1, 0, 1)
> sigma.guess <- c(1, 1, 1)
```

Fit the model for the first regime:

```
> tic <- Sys.time()
> hansen(data = xdata, tree = otree, regimes = regimes[5], alpha = alpha.guess,
+     sigma = sigma.guess, method = "Nelder-Mead", maxit = 3000,
+     reltol = 1e-12)

call:
hansen(data = xdata, tree = otree, regimes = regimes[5], alpha = alpha.guess,
    sigma = sigma.guess, method = "Nelder-Mead", maxit = 3000,
    reltol = 1e-12)
    nodes ancestors times     labels         TW.6       TW.6.1 log.fSVL
1       1      <NA>  0.00                    twig         twig       NA
2       2         3  0.62            trunk-crown  trunk-crown       NA
3       3         4  0.37            trunk-crown  trunk-crown       NA
4       4         5  0.14            crown-giant  crown-giant       NA
5       5         1  0.08                   twig         twig       NA
6       6         8  0.65          trunk-ground trunk-ground       NA
7       7         8  0.61          trunk-ground trunk-ground       NA
8       8        10  0.54          trunk-ground trunk-ground       NA
9       9        10  0.65            grass-bush   grass-bush       NA
10     10        12  0.50          trunk-ground trunk-ground       NA
11     11        12  0.79           trunk-crown  trunk-crown       NA
12     12        14  0.43          trunk-ground trunk-ground       NA
13     13        14  0.63                 trunk        trunk       NA
14     14        23  0.29          trunk-ground trunk-ground       NA
15     15        16  0.78          trunk-ground trunk-ground       NA
16     16        17  0.57          trunk-ground trunk-ground       NA
17     17        22  0.36          trunk-ground trunk-ground       NA
18     18        19  0.77           trunk-crown  trunk-crown       NA
19     19        20  0.72           trunk-crown  trunk-crown       NA
20     20        21  0.51           trunk-crown  trunk-crown       NA
21     21        22  0.46          trunk-ground trunk-ground       NA
22     22        23  0.27          trunk-ground trunk-ground       NA
```

23	23	28	0.14		trunk-ground	trunk-ground	NA
24	24	124	0.60		trunk-crown	trunk-crown	NA
124	124	25	0.47		trunk-crown	trunk-crown	NA
25	25	26	0.41		trunk-crown	trunk-crown	NA
26	26	27	0.33		twig	twig	NA
27	27	28	0.23		twig	twig	NA
28	28	34	0.11		twig	twig	NA
29	29	145	0.40		crown-giant	crown-giant	NA
145	145	34	0.28		crown-giant	crown-giant	NA
30	30	31	0.38		grass-bush	grass-bush	NA
31	31	34	0.22		twig	twig	NA
32	32	33	0.77		trunk-ground	trunk-ground	NA
33	33	161	0.72		trunk-ground	trunk-ground	NA
161	161	34	0.44		trunk-ground	trunk-ground	NA
34	34	1	0.08		twig	twig	NA
36	36	2	1.00	A_aliniger	trunk-crown	trunk-crown	3.815512
62	62	24	1.00	A_allisoni	trunk-crown	trunk-crown	4.100989
54	54	17	1.00	A_allogus	trunk-ground	trunk-ground	3.749504
64	64	134	1.00	A_alutaceu	grass-bush	grass-bush	3.487375
134	134	27	0.54		grass-bush	grass-bush	NA
60	60	110	1.00	A_angustic	twig	twig	3.653252
110	110	26	0.40		twig	twig	NA
49	49	13	1.00	A_breviros	trunk	trunk	3.693867
37	37	2	1.00	A_chlorocy	trunk-crown	trunk-crown	3.992681
38	38	3	1.00	A_coelesti	trunk-crown	trunk-crown	4.000034
44	44	7	1.00	A_cooki	trunk-ground	trunk-ground	3.728100
43	43	7	1.00	A_cristate	trunk-ground	trunk-ground	3.797734
66	66	29	1.00	A_cuvieri	crown-giant	crown-giant	4.782479
70	70	32	1.00	A_cybotes	trunk-ground	trunk-ground	3.927896
50	50	13	1.00	A_distichu	trunk	trunk	3.797734
39	39	413	1.00	A_equestri	crown-giant	crown-giant	5.045359
413	413	4	0.90		crown-giant	crown-giant	NA
48	48	11	1.00	A_evermann	trunk-crown	trunk-crown	3.958907
56	56	18	1.00	A_garmani	crown-giant	crown-giant	4.412798
57	57	19	1.00	A_grahami	trunk-crown	trunk-crown	3.784190
42	42	6	1.00	A_gundlach	trunk-ground	trunk-ground	3.811097
35	35	403	1.00	A_henderso	grass-bush	grass-bush	3.693867
403	403	4	0.58		grass-bush	grass-bush	NA
53	53	16	1.00	A_homolech	trunk-ground	trunk-ground	3.706228
69	69	150	1.00	A_insolitu	twig	twig	3.676301
150	150	31	0.65		twig	twig	NA
46	46	9	1.00	A_krugi	grass-bush	grass-bush	3.671225
59	59	101	1.00	A_lineatop	trunk-ground	trunk-ground	3.788725

```
101   101        21  0.62            trunk-ground trunk-ground       NA
61     61       113  1.00 A_loysiana        trunk        trunk 3.575151
113   113        25  0.59                   trunk        trunk       NA
40     40         5  1.00 A_occultus         twig         twig 3.668677
68     68        30  1.00  A_olssoni   grass-bush   grass-bush 3.703768
55     55        18  1.00 A_opalinus  trunk-crown  trunk-crown 3.701302
52     52        15  1.00 A_ophiolep   grass-bush   grass-bush 3.440418
41     41         6  1.00 A_poncensi   grass-bush   grass-bush 3.678829
63     63        24  1.00 A_porcatus  trunk-crown  trunk-crown 3.998201
45     45         9  1.00 A_pulchell   grass-bush   grass-bush 3.610918
65     65        29  1.00 A_ricordii  crown-giant  crown-giant 4.933754
51     51        15  1.00   A_sagrei trunk-ground trunk-ground 3.688879
67     67        30  1.00 A_semiline   grass-bush   grass-bush 3.616309
71     71        32  1.00  A_shrevei trunk-ground trunk-ground 3.832980
47     47        11  1.00 A_stratulu  trunk-crown  trunk-crown 3.686376
58     58        20  1.00 A_valencie         twig         twig 4.226834
72     72        33  1.00 A_whiteman trunk-ground trunk-ground 3.873282
    log.mSVL
1         NA
2         NA
3         NA
4         NA
5         NA
6         NA
7         NA
8         NA
9         NA
10        NA
11        NA
12        NA
13        NA
14        NA
15        NA
16        NA
17        NA
18        NA
19        NA
20        NA
21        NA
22        NA
23        NA
24        NA
124       NA
```

```
25         NA
26         NA
27         NA
28         NA
29         NA
145        NA
30         NA
31         NA
32         NA
33         NA
161        NA
34         NA
36   4.063885
62   4.382027
54   4.030695
64   3.569533
134        NA
60   3.788725
110        NA
49   3.864931
37   4.276666
38   4.247066
44   4.085976
43   4.195697
66   4.877485
70   4.188138
50   3.935740
39   5.129899
413        NA
48   4.258446
56   4.700480
57   4.182050
42   4.171306
35   3.869116
403        NA
53   3.958907
69   3.737670
150        NA
46   3.906005
59   4.177459
101        NA
61   3.706228
113        NA
```

```
40  3.663562
68  3.802208
55  3.901973
52  3.600048
41  3.819908
63  4.261270
45  3.848018
65  5.023222
51  3.977811
67  3.728100
71  4.001864
47  3.843744
58  4.374498
72  4.082609

alpha:
          [,1]      [,2]
[1,] 3.2885637 0.2651125
[2,] 0.2651125 5.0304925

sigma squared:
          [,1]      [,2]
[1,] 0.1164814 0.1606944
[2,] 0.1606944 0.2412312

theta:
$log.fSVL
 crown-giant   grass-bush        trunk  trunk-crown trunk-ground         twig
    4.966605     3.605634     3.678655     3.912812     3.772731     3.798625

$log.mSVL
 crown-giant   grass-bush        trunk  trunk-crown trunk-ground         twig
    5.003842     3.763422     3.832975     4.171052     4.076491     3.874264

    loglik   deviance        aic      aic.c        sic        dof
  82.05319 -164.10638 -128.10638 -116.10638  -86.15318   18.00000

> toc <- Sys.time()
> print(toc - tic)

Time difference of 15.72148 secs
```

Fitting the Hansen model is much more complex than the Brownian motion, because the α enters non-linearly into the likelihood function. With more variables, the complexity

increases, with a less well-behaved likelihood surface than the univariate case. There are frequently convergence issues.

Using the subplex method from package `subplex` helps. Other things to try include increasing the tolerance, increasing the number of maximum iterations allowed to reach convergence, and drawing initial guesses for alpha and sigma at random (and discarding the bad guesses). This of course increases computer time.

```
> tic <- Sys.time()
> h.subplex <- hansen(data = xdata, tree = otree, regimes = regimes[1],
+     alpha = alpha.guess, sigma = sigma.guess, method = "subplex",
+     maxit = 20000, reltol = 1e-12)
> toc <- Sys.time()
> print(toc - tic)
```

```
Time difference of 1.393377 mins
```

The Brownian motion model is fit:

```
> brown(data = xdata, tree = otree)
```

```
call:
brown(data = xdata, tree = otree)
    nodes ancestors times     labels log.fSVL log.mSVL
1       1      <NA>  0.00                 NA       NA
2       2         3  0.62                 NA       NA
3       3         4  0.37                 NA       NA
4       4         5  0.14                 NA       NA
5       5         1  0.08                 NA       NA
6       6         8  0.65                 NA       NA
7       7         8  0.61                 NA       NA
8       8        10  0.54                 NA       NA
9       9        10  0.65                 NA       NA
10     10        12  0.50                 NA       NA
11     11        12  0.79                 NA       NA
12     12        14  0.43                 NA       NA
13     13        14  0.63                 NA       NA
14     14        23  0.29                 NA       NA
15     15        16  0.78                 NA       NA
16     16        17  0.57                 NA       NA
17     17        22  0.36                 NA       NA
18     18        19  0.77                 NA       NA
```

19	19	20	0.72		NA	NA
20	20	21	0.51		NA	NA
21	21	22	0.46		NA	NA
22	22	23	0.27		NA	NA
23	23	28	0.14		NA	NA
24	24	124	0.60		NA	NA
124	124	25	0.47		NA	NA
25	25	26	0.41		NA	NA
26	26	27	0.33		NA	NA
27	27	28	0.23		NA	NA
28	28	34	0.11		NA	NA
29	29	145	0.40		NA	NA
145	145	34	0.28		NA	NA
30	30	31	0.38		NA	NA
31	31	34	0.22		NA	NA
32	32	33	0.77		NA	NA
33	33	161	0.72		NA	NA
161	161	34	0.44		NA	NA
34	34	1	0.08		NA	NA
36	36	2	1.00	A_aliniger	3.815512	4.063885
62	62	24	1.00	A_allisoni	4.100989	4.382027
54	54	17	1.00	A_allogus	3.749504	4.030695
64	64	134	1.00	A_alutaceu	3.487375	3.569533
134	134	27	0.54		NA	NA
60	60	110	1.00	A_angustic	3.653252	3.788725
110	110	26	0.40		NA	NA
49	49	13	1.00	A_breviros	3.693867	3.864931
37	37	2	1.00	A_chlorocy	3.992681	4.276666
38	38	3	1.00	A_coelesti	4.000034	4.247066
44	44	7	1.00	A_cooki	3.728100	4.085976
43	43	7	1.00	A_cristate	3.797734	4.195697
66	66	29	1.00	A_cuvieri	4.782479	4.877485
70	70	32	1.00	A_cybotes	3.927896	4.188138
50	50	13	1.00	A_distichu	3.797734	3.935740
39	39	413	1.00	A_equestri	5.045359	5.129899
413	413	4	0.90		NA	NA
48	48	11	1.00	A_evermann	3.958907	4.258446
56	56	18	1.00	A_garmani	4.412798	4.700480
57	57	19	1.00	A_grahami	3.784190	4.182050
42	42	6	1.00	A_gundlach	3.811097	4.171306
35	35	403	1.00	A_henderso	3.693867	3.869116
403	403	4	0.58		NA	NA
53	53	16	1.00	A_homolech	3.706228	3.958907

```
69     69       150  1.00 A_insolitu 3.676301 3.737670
150   150        31  0.65                NA       NA
46     46         9  1.00    A_krugi 3.671225 3.906005
59     59       101  1.00 A_lineatop 3.788725 4.177459
101   101        21  0.62                NA       NA
61     61       113  1.00 A_loysiana 3.575151 3.706228
113   113        25  0.59                NA       NA
40     40         5  1.00 A_occultus 3.668677 3.663562
68     68        30  1.00  A_olssoni 3.703768 3.802208
55     55        18  1.00 A_opalinus 3.701302 3.901973
52     52        15  1.00 A_ophiolep 3.440418 3.600048
41     41         6  1.00 A_poncensi 3.678829 3.819908
63     63        24  1.00 A_porcatus 3.998201 4.261270
45     45         9  1.00 A_pulchell 3.610918 3.848018
65     65        29  1.00 A_ricordii 4.933754 5.023222
51     51        15  1.00   A_sagrei 3.688879 3.977811
67     67        30  1.00 A_semiline 3.616309 3.728100
71     71        32  1.00  A_shrevei 3.832980 4.001864
47     47        11  1.00 A_stratulu 3.686376 3.843744
58     58        20  1.00 A_valencie 4.226834 4.374498
72     72        33  1.00 A_whiteman 3.873282 4.082609

sigma squared:
          [,1]      [,2]
[1,] 0.1523957 0.1564986
[2,] 0.1564986 0.1746020

theta:
NULL
   loglik  deviance       aic     aic.c       sic       dof
 22.56325 -45.12651 -35.12651 -34.26937 -23.47284   5.00000
```

13.3 Exercises

1. Run the Hansen model on the remaining regimes. Can you use an apply method to run them all at once?

2. Plot the multiple regime hypotheses.

3. Compare results.

13.4 Variations of the OU Model — Brian?

Instead of assuming a constant σ across the entire tree, O'Meara et al. (2006) developed a Brownian motion model that allows two or more σ values. This can be interpreted as having different rates of evolution in different regions of the tree.

Other possibilities exist. The difficulty for the future will be twofold: (1) Ensuring that the complexity of the model is reasonable given the information content of the data (i.e., are the parameter estimates and likelihoods well-behaved?). (2) Thinking hard about the best evolutionary and biological interpretations of the models.

Chapter 14

Stochastic Simulations

Let's make some graphical animations to illustrate the BM and OU model.

14.1 Brownian motion model

Recall that we described a Brownian motion process using the following equation:

$$dX(t) = \sigma\, dB(t). \tag{14.1}$$

This equation says that the value of X in some small interval in time by an amount σ times a draw from a normal distribution. We can mimic this behavior by a simulation in discrete time:

```
> nsteps = 100       # number of steps in our simulation
> devs =rnorm(nsteps)    # 100 draws from a normal distribution

> x <- c(0:100)
> for (i in 1:nsteps)
+ {
+    x[i+1] <- x[i] + devs[i]
+ }

> x

  [1]   0.0000000   0.1398150  -0.9129790  -1.0606427  -2.2525884  -4.6723994
  [7]  -5.2284680  -4.1715896  -4.6292394  -3.2756350  -3.4841759  -1.7542901
 [13]  -1.0170635  -2.5631703  -3.3620044  -2.7222800  -1.4698245  -3.2159180
```

```
 [19]  -3.3172463  -3.2997350  -2.7618601  -2.2288675  -1.2121975  -4.1983719
 [25]  -2.6316545  -0.2411552  -0.2740777  -0.8952370  -1.3863338  -1.7818611
 [31]  -1.1894663  -1.9055268  -2.7520503  -2.5360662  -3.6123742  -3.5832260
 [37]  -2.7873807  -1.2955992  -1.1005614  -1.6435168  -2.0445420  -4.1914657
 [43]  -4.3461264  -3.8795727  -3.7019795  -4.1371830  -4.1151274  -4.4752891
 [49]  -5.1085778  -3.2530366  -4.0491956  -4.7932082  -4.4860720  -4.3282460
 [55]  -3.3647450  -2.1359339  -0.6151603   0.7838015   1.1423835   2.8347385
 [61]   2.9387732   2.0933772   1.7653739   2.3192160   2.4614209   2.0293380
 [67]   3.1288531   2.0496851   2.4685896   2.0939789   2.3612080   2.8136793
 [73]   1.9271181   1.2493612   2.4380965   1.8877843   1.3367051  -0.3797799
 [79]  -0.8989621  -2.9412598  -2.4049521  -3.7088834  -5.4439704  -6.7582157
 [85]  -6.7754452  -7.8426826  -8.1360784  -6.8419380  -7.2002330  -9.5248951
 [91]  -9.6562674 -10.3370857 -10.2506347 -10.5026172 -10.6147336 -10.0468655
 [97] -10.3139792  -9.7870474  -9.9488377  -8.9038718  -9.3426409
```

We can plot this single random walk:

```
> plot(1:length(devs), devs, type = "n", col="red",
+ ylim = c(-max(devs)*30, max(devs)*30),
+ xlab="Time", ylab="Value", main="BM Simulation")
> x <- c(0:100)
> for (i in 1:nsteps)
+ {
+    x[i+1] <- x[i] + devs[i]
+    lines(i:(i+1), x[i:(i+1)], col="red")
+ }
```

In order to do 100 random walks, we need to place an outer loop, once for each random walk:

```
> bm.plot.slow <- function( sigma=1, nsteps=100, nlineages=100, yylim=c(-50, 50) )
+ {
+     plot(1:length(devs), devs, type = "n", col="red",
+     ylim = c(-max(devs)*30, max(devs)*30),
+     xlab="Time", ylab="Value", main="BM Simulation")
+
+     for (i in 1:nlineages)       # number of lineages to simulate
+     {
+         x <- c(0:100)
+         devs =rnorm(nsteps)    # 100 draws from a normal distribution
+         for (i in 1:nsteps)
+         {
+            x[i+1] <- x[i] + sigma*devs[i]   # BM equation
```

```
+                # step through time, increasing x a little bit each time
+             lines(i:(i+1), x[i:(i+1)], col="red")   # plot line segment
+         }
+     }
+ }
```

The loop is easier to understand in terms of a stochastic process, but actually we can write this code much more compactly:

Adding up a series of BM steps using the cumulative sum function:

```
>     sigma=1
>     cumsum(rnorm(nsteps, sd=sigma))
```

Plotting all the line segments at once:

```
>     y <- c(0, cumsum(rnorm(nsteps, sd=sigma)) )
>     lines(0:nsteps, y)
```

Or even more compactly:

```
>    sigma=1
>    lines(0:nsteps, c(0, cumsum(rnorm(nsteps, sd=sigma))))
```

And doing all of the lineages using lapply:

```
> bm.plot <- function( sigma=1, nsteps=100, nlineages=100, yylim=c(-50, 50))
+ {
+ # Set up plotting environment
+         plot(0, 0, type = "n", xlab = "Time", ylab = "Trait",
+                 xlim=c(0, nsteps), ylim=yylim)
+
+ # Draw random deviates and plot
+         lapply( 1:nlineages, function(x)
+           lines(0:nsteps, c(0, cumsum(rnorm(nsteps, sd=sigma)))))
+ }
```

If you want to show the simulations to screen, then you may actually prefer to do the slower for-loops, as the lapply is too fast.

14.2 Exercises

1. Go back to `bm.sim.slow` and modify it to an OU. Recall the OU equation:

$$dX(t) = \alpha\,(\theta - X(t))\,dt + \sigma\,dB(t). \tag{14.2}$$

   Hint: You will need to make two new parameters.

2. Introduce a branch to either the BM or simulation. You will need to simulate a single lineage for half the time, then two lineages for the rest of the time.

14.3 Making movies

In order to make a movie of the plot, you will need to save a series of plots as separate graphics files, similar to the "flip-books" you played with as a child. You need to make a plot with the first lineage, then the first two lineages, then the first three lineages, and so on.

So it would make sense to make a matrix to store the lineages, then plot through cumulatively:

```
> nsteps=100
> nlineages=30
> sigma=1
> sims <- sapply(1:nlineages, function(x) c(0, cumsum(rnorm(nsteps, sd=sigma))))
> yylim <- c(-30, 30)
> png(filename="movies/Rplot%03d.png")
>               # turn on png graphical device (write to file)
> for (i in 2:nlineages)
+ {
+         plot(0, 0, type = "n", xlab = "Time", ylab = "Trait",
+         xlim=c(0, nsteps), ylim=yylim)
+         apply( sims[,1:i], 2 , function(x) lines(0:nsteps, x, col="red"))
+ }
> dev.off()      # turn off png
```

Then in a terminal, move into the `movies` directory. If you have imagemagik installed:

convert -delay 10 Rplot.png Rplot.gif

To make a .mov file, you can use Quicktime Pro (but you have to pay for the Pro upgrade). In R version 2.8 there is a new package named `animation` which calls ImageMagick from R. It was sort of touch-and-go on my Mac under R 2.7.

14.4 RGL graphics

The 3D animations that I showed were produced using the package `rgl`. Unfortuantely, there is a bug that is currently being fixed right now so I cannot demonstrate it for you. It is a bug on the mac platform.

You can see the graph gallery at http://rgl.neoscientists.org/gallery.shtml. I have also included my source code in the webdav. Under `ou2drgl.R`.

Chapter 15

Writing Simple Packages by Jason Pienaar and Marguerite Butler

The easiest way to start making a package is to use the package skeleton function:

```
> f <- function(x,y) x+y
> g <- function(x,y) x-y
> d <- data.frame(a=1, b=2)
> e <- rnorm(1000)
```

```
> package.skeleton(list=c("f","g","d","e"), name="mypkg")
```

This will make a package directory in your working directory called `mypkg`. This is a good option if the package is very small. However, if you are building up a number of functions, you will want to save all of your functions to a folder, and then run the package skeleton directly on the files in that directory. For example, if our files for the package `phylohelper` are in a folder called "ourpackage", then in a terminal window, change directory to inside "ourpackage", then run the command:

```
> package.skeleton(name="phylohelper", code_files=list.files(), force=T)
```

Three elements are required:

DESCRIPTION a file

R your source code

man the help file directory

Every named function and dataset in R requires a help page or listing as an alias on a help page. The package skeleton creates templates, you simply have to fill them with your information.

There are also optional directories:

data included datasets

demo demonstrations

exec

inst (see below)

po

src C code or code in other languages

tests developer provided code tests

Optional files:

INDEX

NAMESPACE (discusssed in a later session)

configure script files executed before installation on Unix-alikes

cleanup script files, after installation on Unix-alikes if "–clean" was given as argument

LICENSE/LICENCE/COPYING copy of GNU public license, GPL-2, etc. Refer to the copies on http://www.r-project.org/Licenses or with base package in directory `share/licenses`.

NEWS see conventions in http://www.gnu.org/prep/standards/standards.html#Documentation.

README and ChangeLog are ignored by R but useful for users

15.1 Cross-platform compatibility

- Avoid using `file names` containing ASCII control characters as well as " * : / < > ? backslash and `|`.
- Avoid using filenames containing `con, prn, aux, clock$, nul, com1 -- com9, and lpt1 -- lpt9`.

- Avoid filenames in the same directory which only differ by upper/lower case.
- Names of '.Rd' (help) files must be ASCII and not contain %.
- No spaces in file names
- It is a good idea to avoid shell metacharacters `(){}'[]$`

15.2 Description File

The entire file should be written in ASCII, and continuation lines must start with a space or tab.

Mandatory elements: Package, Version, License, Description, Title, Author, and Maintainer. All else is optional.

Package The package name should start with a letter and be only contain only alphanumeric and '.' characters.

Version Version number should have the form '0.1-0'.

Description Can be multi line but only one paragraph.

Title short description of package (sometimes truncated to 65 char).

Author package writer

Maintainer A single name

Optional:

Date optional, but use yyyy-mm-dd format.

Depends comma-separated list of package names which the current package requires. Particular versions or comments are enclosed in parentheses (with version number).

Imports optional, lists packages whose namespaces are imported but don't need to be attached. See Writing R Extensions. Any name spaces accessed by "::" or ":::" need to be listed in depends, imports, or suggests, as R uses this info to decide which additional packages to install or reinstall (esp. important for S4 classes, as their class definitions may be evolving).

Suggests packages that are not necessarily needed, for example if they are only used in examples of vignettes, or packages loaded in the body of particular functions.

Enhances lists packages "enhanced" by your package, if you are writing additional methods for their classes.

Rules of thumb: `Imports` if you only need the namespace to load the package. `Depends` much stronger — need the package to be attached to successfully load the package. Will be loaded when your package is loaded. `Suggests` includes all packages needed to pass `R CMD check` (i.e., packages used anywhere at all in the package, no matter how obscure or infrequently). Use suggests especially in the case where you're only using the datasets from the package.

Optional fields:

Collate can be used to control the collation order for R code files when they are concatenated into a single file upon installation from source. If present, must list all R code files in package.

LazyLoad and LazyData control whether the R objects use lazy-loading. If using the `methods` package, should specify '`LazyLoad: yes`'.

15.3 Other directories

`demo` (optional) contains (`.R`) scripts for demonstrating some features. Run using the `demo()` command. If present, must contain a '`00Index`' file with one line for each demo giving its name and a description separated by a white space.

`inst` The contents of this optional directory will be copied recursively into the installation directory. Happens after `src` is build so its `Makefile` can create files to be installed. May want to add a `CITATION` file for the `citation` function. `tests` is a subdirectory for test code. Usually tests of specific functions within the package.

15.3.1 Documentation

All named R objects (data, functions) must be referenced either by having a page of its own ("
`name{}`") or being mentioned on another page ("
`alias{}`"). There is only one `name` per .Rd file, but there can be many `alias`es (this is a means of grouping together related functions).

The package skeleton will create a separate file for each object that you have included. You may want to delete some of them if they are redundant (as a rule of thumb, help directories with over 40 pages or so become a bit overwhelming for users to browse through).

If you want to add additional help files, use the command:

```
> prompt(object.name, file="test.Rd")
```

The object name and the file need not share the same name.

15.3.2 Vignettes

Package vignettes are included in a subdirectory (`inst/doc`). When they are placed here, `R CMD check` checks all code chunks (but not those with `eval=F`. Once the packages is installed, the vignette is inserted into `doc` directory. Make sure all files needed by the vignetter are accessible by placing them in the `inst/doc` hierarchy of the source package or using calls to `system.file()`. See the Sweave chapter for instructions and examples for writing Sweave documents.

15.4 Checking the entire package

Change to a terminal window, and move to the directory directly above the start of the package directory tree. We can use the R CMD check command on the package as follows:

```
R CMD check PACKAGENAME
```

This will print a number of diagnostic tests as well as build a version of the package (in the same directory that the package is in). Run the example and have a look at all the diagnostics, it should OK everything, if there was a problem we would get an error message pointing us to the likely source of the error.

It is also possible to check single documentation files using

```
R CMD Rd2txt help.page.name.Rd
```

15.5 Building the package

The next step is to build the package using the R CMD build command. This will create a tar file which can then be distributed as a (hopefully) functional package:

```
R CMD build PACKAGENAME
```

To install this package on your machine use the R CMD install command:

```
R CMD INSTALL PACKAGENAME
```

15.6 Distributing the package

15.6.1 CRAN

CRAN is the main repository for R packages. It is a mirrored-network of web sites that store the R distributions, User manuals as well as contributed packages.

One way to distribute you package is to submit it to the CRAN network so that anybody can download and use it. The R package must have passed R CMD check. The R CMD build command makes the .tar.gz release file. When all the testing has been done, the tar.gz file can be uploaded to ftp://cran.r-project.org/incoming/, using "anonymous" as a user name and your email address as a password. Also send an email a message to CRAN@R-project.org. The CRAN maintainers will run further tests on your package before putting the submission in the main package archive.

15.6.2 R-forge

For developing packages, R-forge http://r-forge.r-project.org/ is a good choice for hosting. R-forge has svn capabilities, so multiple developers can simultaneously work on a package and curious users can download and test it.

15.6.3 Creating Binaries

It is possible to build binaries on your computer by using the command:

```
R CMD build -binary PACKAGENAME
```

Note however, that this will be specific to your architecture and OS.

Chapter 16

System Commands by Brian O'Meara

R can interact with non-R code in at least two ways. One is with commands for direct passing of objects to and from other code: see `.C`, `.Call`, and `.External`. These are for calling functions, rather than programs. To speed up execution, many packages move some operations from R code to C code (`ape` does this, for example, as does `phylobase` for NEXUS file reading) and they use one of these functions (perhaps hidden in a helper package) to call that C code. These functions require access to the C or other external code and an understanding of objects in R and the target language. Thus, they won't work if you want to, say, call paup, which is closed source, or if you just want to run MrBayes without modifying the code to work with R. `system` is a command that will run commands in the shell. This function is common in programming languages: you can find it in Perl, PHP, C, C++, Java, and probably others (it's often called `system` or `exec`).

The basic function is

```
system(command, intern = FALSE, ignore.stderr = FALSE, wait = TRUE, input = NULL)
```

(there are other functional options on other systems – we're just focusing on Mac OS X).

`command` is just a text string containing the command. If you can do it in Terminal, you should be able to do it from R.

```
> system(command = "ls -l")
```

```
> system(command = "cal")
```

Just this basic function alone is terrifically handy. For example, Christoph Heibl (http://www.christophheibl.de/) has functions that can create NEXUS files (just using `write`),

then use `system` to start MrBayes, Garli, or other programs, then use `ape`'s tree reading functions to load the trees back into R. See his page at http://www.christophheibl.de/r.html for these and other useful scripts. The basic idea is to create an input file, run the relevant program using `system`, and then load the output file back into R. The only gotcha here is to make sure the command will run properly: for example, MrBayes has a command line executable called `mb` (different from the double-clickable icon). The computer only knows that typing "mb" means you want to run this program if `mb` resides in an area where the computer looks for executables (such as `/usr/bin`) or you pass the computer the full path to the executable (such as `/Users/bcomeara/Desktop/mb` if the executable is on the desktop).

The `wait` option tells R to wait for the command to finish (if true, the default) or immediately go to the next line in your R batch file (or back to the command prompt) after starting the command. Normally, you do want to wait for the command to finish: for example, if you want to run paup to find a tree, then load the tree back into R, it makes sense to wait for paup to finish its search and save the tree to a file before trying to load that file. In some cases, you might want to start the command running and not wait for it to finish.

```
> system(command = "ls -lh /usr/bin > ~/Desktop/ls.txt", wait = F)
```

This command will list all the items in the /usr/bin directory with their file sizes and modification times and store this info in a file on the desktop. You could imagine using this sort of command to store a list all the output files in a working directory for future reference – it might take a little while to run, but you don't depend on the results in R, so waiting for the command to finish before moving on is just a waste of time.

This is actually a way to run your computer as a mini high performance computing cluster. Most computers now have multicore processors ("Intel Duo" is dual core). Mac laptops often have two cores. Some desktops may have up to 8. These are treated almost like separate CPUs (though with shared memory), so an R session just runs on one of these cores (a relative handful of programs have been written to run on multiple cores simultaneously; as far as I can tell, R isn't one of them). If you have N cores, and want to do, say, 100 bootstrap replicates, you could divide these into sets of $100/N$ replicates. Then, N-1 of these sets could be set to run using `system(command="R CMD BATCH batchfileReplicate1.R", wait=F)` and the last set could be run in the main program. Thus, you would be running N operations at once rather than just one at a time, taking $1/N$ as long.

So far, the results have not come back directly to R, but are just stored in whatever output files the command creates. `intern=T` configures `system` to return output from the function directly (and implicitly sets `wait=T`).

```
> myfiles <- system(command = "ls", intern = T)
```

The output is now stored in `myfiles`:

```
> myfiles
```

which we can see is a vector:

```
> class(myfiles)
```

```
[1] "character"
```

```
> length(myfiles)
```

```
[1] 139
```

Incidentally, if `intern=F`, the `system` function actually does return information: 256 times the return code of the command.

The `input` argument passes its contents a vector of character strings, element by element to lines in a temporary file. This file is passed to the command as an input.

As a somewhat silly example, pass this vector

```
> chorus <- c("It's a long way from Amphioxus.", "It's a long way to us.",
+     "It's a long way from Amphioxus to the meanest human cuss.",
+     "Well, it's goodbye to fins and gill slits, and it's welcome lungs and hair!",
+     "It's a long, long way from Amphioxus, but we all came from there.")
```

to the command line application grep (yes, it's also an R function).

```
> amphioxuslines <- system(command = "grep -i amphioxus", intern = T,
+     input = chorus)
> amphioxuslines
```

```
[1] "It's a long way from Amphioxus."
[2] "It's a long way from Amphioxus to the meanest human cuss."
[3] "It's a long, long way from Amphioxus, but we all came from there."
```

which returns just the lines with the word Amphioxus.

16.1 Exercises

1. Store a vector listing all the items on your desktop
2. Create a batch file in NEXUS format to get a list of command options in MrBayes. Use R to run MrBayes with this batch file
3. Download some sequences from Genbank within R, export them to fasta format, run clustalw to align them, and then import them back into R

Chapter 17

Divergence Times and Rates of Evolution

Chapter 18

Other Packages Available For Comparative Analysis

18.1 ade4

Here is a description from Thibaut Jombart, one of the package developers. `ade4` is a package for ecological data analysis within a phylogenetic context.

`ade4` is soon to undergo a major revision to handle phylo4d objects. This will likely take the form of a new project `adephylo`, that I will start after September 2008. What is for (quite) sure is that everything that currently exist for comparative methods in `ade4` will be available and improved in `adephylo`, along with some news.

Meanwhile, here is a small summary of what is currently available. Please do not hesitate if you have further questions.

newick2phylog this is the input function which reads character strings and outputs a `phylog` object (described in ?phylog). This is the main way to create a phylog object, which is the class used in ade4. Other related input functions are `hclust2phylog` and `taxo2phylog`. Objects of class `phylog` have optional components that can take a large amount of space. To disable this, use `add.tools = TRUE` in the input function. Ape imports `phylog` using `as.phylo`.

plot.phylog tree plot from `phylog` object

table.phylog this is the main graphical function for tree+data, quite similar to that for `phylo4d` objects in `phylobase`. Note that `phylog` does not possess data, so the represented dataset has to be specified to the function. `orthogram` implements the orthogram described by Ollier, S. et al. (2005) Biometrics, *62*, 471-477. This method decomposes the variance of a trait into several components, representing

different 'levels' (i.e. sets of nodes sharing the same common ancestor) of the phylogeny. There are 4 associated tests that can detect different kinds of phylogenetic structuring. `variance .phylog` performs a phylogenetic version of the classical ANOVA.

There is no function to perform the test of Abouheif, yet the neighbouring matrix underlying Abouheif's test is the $Amat component of a phylog and can be used to compute a Moran's I, which the test of Abouheif truely is.

I think these are the main features. There is a ML which can be used by your students if they have questions about ade4:
`adelist`cisrweb.univ-lyon1.fr@

18.2 geiger

18.3 picante

http://picante.r-forge.r-project.org/

`caic` is coming soon!

Bibliography

Butler, M. A. and A. A. King. 2004. Phylogenetic comparative analysis: a modeling approach for adaptive evolution. American Naturalist, **164**:683–695. URL http://www.journals.uchicago.edu/AN/journal/issues/v164n6/40201/40201.html.

Felsenstein, J. 1985. Phylogenies and the comparative method. American Naturalist, **125**:1–15. URL http://links.jstor.org/sici?sici=0003-0147%28198501%29125%3A1%3C1%3APATCM%3E2.0.CO%3B2-Y.

Felsenstein, J. 2004. Inferring Phylogenies. Sinauer, Sunderland, Mass.

Hansen, T. F. 1997. Stabilizing selection and the comparative analysis of adaptation. Evolution, **51**:1341–1351.

Harvey, P. H. and M. D. Pagel. 1991. The Comparative Method in Evolutionary Biology, volume 1 of *Oxford Series in Ecology and Evolution*. Oxford University Press, Oxford.

Martins, E. P., editor. 1996. Phylogenies and the Comparative Method in Animal Behavior. Oxford University Press.

O'Meara, B. C., C. Ane, M. J. Sanderson, and P. C. Wainwright. 2006. Testing for different rates of continuous trait evolution using likelihood. Evolution, **60**:922–933.

Paradis, E. 2006. Analysis of Phylogenetics and Evolution with R, volume XII of *Use R*. Springer-Verlag.

Paradis, E., J. Claude, and K. Strimmer. 2004. Ape: analyses of phylogenetics and evolution in the R language. Bioinformatics, **20**:289–290.