Getting Started in R for Phylogenetics

Marguerite A. Butler[1,2], Brian C. O'Meara[3], Jason Pienaar[1,4], Michael Alfaro[5], Graham Slater[6], and Todd Oakley[7]

[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`
[5]UCLA address, `michaelalfaro@ucla.edu`
[6]`gslater@ucla.edu`
[7]`todd.oakley@lifesci.ucsb.edu`

March 12, 2018

Contents

Chapter 1

Preliminaries

1.1 Computer Requirements and Installing R

This chapter is about 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.*

Computers I will be using a macintosh running El Capitan (OS 10.11.6), however, R is open source and available on PC and Linux as well. For the most part, the R commands are cross-platform compatible. The only exceptions are those that deal directly with other programs on your computer (the main one being to bring up a new graphics window – quartz() on a mac, and x11() on a PC or Linux).

R version 3.4.0 (Amusingly nicknamed "You Stupid Darkness". 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-3.4.0.pkg. Follow the directions from there.

1.1.1 Installing from source

If you would like to be able to install packages from source, you will need these components: C compiler (gcc), a fortran compiler (e.g., gfortran), and X11. If you don't know what this is about, it's OK – just skip it. If you do want to do it, take a look at the instructions on: `http://cran.r-project.org/bin/macosx/tools`

Xcode Tools This contains the C/C++ compiler. Install from the system disks that came with your computer. 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 Install from the link above (tools directory on the CRAN install page).

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

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 > R for Mac OS X FAQ > Building R from sources.

Note: for people who've recently upgraded their systems, please make sure you have Xcode Tools and X11 installed from the discs that came with your computer (they have to be the correct version for your new OS. For example, you can't use your Xcode Tools from Tiger on your mac running Snow Leopard).

1.2 R packages

Many of the packages that you will ever use 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):

ggplot2

pspline

ks

1.3 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.

1.4 Help! and Useful References

1.4.1 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.

1.5 For Folks who get serious about R programming

Programming with Data: A guide to the S Language by John M. Chambers. 2004. Springer. This is written by one of the authors of the S language, which R is based on. It has a lot of details that you will never find in the glossy books.

Chapter 2

Playing with R for the first time

2.1 Instructions

In this exercise, I want to introduce you to some of the built-in help facilities and documentation in R, and get you started with manipulating variables in R.

- If you haven't already done so, make a directory for this class. I would recommend naming it "Rclass" and putting it at some accessible location in your user directory (like at the top level of your User directory on a Mac, or at `C:/` on a windows machine). On my computer it would be like so: Fig. 2.1.

- Also within this directory, make another directory called "data". You will store all of your data files in there.

- Start up R.

- Move to your Rclass directory by using the `setwd("`*`path to Rclass`*`")` command.

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

  On a PC it will be something like:

  ```
  > setwd("C:/Rclass")
  ```

- Open the help facility using the command

  ```
  > help.start()
  ```

- Click on "An Introduction to R". The is *"the Bible"* for learning R.

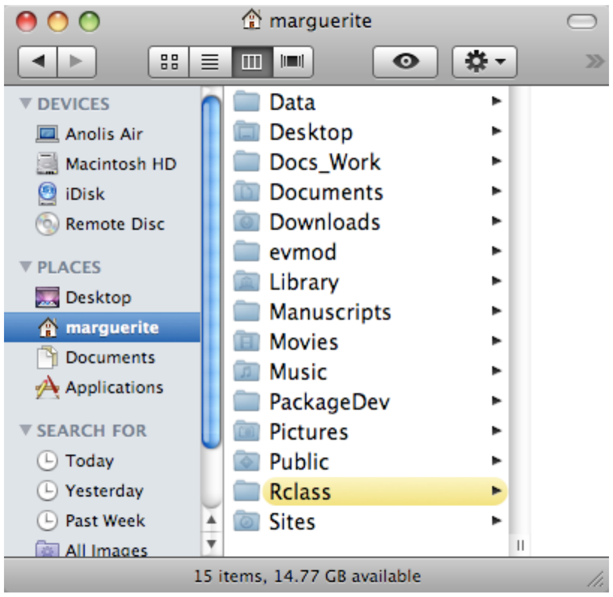


Figure 2.1: Rclass directory for saving course work. Make a folder in a convenient location on your computer, like at the top level of your user directory. When you are done with the course, you can move the whole folder to a permanent location with your other R code.

2.2 R session

Later, when you have more time, you will want to read and try out all of the section "Simple manipulations; numbers and vectors" (2.1 – 2.8). Please type the commands in yourself rather than cut-and-pasting. The typing helps develop ”finger memory” which you will need to become proficient at programming.

2.2.1 Vectors

For now, let's try playing around with R. Create a variable or "object" named `height` and save a value of 10. The arrow means to put "10" into `height`:

```
> height <- 10
```

To see the value of `height`, type it and press return:

```
> height
```

```
[1] 10
```

Now let's create a vector, a variable with several elements or "observations":

```
> height  <- c(10, 12, 51, 24, 32)
> height
```

```
[1] 10 12 51 24 32
```

The `c()` function *combines values into a vector or a list.* You will use it a lot. Create a vector of weights:

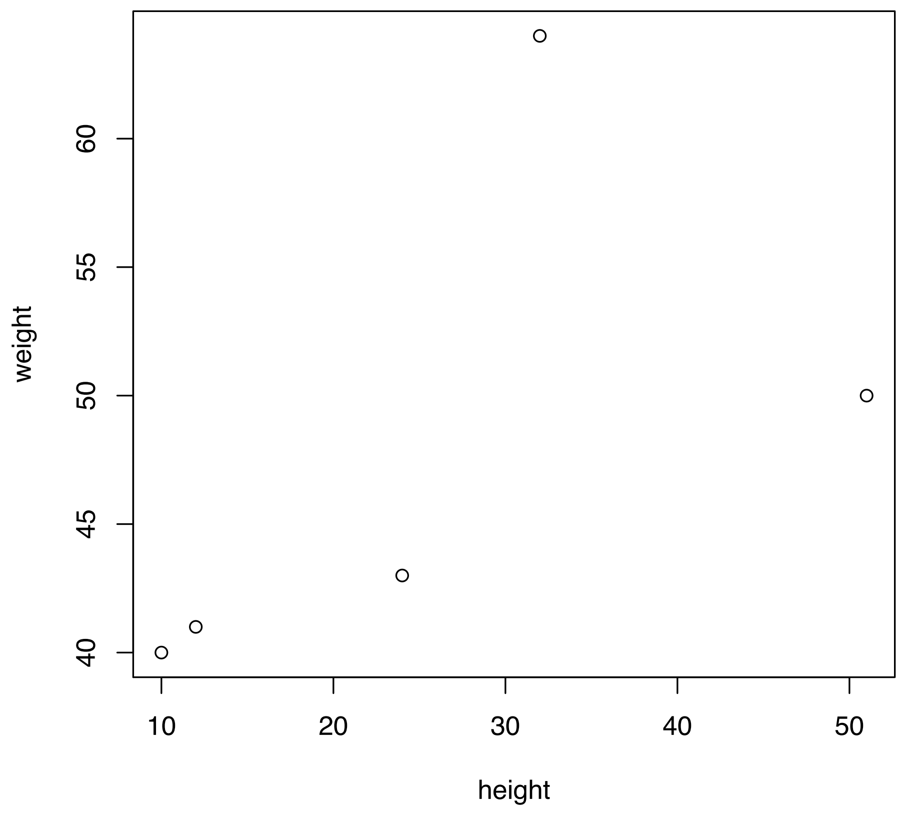
```
> weight <- c(40, 41, 50, 43, 64)
> weight
```

```
[1] 40 41 50 43 64
```

Whenever I am writing new code, I *ALWAYS* check to make sure the code produced the results I wanted. You should do the same. *Verify that EACH step worked correctly.* This means without errors!

Let's create our first plot. We'll use the `plot()` function, which is a generic function for just about any type of R object:

```
> plot(height, weight)
```



Voila!! Our first plot. Beautiful. Now suppose that we have males and females in the data, so we'd like some categorical variables for sex. In R, it's easy to do. Just create a character vector:

```
> sex <- "male"
> sex
```

```
[1] "male"
```

```
> sex <- c("male", "male")
> sex
```

```
[1] "male" "male"
```

2.3 Functions

We can create an object `sex` that contains one or more character strings in it, and use the generic `c()` function to create a vector of character strings. But it can get tedious typing the same thing over. So we can use the `rep()` function to repeat values:

```
> sex <- rep("male", 3)
> sex

[1] "male" "male" "male"

> sex <- c(sex, "female", "female")
> sex

[1] "male"   "male"   "male"   "female" "female"

> sex <- c( rep("male", 3), rep("female", 2) )
> sex

[1] "male"   "male"   "male"   "female" "female"
```

So what just happened? Why do the last two lines of code give the same result? *In R, as in most programming languages, the code is nested. The innermost function or bit of code is evaluated first, and whatever is returned is then the argument for the next outer bit of code.* So in the first line, we take three copies of "male" and shove it into `sex`. In the second line, we take the object `sex`, which is now a vector of three "male", and combine it with two copies of "female", into a vector of 5 elements. In the last line, first we create a vector of 3 males using the `rep` function, and then a vector of 2 females, and combine these two vectors together into a vector of 5 elements.

In fact, the last line above is equivalent to the following (of course you would never actually write a line like the one below, the nested `c()` are completely unnecessary, this is just for demonstration):

```
> sex <- c(c("male", "male", "male"), c("female", "female"))
> sex

[1] "male"   "male"   "male"   "female" "female"
```

You can see that using functions like `rep()` and `seq()` for sequence can save you a lot of time, when you have lots of repeated data.

```
> sex <- c(rep("male", 50), rep("female", 50))
> sex

  [1] "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"   "
 [12] "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"   "
 [23] "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"   "
 [34] "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"   "male"   "
 [45] "male"   "male"   "male"   "male"   "male"   "male"   "female" "female" "female" "
 [56] "female" "female" "female" "female" "female" "female" "female" "female" "female" "
 [67] "female" "female" "female" "female" "female" "female" "female" "female" "female" "
 [78] "female" "female" "female" "female" "female" "female" "female" "female" "female" "
 [89] "female" "female" "female" "female" "female" "female" "female" "female" "female" "
[100] "female"
```

2.3.1 Generating Random Deviates

Now let's go back to our original height and weight variables and make up some larger samples. This time, let's use the random number generator function `rnorm()`, which generates random normal deviates. We can specify the mean and standard deviation as below. Let's make the males with larger mean, but same standard deviation. To save paper, I'm not going to display the object contents to the screen, but you should keep doing it.

```
> height_m <- rnorm(50, mean=55, sd=5)
> height_f <- rnorm(50, mean=45, sd=5)
```

How do we combine these into one vector?

```
> height <- c(height_m, height_f)
```

We could have also created the height vector in one step. While we're at it, let's also make up some data for weight. Let's pretend that this data is for children in inches and pounds:

```
> height <- c( rnorm(50, mean=55, sd=5), rnorm(50, mean=45, sd=5) )
> weight <- c( rnorm(50, mean=80, sd=10), rnorm(50, mean=65, sd=8) )
```

In general, it's best to keep your coding simple, especially when you are learning. Write clean code that is easy for you to understand. If it takes an extra line, it's not a big deal. The computer is VERY fast, you will not slow down your program this way. On the other hand, you can easily confuse yourself and make BIG MISTAKES by trying to be too clever.

To plot by sex, we need to do tell R that the object `sex` contains categories or "factors". We do this using the `factor()` function:

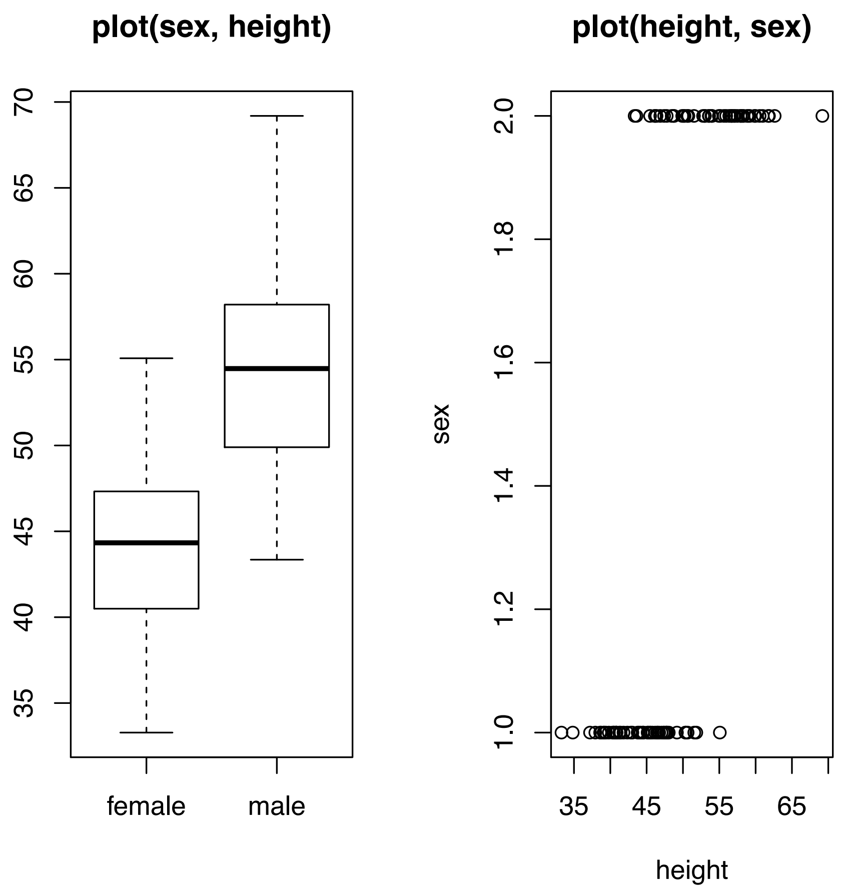
```
> sex <- factor(sex)
> sex
```

```
  [1] male   male   male   male   male   male   male   male   male   male   male   male
 [15] male   male   male   male   male   male   male   male   male   male   male   male
 [29] male   male   male   male   male   male   male   male   male   male   male   male
 [43] male   male   male   male   male   male   male   male   female female female femal
 [57] female female female female female female female female female female female femal
 [71] female female female female female female female female female female female femal
 [85] female female female female female female female female female female female femal
 [99] female female
Levels: female male
```

`sex` is now a factor, or categorical variable with two levels. Now we can plot with sex. Note that in R, when you make a bivariate plot where the first variable is a factor, it will create a barplot by default. If you put the quantitative variable first, you will get a scatterplot:

```
> plot(sex, height, main="plot(sex, height)")
> plot(height, sex, main="plot(height, sex)")
```

We added titles to the plots with the `main="mytitle"` argument, which is optional.



Now, let's run some statistics on our data. Is a child's weight related to height? We might want to run a linear regression, which we so using the `lm()` or linear model function. It produces a linear model object, let's save the output as lm.mf:

```
> lm.mf <- lm( weight ~ height )
```

There are several ways to give the linear model argument to `lm`, I prefer to use the formula representation `weight ˜height`, which is read *weight as a function of height.* You can produce the a summary of the regression using `summary()`. Often, however, you want to see an anova table:

```
> summary(lm.mf)

Call:
lm(formula = weight ~ height)

Residuals:
    Min      1Q  Median      3Q     Max
-22.252  -6.624  -1.007   5.587  40.851

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  49.9385     7.6824   6.500 3.37e-09 ***
height        0.4568     0.1550   2.948    0.004 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 11.12 on 98 degrees of freedom
Multiple R-squared: 0.08146,        Adjusted R-squared: 0.07208
F-statistic: 8.691 on 1 and 98 DF,  p-value: 0.003998

> anova(lm.mf)

Analysis of Variance Table

Response: weight
          Df  Sum Sq Mean Sq F value   Pr(>F)
height     1  1074.6 1074.63  8.6907 0.003998 **
Residuals 98 12118.0  123.65
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Oh wow, there is a very significant effect. But wait! We have boys and girls in the dataset. We need to add in gender as a covariate:

```
> lm.mf <- lm(weight ~ sex + height)
> anova(lm.mf)

Analysis of Variance Table

Response: weight
          Df Sum Sq Mean Sq F value    Pr(>F)
sex        1 4045.2  4045.2  44.264 1.707e-09 ***
height     1  282.8   282.8   3.094   0.08173 .
Residuals 97 8864.7    91.4
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

See what happened? Actually, males and females are significantly different, and there is no relationship between height and weight after accounting for sex. Does this make sense, given how we generated the data?

Note: Using the "+" between the parameters `sex` and `height` means to put them in as additive factors. If you want to include these as well as interactions, use "*". For interactions only (hardly ever done), use ":". Give it a try. For more explanation, see the formula help page:

```
> ?formula
```

2.3.2 Building a dataframe

The most typical data structure you will use is a dataframe. It is a "record format" type of layout, with the idea being one row per observation. You may have additional information or metadata that you want stored with your individual observations. For example, you may want a unique ID for each individual, and what city they are from, etc.

Let's create a unique ID. For some reason, we want each boy numbered from 1 to 50. Let's use the `seq()` function to create a sequence from 1 to 50, and the `paste()` function to combine them with "boy":

```
> seq(1,50)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
[34] 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

Is the same as:

```
>  1:50
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
[34] 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

Paste together with the word "boy":

```
>  boys <- paste("boy", 1:50, sep="")
>  girls <- paste("girl", 1:50, sep="")
>  ID <- c(boys, girls)
```

We separated the two parts of the paste with nothing, "". We could have separated with a "." or whatever we want.

Now let's create a city object. Suppose we collected 25 observations of each sex in Honolulu and Santa Barbara:

```
> city <- c( rep("Hon", 25), rep("SB", 25), rep("Hon", 25), rep("SB",25))
```

We could also repeat the repeat, since the 25 per city is a repeating pattern:

```
> city <- rep( c(rep("Hon", 25), rep("SB", 25)), times = 2)
```

This time we had to use the `times=` option, which means how many times to repeat the whole sequence. The other popular option is `each=`, which repeats element by element. To see the difference more clearly, try this simple example:

```
> rep(1:5, times=3)

 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5

> rep(1:5, each=3)

 [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

Now let's save all of our dataset together into one neat dataframe:

```
> dat <- data.frame(ID, sex, height, weight, city)

> dat
```

For really large datasets (saving paper here, you can print to screen), we can just look at the beginning and end of our dataframe:

```
> head(dat)

    ID  sex   height   weight city
1 boy1 male 50.74248 88.83133  Hon
2 boy2 male 46.14909 66.38892  Hon
3 boy3 male 53.56522 63.32075  Hon
4 boy4 male 46.36637 87.32131  Hon
5 boy5 male 50.58725 85.69922  Hon
6 boy6 male 62.62308 67.86794  Hon

> tail(dat)

        ID    sex   height   weight city
95  girl45 female 34.83214 68.87919   SB
96  girl46 female 43.03704 78.05635   SB
97  girl47 female 47.72882 71.00597   SB
98  girl48 female 40.86659 78.12755   SB
99  girl49 female 40.81870 73.01877   SB
100 girl50 female 41.40929 63.28541   SB
```

2.4 Save Your History

At the end of your session, save your session history. On a mac, press the little blue and yellow history icon and you will see the history sidebar appear. Click on “Save History.” Give it a file name as below. On a PC, use the savehistory function, read the following help page :

```
> ?savehistory
```

You will want to save it with an informative file name like `VecPractice.history`. For example:

```
> savehistory(file="VecPractice.history")
```

2.5 Insert Comments

Outside of R, open up your history (in a text editor) and clean up your code. Add comments to help you remember what this code means. Place them in your history, just before the relevant section. Comments in R are indicated by the `#` symbol. Anything to the right of one or more `#` is considered a comment, and not executed by R.

2.6 Exercises

Work through the section ”Simple manipulations; numbers and vectors” (2.1 – 2.8) in the *Introduction to R* – see top of this chapter for how to find it – and answer the questions below.

1. What is a numeric vector?
   Answer: `## A numeric vector is an ordered collection of numbers.`
2. Is ordinary arithmetic (+, -, *, /) on vectors in R done element-by-element or using matrix math? (to test an example, try or think about `x*y` where:
   $$x = \begin{pmatrix}1\\2\end{pmatrix} \qquad y = \begin{pmatrix}5\\1\end{pmatrix}$$
3. What is a sequence?
4. What is an logical value? What is a logical vector?
5. What is a missing value?

6. What is a character vector?

7. What is an index vector?

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

- Directories and file organization

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")      # 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="tree.primates.rda")  # save tree in Rdata format
```

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 `Rclass`. You will have to do this outside of R. Either create the `Rclass` folder through the Finder or open a terminal, change to your user directory if you're not there via "`cd` " then "`mkdir Rclass`." Please make two additional folders inside "`Rclass`" called "`Data`."

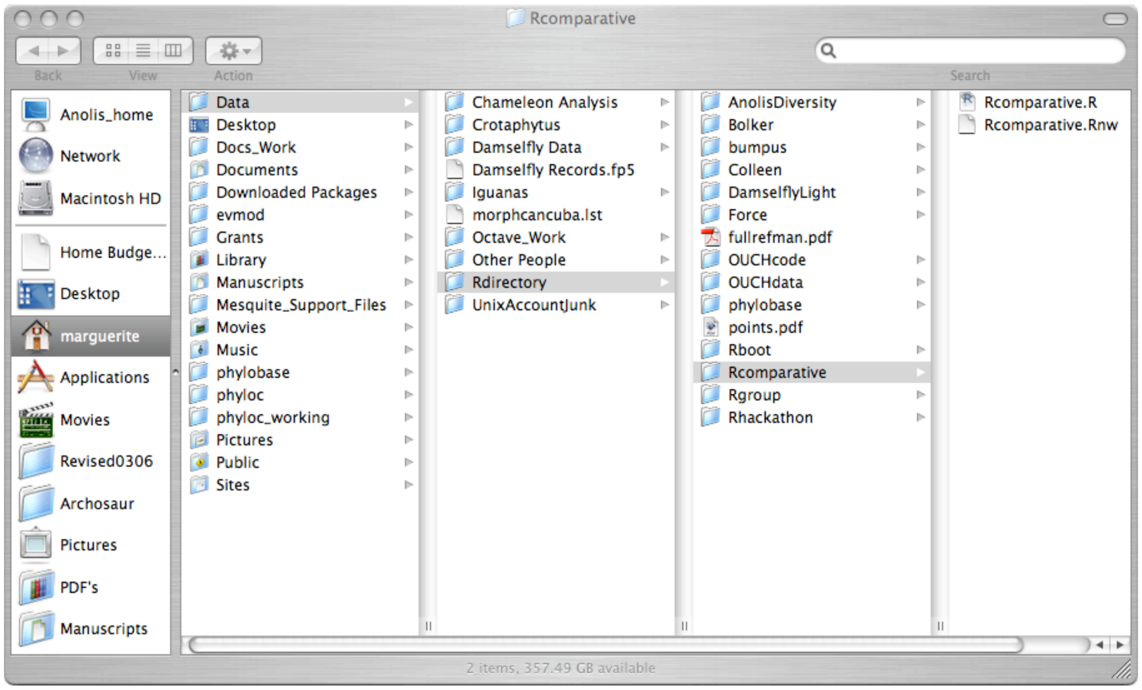


Figure 3.1: An example of directory organization to keep your R programming projects organized. Here the Rcomparative folder is moved to its final location. When you are actively working on a project, it may be more convenient to have it at the top level of your user directory (here ”marguerite”).

`Rclass` 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.

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

`Rdata` if you really start to have a large number of files, or if you want to keep your “raw” data pristine, set up a separate folder 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.

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 ”Rclass”.

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/Rclass")
```

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

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

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 Rclass folder.

```
> setwd("~/Rclass")
> 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="tree.primates.rda")  # save tree in Rdata format
```

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 data-point.

```
> 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)
```

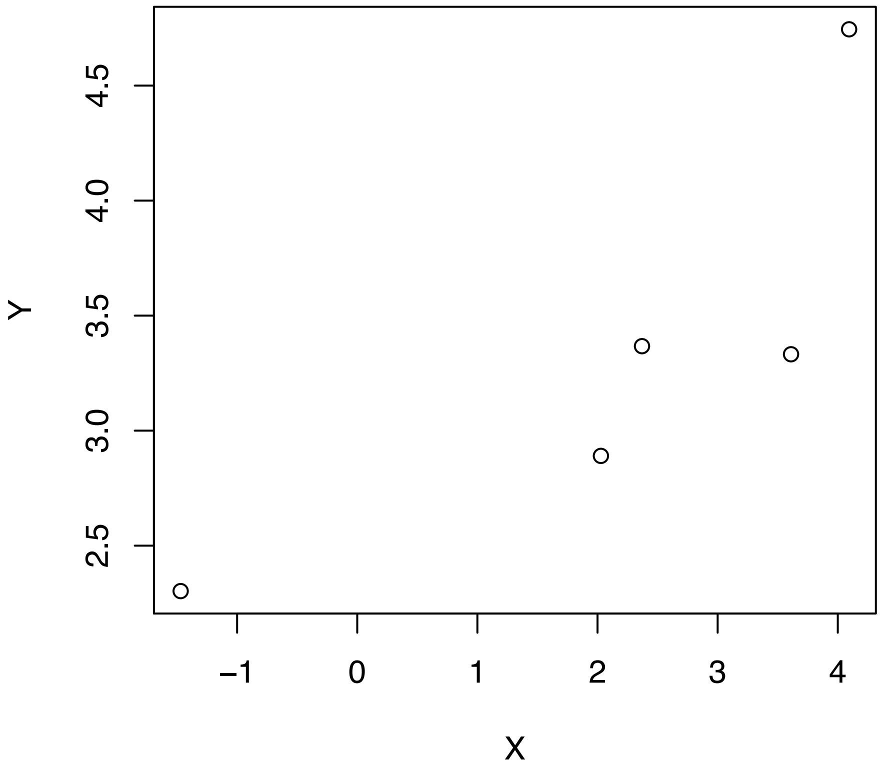


Figure 3.2: A plot of our raw data.

```
> 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 covariance 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 **?**. 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

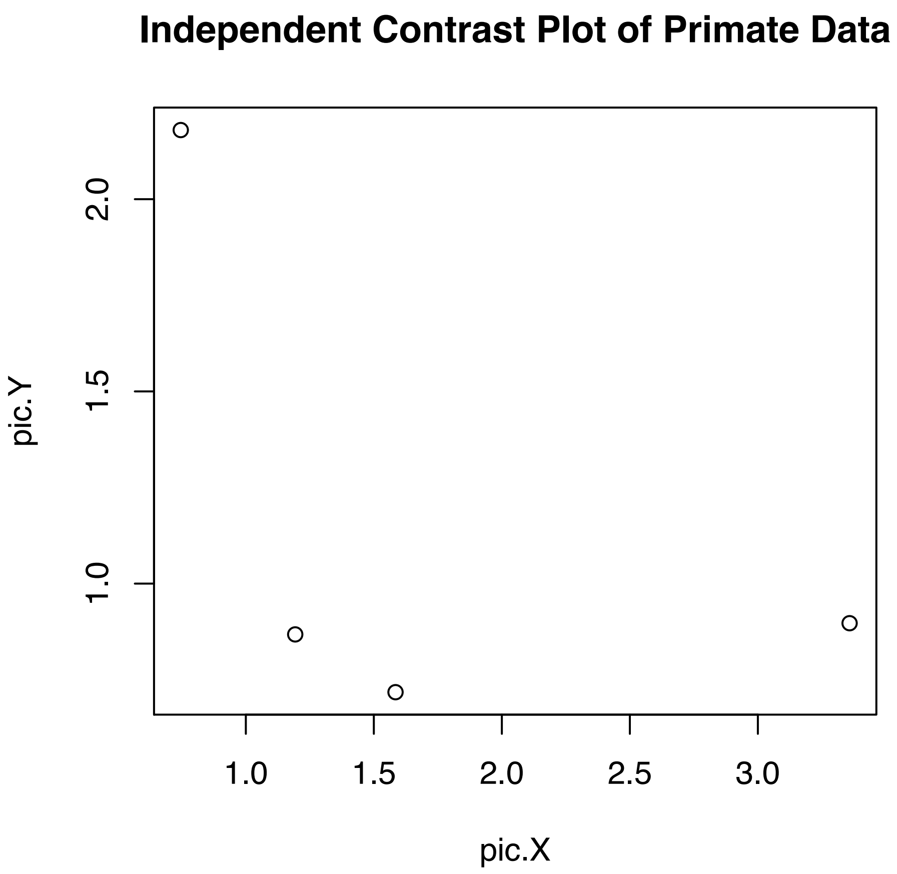


Figure 3.3: A plot of the independent contrasts on X, Y with phylogeny tree.primate.

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:

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

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 (available in package `nlme`), in which you can specify a matrix of correlated errors (in other words, it doesn't assume that correlations among observations are zero):

```
> require(nlme)
> 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

    Ancestral Character Reconstruction
```

```
Call: ace(x = X, phy = tree, type = "continuous")

    Log-likelihood: -6.714469

$ace
       6        7        8        9
1.183725 2.192018 2.571320 3.503182

$sigma2
[1] 1.9711502 0.6970463

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


> ancstatesPIC


    Ancestral Character Reconstruction

Call: ace(x = X, phy = tree, type = "continuous", method = "pic")

    Log-likelihood:

$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
```

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.

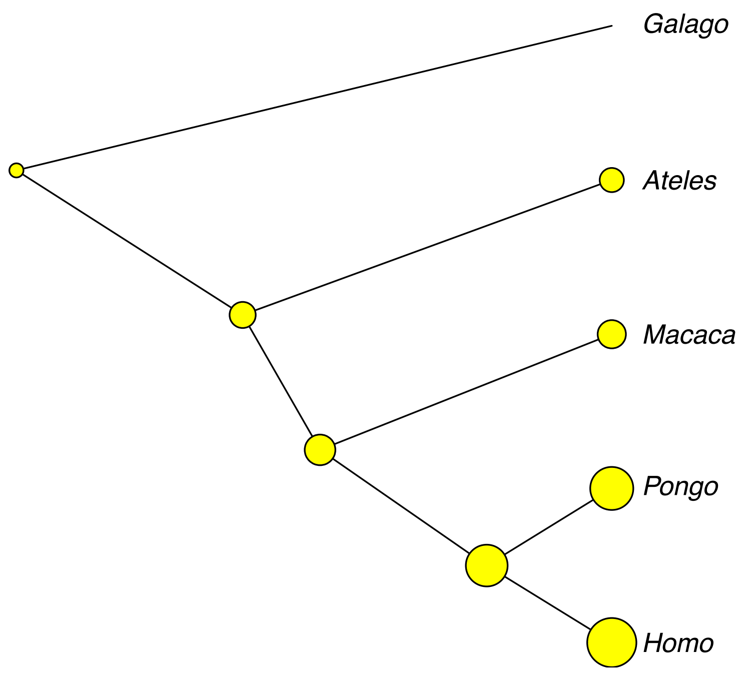
loglik if method = ”ML”, the maximum log-likelihood.

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")
```

End by saving your history.

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

Chapter 4

Finding 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.

Every object that comes with the R program is documented in some way – this means every function, internal dataset, as well as methods and classes (which we won't have time to cover).

4.1 When you know the name of the function

Say you want to find the mean of your data, so you guess that there is a function called `mean()`. Finding help is easy:

```
> ?mean
```

Will bring up the help page, and is equivalent to:

```
> help(mean)
```

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).

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.

There are also hyperlinks in many help documents, to related help pages, so you can “surf” you way through help.

4.2 Don’t know the name of the function

But first to access the help for a specific function, you need to know what it is called.

Two good options are:

```
> help.start()
```

Which will bring up an an html browser, which you can browse. Click on “Packages”, then “base” if you are looking for a basic function that should be in the base distribution of R. Click on the package name if you are looking for a function in a package. Browsing through the help is very useful for beginners.

```
> help.search("plot")
```

Will do a “fuzzy” search (i.e., will also match words close in spelling – not exact – to plot). Of course, replace ”plot” with whatever you are looking for. This function searches through the full text of the help docs, so for a common word like plot, this will return a huge list, which you can look through package by package.

4.3 Package-specific help

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)
```

Here's a harder example. you might want to know more about the phylogeny plotting function in `ape`. If `tree` is a tree object 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)` or `?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, so that there is no function in ape named "plot" (R requires all named functions in packages to be documented).

Huh? How does `plot()` plot a phylogenetic tree when there is no function called `plot` in `ape`? This is an example of a *generic* function. The function `plot` is actually a generic, with different specific functions for different types of objects – R automatically chooses the correct one by looking at the objects `class`.

Anyway...

You have a couple more options (in addition to the general options above):

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 5

Creating Data Objects and Plotting

5.1 Data objects

Now that you have been introduced to R's data objects, we'll practice creating them. R has a rich collection of functions which are very helpful for creating and manipulating objects, so a bit of code can substitute for whole lot of typing!

The tables below list some helpful functions. Look up help for anything you don't know. It will soon start making sense!

commands	actions
c(n1, n2, n3)	combines elements into an object
cbind(x, y)	binds objects together by column
rbind(x, y)	binds objects together by row

Table 5.1: Common combine functions used for creating data objects from existing objects

commands	actions
seq()	generate a sequence of numbers
1:10	sequence from 1 to 10 by 1
rep(x, times)	replicates x
sample(x, size, replace=FALSE)	sample size elements from x
rnorm(n, mean=0, sd=1)	draw n samples from normal distribution

Table 5.2: Functions used for creating sequences and sampling

Factors are categorical data, for example, "large" and "small", or "blue", "red"', and "yellow". Factors may be ordered, which means that the order of the categories has meaning (like size categories). By default, factors are unordered. Levels are the values (i.e., names of the categories) that the factor can take.

commands	actions
vector()	create a vector
matrix()	create a matrix
data.frame()	create a data.frame
as.vector(x)	coerces x to vector
as.matrix(x)	coerces to matrix
as.data.frame(x)	coerces to data frame
as.character(x)	coerces to character
as.numeric(x)	coerces to numeric
factor(x)	creates factor levels for elements of x
levels()	orders the factor levels as specified

Table 5.3: Functions used for creating and coercing objects to new type/class

Examples

To get you started, here are some examples. Creating vectors:

```
> x <- c( 1, 5, 7, 14)
> x

[1]  1  5  7 14

> x <- rep( x, times=2)
> x

[1]  1  5  7 14  1  5  7 14

> y <- rnorm(8)
> y

[1] -0.0599902  0.5525935 -1.1080165 -1.0950411  0.6993651  0.7515395 -0.8649139
[8] -2.2238107

> species <- letters[1:4]    # special stored data object: lower case letters a - d
> species

[1] "a" "b" "c" "d"

> LETTERS[1:3]   # A B C
```

```
[1] "A" "B" "C"

> treatment <- c("high", "med", "low")
> treat <- factor(treatment)   # create a factor
> treat

[1] high med  low
Levels: high low med

> as.numeric(treat)   # coerce to numeric

[1] 1 3 2

> x <- factor(x)   # factor
```

Notice that your work is only saved if you STORE the result in an obect

Creating a matrix:

```
> xy <- cbind(x,y)  # column bind
> xy

     x          y
[1,] 1 -0.0599902
[2,] 2  0.5525935
[3,] 3 -1.1080165
[4,] 4 -1.0950411
[5,] 1  0.6993651
[6,] 2  0.7515395
[7,] 3 -0.8649139
[8,] 4 -2.2238107

> z <- matrix(1:25, nrow=5)  #create a matrix with 5 rows
> z

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

Creating a data matrix:

```
> dat <- data.frame(species, x, y)
```

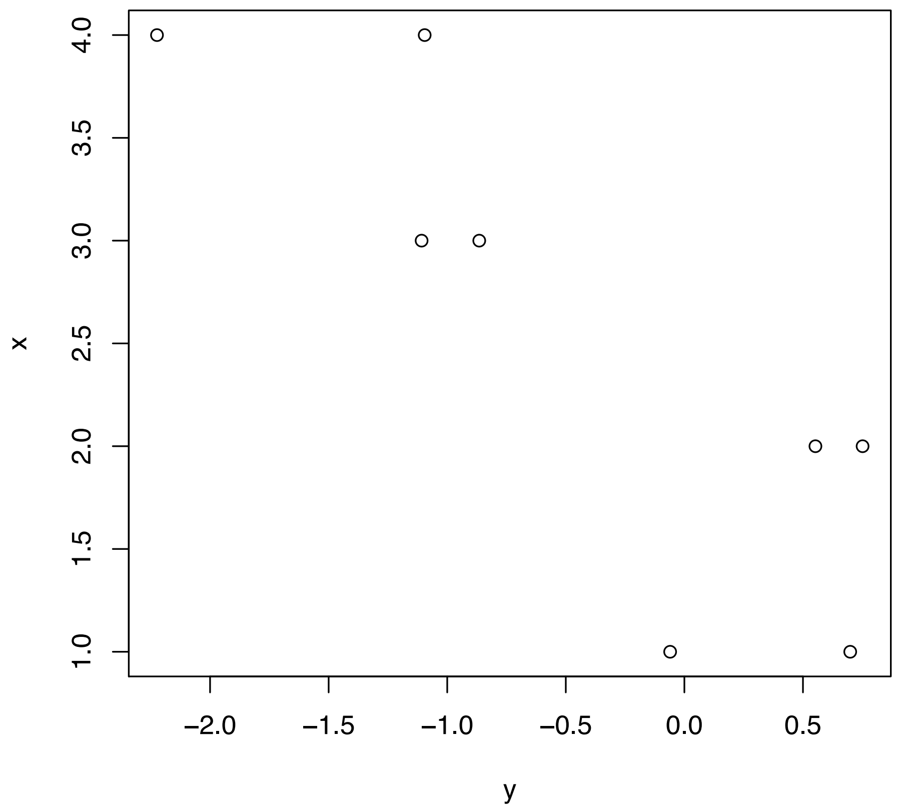
5.2 Simple plotting

The generic function for plotting in R is `plot`.

5.2.1 Bivariate plot

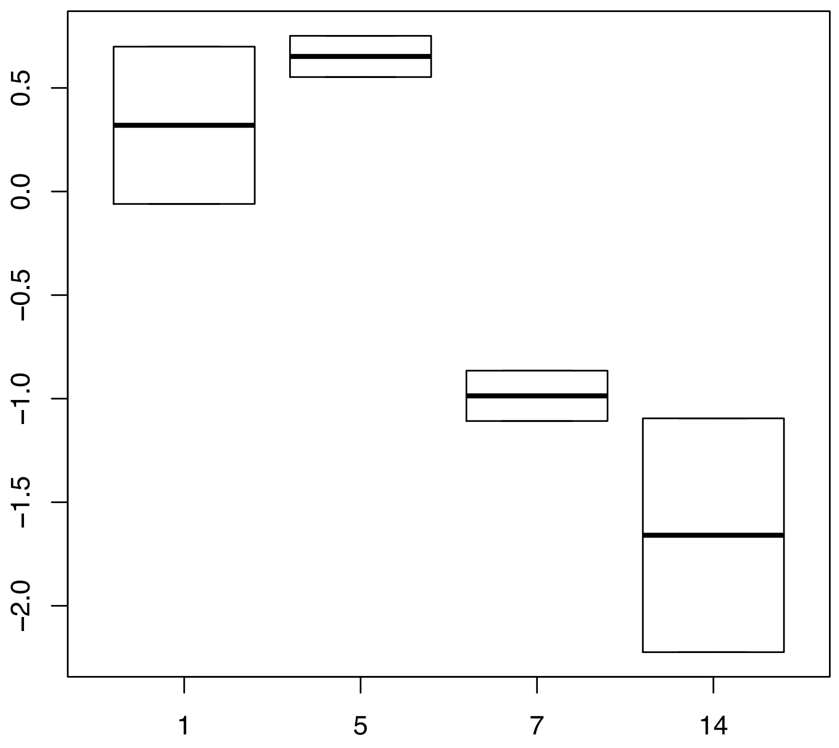
When you supply two vectors to plot, is assumes that the first one is the X coordinate, and the second is the Y. If the first object is a continuous variable, you will get a scatterplot.

```
> plot(y,x)   # continuous variable first - plots as a scatterplot
```



However, if the first object is a factor, you will get a boxplot.

```
> plot(x, y)   # categorical variable first - plots as a boxplot
```

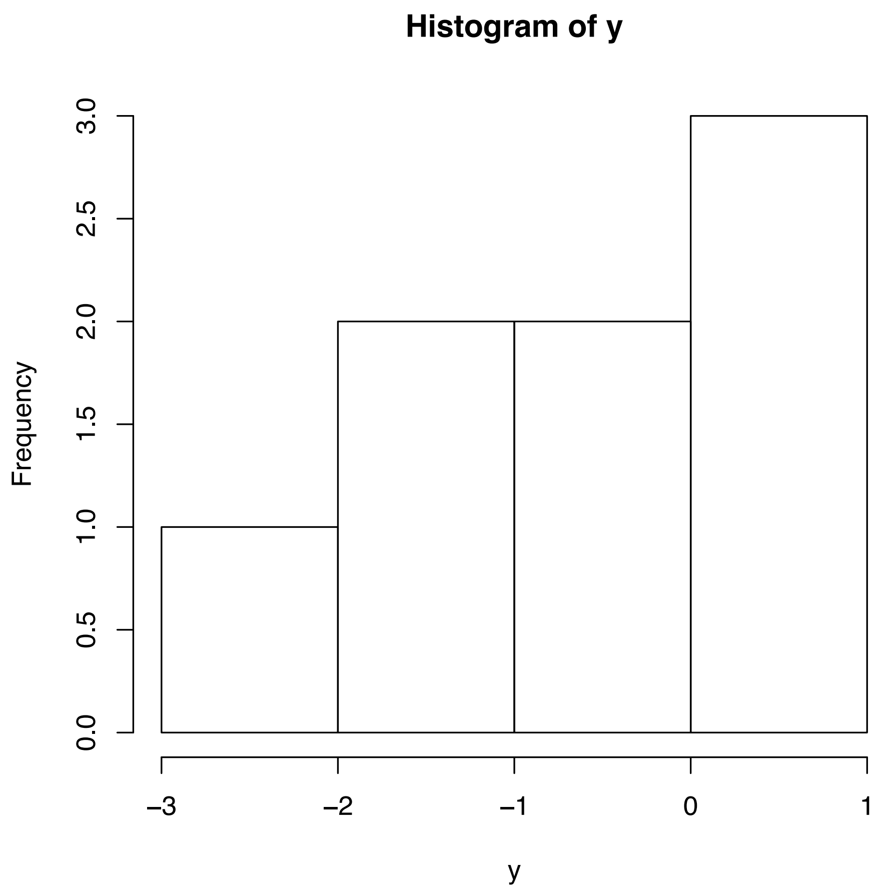


Plot has a huge number of options for changing the symbols (see `?points`, color, size of symbols, axes. labels, adding regression lines or straight lines, etc. Creating multiple panels on a page, etc. Help pages you may want to visit include `?lines`, `?abline`, `?par`, `?axis`.

5.2.2 Univariate plot

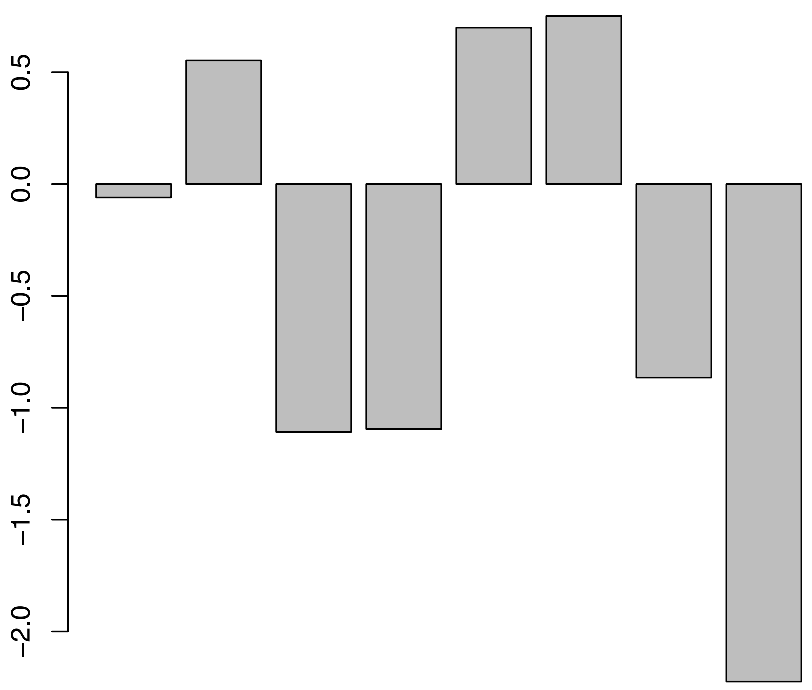
To plot a histogram, use:

```
> hist(y)
```



To plot a bar plot, use:

```
> barplot(y)
```



Practice

1. Create a dataset with simulated data using `rnorm()`.

   (a) Simulate 21 random data points drawn from a normal distribution (create a numeric vector), and save it in the variable "y". Create a second set of 21 points and save it as "y1".

```
> y <- rnorm(21)
> y1 <- rnorm(21)
> y
```

```
 [1] -0.168984468  1.744057572 -0.973792381  2.181253325 -0.271815623
 [6] -1.519178459  0.464675948  0.441336004 -1.269421076 -0.473276722
[11]  0.487935722 -0.291422130 -0.165038976  1.998638212  0.305123817
[16]  0.003532067 -0.967991230 -0.355547127  0.331657651  0.162875050
[21] -1.555495898
```

```
> y1
 [1]  0.41339352  0.82268376  2.30615544  0.63824071  0.80825783  1.07801740
 [7] -0.15137476 -1.05348975 -0.21766137  2.60354690 -1.02202193 -0.08451111
[13] -0.76071467 -1.37201083 -0.86821755 -0.27252772  0.38823803  1.06362074
[19] -0.09175088  1.78317876  0.67630249
```

(b) Create a treatment vector with levels "low", "med", and "high", save it as a factor.

```
> treatment <- factor( c("low", "med", "high") )
> treatment
[1] low  med  high
Levels: high low med
```

(c) Our treatment has numeric values also, so create a numeric vector with the values 2, 4, 8, save it as x.

(d) Create a species vector with seven names.

(e) Create a matrix with y in the first column and x in the second column, save it as dat.matrix.

(f) Create a data frame with species, x, treatment, y and y1, save as dat. Why can't you make a matrix with these columns?

(g) Make a bivariate plot of the numeric value of the treatment (x) versus the response (y). You may want to check the help documentation for "plot". You will have to select the columns of the data frame.

(h) Make a plot on the treatment as factor versus the response. What is the difference between these two plots?

(i) Is the factor displayed in the plot in the order that makes sense? If not, fix this by applying factor to the treatment column of dat again, but this time specifying the levels vector with names of the levels in the order you want. You may want to look at the help page for factor. Plot it again.

(j) Let's make a scatterplot (`plot(y, y1)`) to see if there is any structuring in the data (eventually with respect to the treatment levels – the rest of this exercise is in the chapter on Workhorse Functions of Data Analysis). While we're at it, let's make it prettier. Change the symbols to solid circles by adding the optional parameter `pch=16`, and the points bigger by `cex=2`. Change the color to red using `col="red"`.

(k) Now let's make some data which should differ. For the "low" treatment, simulate y and y1 as normally distributed data with mean = -2 and sd=.5, and "high" as mean=5, and sd=3. Remake the dataframe.

(l) Now make two boxplots: treatment vs. y and treatment vs. y1.

(m) Make boxplots of species vs. y and species vs. y1. Why would you make this plot?

Chapter 6

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?
4. And perhaps most importantly – can you reproduce it?

To be able to accomplish these goals, you want to create clean scripts. Scripts are lines of code saved in an ordinary text file with a `.R` or `.r` ending. (Make sure it is plain text, and NOT a `.rtf`, or a `.doc`, etc file).

All good script follows the first three R's, as you increase along the path of R jedi-hood, you will add on the 4th R:

1. Readable – If you look at the script in a month or 6 months, will you be able to easily understand it?
2. Right – Does it run free of error, and does it produce correct results?
3. Repeatable – Can you reproduce your results from your input data?
4. Reusable – Is your coding modular and designed well so that your code can interact with other scripts, and/or use it for other purposes?

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 from your history file into here. Let's make a script for the analyses we've done thus far.

6.1 Script template

First, make sure that you are in the directory that you want the script to execute from (`Rclass`). 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:

Here is the basic structure of a script:

```
> require( ...addonpackage... )    # anything between ... needs to be changed
>                            # if none, then you don't need that line
>
> dat <- read.csv(..."your input file.csv"... )      # input data
>
> # Your lines of code to run analyses
> # You may have output or processed data that you want to save,
> # create an object for it and write it out to a csv file at the end
>
> # plot graphics
>
> write.csv(out, file="myoutput.csv")      # output data
```

And here is a simple example script that reads in data, calculates summary statistics, a linear regression, and a couple of figures.

```
> #require(stats)        # stats is part of the base package and doesn't need to be loa
>                 # but if you need an add-on package, you would require it here.
>
> dat <- read.csv ("Data/morphpre.csv")  # read in data
> lm.HLSVL <- lm(dat$HandL ~ dat$SVL)   # run a linear model
> summary(lm.HLSVL)        # get summary statistics
> str(lm.HLSVL)        # look at the linear model object
> coef(lm.HLSVL)[2]        # get the slope of the regression
> plot(dat$HandL ~ dat$SVL, cex=2)    # make a plot with big dots (cex controls size o
> abline(lm.HLSVL, col="red")              # plots the regression line, in red
>   title("Microhylid Hand Length vs Body Size")        # add a title
>   text(x=15, y=13, paste("slope = ", coef(lm.HLSVL)[2]))
>                   # add important info to the text
+
+ ###
+ # please insert your other lines of code here -- enough
+ #         to save a meaningful analysis
+ ###
```

Note that I have used spacing and indents to increase the "readability" of the code. Use it to set of blocks of code that accomplish one task, with indents to indicate heirarchy. We will talk more about this in the functions section.

Save the script file as "`testScript.R`" or an appropriate title in your `Rclass` folder. Now if you want to run the code, you simply type at the R console (from within your Rclass directory):

```
> source("testScript.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.

6.1.1 Writing pdf to file

If you'd like to print your pdf to a file instead of to the screen, you can add the following code into your script:

```
> pdf(file="MicrohylidHandLvsSize.pdf")        # open pdf device for printing
>   plot(dat$HandL ~ dat$SVL, cex=2)        # remake plot as before
>   abline(lm.HLSVL, col="red")
>     title("Microhylid Hand Length vs Body Size")
>     text(x=15, y=13, paste("slope = ", coef(lm.HLSVL)[2]))
> dev.off()        # turn off pdf device so future plots go back to screen
```

6.1.2 History file

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 – you can see it from the terminal by using the `ls -a` command). 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.

Let's use a built-in dataset called `iris`, which is the famous Fisher iris dataset. Make a test script file and save it as `test.R`:

```
> names(iris)          # will not print to console when sourced
> spp <- unique(iris$Species)  # only unique values
> spp <- as.character(spp)     # factor -> character
> spp                  # will not print to console when sourced
> print('Species names')  # will print
> print(spp)               # will print
> cat('\n', 'Species names =', spp)  # concatenate
>                  # \n is a carriage return character
> summary(iris)
```

Then test it by running:

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

```
[1] "Species names"
[1] "setosa"     "versicolor" "virginica"

 Species names = setosa versicolor virginica
```

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.

6.2 Remember the workspace

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.

In general, most of my analyses are pretty quick in terms of computer time (not programming time!). So I never save my workspace, because I don't want to deal with any "ghost" objects I have forgotten about. Instead, I write a nice script that will generate the whole analysis. If it's a really big complex analysis, you can save intermediate output as r data files (more on this later).

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

6.3 Exercises

1. Create a script of the work we've done so far.

2. R has great diagnostic plots for linear models. Read about them in the help page for ?plot.lm and incorporate a multi-panel figure by adding two lines of code to the script you've already made:

```
> par(mfrow = c(2,2))   # set the plot environment to have two rows and two colum
> plot(lm.HLSVL)
```

3. Save output to a file.

4. Modify 'test.R' so that a summary of the iris data prints to the console when sourced.

5. Explore other datasets in R. At the R command prompt type `data()` to see what is available.

Chapter 7

The Workhorse Functions of Data Manipulation

Chapter Topics/Skills:

Indexing/Subsetting accessing particular elements of your data object

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.

7.1 Indexing and subsetting

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

7.1.1 Vectors

The "index" of a vector is it's number in the order. Each and every element in any data object has at least one index (if vector, it's position along the vector, if a matrix or data frame, it's row and column number, etc.)

Let's create a vector:

```
> xx <- c(1, 5, 2, 3, 5)
> xx
```

```
[1] 1 5 2 3 5
```

Access specific values of `xx` by number:

```
> xx[1]
```

```
[1] 1
```

```
> xx[3]
```

```
[1] 2
```

You can use a function to generate an index. Get the last element (without knowing how many there are) by:

```
> xx[length(xx)]
```

```
[1] 5
```

Retrieve multiple elements of xx by using a vector as an argument:

```
> xx[c(1, 3, 4)]

[1] 1 2 3

> xx[1:3]

[1] 1 5 2

> xx[c(1, length(xx))]  # first and last

[1] 1 5
```

Exclude elements by using a negative index:

```
> xx

[1] 1 5 2 3 5

> xx[-1]  # exclude first

[1] 5 2 3 5

> xx[-2] # exclude second

[1] 1 2 3 5

> xx[-(1:3)] # exclude first through third

[1] 3 5

> xx[-c(2, 4)] # exclude second and fourth, etc.

[1] 1 2 5
```

Use a logical vector:

```
> xx[ c( T, F, T, F, T) ]  # T is the same as TRUE
```

```
[1] 1 2 5
```

```
> xx > 2
```

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

```
> xx[ xx > 2 ]
```

```
[1] 5 3 5
```

```
> xx > 2 & xx < 5
```

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

```
> xx[ xx>2 & xx<5]
```

```
[1] 3
```

Subsetting (picking particular observations out of an R object) is something that you will have to do all the time. It's worth the time to understand it clearly.

7.1.2 Matrices and Dataframes

Matrices and dataframes are both rectangular having two dimensions, and handled very similarly For indexing and subsetting. Let's work with a dataframe that is provided with the `geiger` package called `geospiza`. It is a list with a tree and a dataframe. The dataframe contains five morphological measurements for 13 species. First, let's clear the workspace (or clear and start a new R session):

If you have the package `geiger` installed, get the built-in dataset this way:

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

```
[1] "geospiza"
```

Let's find out some basic information about this object:

```
> class(geospiza)

[1] "list"

> attributes(geospiza)

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

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

```
> geo <- geospiza$geospiza.data
> dim(geo)

[1] 13  5
```

You can also read it in as a .csv input file in the Data directory and proceed.

```
> geo <- read.csv("Data/geospiza_raw.csv")
> dim(geo)
```

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

$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).

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
```

To prove to ourselves that we can access matrices in the same way, let's coerce geo to be a matrix:

```
> geom <- as.matrix( geo )
> class(geom)

[1] "matrix"

> class(geo)

[1] "data.frame"

> geo[1,5]  # try a few more from the choices above to test

[1] 2.675983
```

Since `geo` and geom have row and column names, we can access by name (show that this works for `geom` too):

```
> geo["pauper", "wingL"]  # row pauper, column wingL

[1] 4.2325

> geo["pauper", ]  # row pauper, all columns

        wingL tarsusL culmenL  beakD gonysW
pauper 4.2325  3.0359   2.187 2.0734 1.9621
```

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

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

[1] "pallida" "fusca"   "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 3.089450 2.430250 2.016350 1.949125
fusca   3.975393 2.936536 2.051843 1.191264 1.401186
pauper  4.232500 3.035900 2.187000 2.073400 1.962100
```

One difference between dataframes and matrices is that 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
```

Prove to yourself that selecting by a single index has a different behavior for matrices (and sometimes produces an error. Why? Because internally, a dataframe is actually a list of vectors. Thus a single name or number refers to the column, rather than a coordinate in a cartesian-coordinate-liek system. However, a matrix is actually a vector with breaks in it. So a single number refers to a position along the single vector. A single name could work, but only if the individual elements of the matrix have names (like naming the individual elements of a vector).

Another difference is that dataframes (and lists below) can be accessed by the `$` operator. It means indicates a column within a dataframe, so `dataframe$column`. This is another way to select by column:

```
> geo$culmenL
```

```
 [1] 2.724667 2.654400 2.277183 2.621789 2.407025 2.094971 2.430250 2.051843
 [9] 1.974420 2.187000 2.311100 2.331471 2.259640
```

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`.

7.1.3 Lists

A list is like a vector, except that whereas a vector has the same type of data (numeric, character, factor) in each slot, a list can have different types in different slots. They are sort of like expandable containers, flexibly accommodating any group of objects that the user wants to keep together.

They are accessed by numeric index or by name (if they are named), but they are accessed by double square brackets. Also, you can't access multiple elements of lists by using vectors of indices:

```
> mylist <- list( vec = 2*1:10, mat = matrix(1:10, nrow=2), cvec = c("frogs", "birds")
> mylist


$vec
 [1]  2  4  6  8 10 12 14 16 18 20

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

$cvec
[1] "frogs" "birds"


> mylist[[2]]


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


> mylist[["vec"]]


 [1]  2  4  6  8 10 12 14 16 18 20
```

```
> # mylist[[1:3]]  # gives an error if you uncomment it
> mylist$cvec

[1] "frogs" "birds"
```

7.2 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  9 10 12 13

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

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

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

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

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

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

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

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

7.3 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]  2  3  5  6  8  1  7  9 10 11 12 13  4

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

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

> 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.

7.4 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?
```

7.5 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
```

7.6 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
```

Practice

1. Recall from the chapter on Data Objects that we were simulating data in different treatment groups, and wanting to visualize the groups. Now that we know how to index and subset, we can use the `points` function to add different colored points to the plot for different groups.

   (a) Now let's make some data which should differ. For the "low" treatment, simulate y and y1 as normally distributed data with mean = -2 and sd=.5, and "high" as mean=5, and sd=3. Remake the dataframe.

```
> species <- LETTERS[1:7]
> x <- c(2, 4, 8)
> y <- c(rnorm(7, mean=-2, sd=0.5), rnorm(7), rnorm(7, mean=5, sd=3))
> y1 <- c(rnorm(7, mean=-2, sd=0.5), rnorm(7), rnorm(7, mean=5, sd=3))
> y
```

```
 [1] -1.7324331 -1.6198601 -1.6943661 -1.5550680 -1.5659884 -2.6173233
 [7] -2.8677405  0.7436305  0.1378092 -0.5159335 -1.5854286 -0.1596901
[13]  1.6204093  1.8036682  5.4798608  2.8345419  5.9276164  4.8582151
[19]  3.1012095  8.3641030  0.1240776
```

```
> y1
```

```
 [1] -2.7783919 -2.0899730 -2.1505971 -2.0055296 -1.5302355 -2.0239404
 [7] -1.8948294  0.4669686 -0.3045481  0.0534605 -1.6229322 -1.6686932
[13]  0.2173391  0.4160709  3.1820160  6.2624768  5.0295437  5.3543262
[19]  1.9442462 11.2799850 -1.9427580
```

```
> dat <- data.frame(species, x, treatment=factor(rep(c("low", "med",
+  "high"), each=7), levels=c("low", "med", "high")), y, y1)
> dat
```

```
   species x treatment          y         y1
1        A 2       low -1.7324331 -2.7783919
2        B 4       low -1.6198601 -2.0899730
3        C 8       low -1.6943661 -2.1505971
4        D 2       low -1.5550680 -2.0055296
5        E 4       low -1.5659884 -1.5302355
6        F 8       low -2.6173233 -2.0239404
7        G 2       low -2.8677405 -1.8948294
8        A 4       med  0.7436305  0.4669686
9        B 8       med  0.1378092 -0.3045481
10       C 2       med -0.5159335  0.0534605
11       D 4       med -1.5854286 -1.6229322
12       E 8       med -0.1596901 -1.6686932
13       F 2       med  1.6204093  0.2173391
```

```
14       G 4       med  1.8036682  0.4160709
15       A 8      high  5.4798608  3.1820160
16       B 2      high  2.8345419  6.2624768
17       C 4      high  5.9276164  5.0295437
18       D 8      high  4.8582151  5.3543262
19       E 2      high  3.1012095  1.9442462
20       F 4      high  8.3641030 11.2799850
21       G 8      high  0.1240776 -1.9427580
```

(b) Let's differentially color the "high", "medium", and "low" points. First set up the plot window without any points by plotting y, y1 with the plot parameter `type="n"`. Then select only the "high" points by subsetting. You'll want to make an index vector to choose only the points you want. Then use the `points()` function (which has the same form as the `plot()` function, but only adds points to an existing plot. Choose three different colors for each treatment level and plot all the data. Is there any patterning in y, y1?

(c) Ooops! The data are actually supposed to be blocked by treatment (the first seven rows correspond to low, the second 7 correspond to med, etc.) Can you remake the dataframe keeping the y and y1 in the same position, but fixing the treatment?

(d) Make three plots: boxplot of treatment vs. y, treatment vs. y1, and three color scatterplot of y vs. y1 (treatments should be indicated by different colors).

2. Matrix reshaping and indexing

   (a) Create a matrix with the values 1 through 20, filling four rows. Save it as "x". item What are the attributes of x?

   (b) Change it to a matrix with 2 rows and 10 columns by changing its attribute. item Change x to a vector.

   (c) Change x to a matrix with four rows, this time filling it by rows rather than by columns (you may want to check the help page).

   (d) Coerce x to a vector again. Is it in the same order as the previous vector? What does this tell you about R's default behavior when flattening matrices to vector?

   (e) Create the original x matrix again. Select only the 3rd row, 4th column. What is it?

   (f) Select rows 3 and 4, columns 4 and 5. Print it to the console by using the `print(x)` function.

   (g) Select the first and last rows, first and last columns. Print it.

3. Reading in Data and adding on

(a) Read in the external file `bimac.csv` in comma separated format. Save it as "`bimac`".

(b) This is a phylogenetic tree and data for the OUCH package. Without going into details for now, this method allows biologists to specify selective regimes on branches of the phylogeny, by specifying categories which correspond to alternative "niches". This is a body size evolution dataset, and "OU.LP" is a hypothesis with three size categories. We would like to make three additional hypotheses. Add additional columns to this dataframe: OU.1 which has values of "`global`" for all rows, OU.3 which is the same as OU.LP, except those rows with "`NA`" in the species names should get a value of "`medium`", and OU.4 which is again similar to OU.LP, except that those rows with "`NA`" in the species names get a value of "`anc`".

Chapter 8

Data Input and Output

So far, we have been working within R, either typing data in directly or using R's functions to generate data. In order to analyze your own data, you have to load data from an external file into R. Similarly, to save your work, you'll probably want to write files from R to your hard drive. Both of these require interacting with your computer's operating system. In this chapter, we're just going to do it. We'll talk more about what's going on in a later section on the R Environment.

8.1 Getting your data into R

The most convenient way to read data into R is using the `read.csv()` function. This requires that your data is saved in .csv format, which is possible from Microsoft Excel (save as... csv) or any spreadsheet format. It is a text format with data separated by commas. It is very nice because it is unambiguous, not easily corruptible, and non-proprietary. Thus it is readable by nearly every program that reads in data.

First, within your "Rclass" folder, create a folder named "Data". Copy the file "anolis.csv" and "Iguanamass.csv" into this folder.

Next, from within R, check which working directory you are in. You should be in your Rclass folder. If you are not, use `setwd()` to get there.

```
> getwd()
> setwd("~/Rclass")  # my folder is at the top level of my user directory
```

8.1.1 read.csv

Getting the file in is easy. If it is in csv format, you just use:

```
> read.csv("Data/anolis.csv")  # look for the file in the Data directory
```

This is an *Anolis* lizard sexual size dimorphism dataset. It has values of dimorphism by species for different ecomorphs, or microhabitat specialists.

To save the data, give it a name and save it:

```
> anolis <- read.csv("Data/anolis.csv")
```

It is a good practice to *always* check that the data were read in properly. If it is a large file, you'll want to at least check the beginning and end were read in properly:

```
> head(anolis)
```

```
  species   logSSD    ecomorph
1      oc -0.00512        twig
2      eq  0.08454 crown-giant
3      co  0.24703 trunk-crown
4     aln  0.24837 trunk-crown
5      ol  0.09844  grass-bush
6      in  0.06137        twig
```

```
> tail(anolis)
```

```
   species  logSSD     ecomorph
18      cr 0.39796 trunk-ground
19      st 0.15737  trunk-crown
20      cy 0.26024 trunk-ground
21     alu 0.08216   grass-bush
22      lo 0.13108        trunk
23      an 0.13547         twig
```

Voila! Now you can plot, take the mean, etc. Which prints out the first six and last six lines of the file.

R can read in many other formats as well, including database formats, excel native format (although it is easier in practice to save as .csv), fixed width formats, and scanning lines. For more information see the R manual "R Data Import/Export" which you can get from `help.start()` or at http://www.r-project.org.

8.2 Summary statistics on your data

Suppose you wanted to compute and save the means and standard deviations for the sexual size dimorphism values. A very convenient function for computing any function over groups in your dataframe (here, ecomorphs), is the function `aggregate` (look up help via `?aggregate`).

Calculate the mean by ecomorph group:

```
> aggregate(anolis$logSSD, by=list(anolis$ecomorph), mean)
```

```
       Group.1         x
1  crown-giant 0.1391750
2   grass-bush 0.1437525
3        trunk 0.1467167
4  trunk-crown 0.2626575
5 trunk-ground 0.3339650
6         twig 0.0848450
```

Notice we had to type `anolis$` in front of the variables we wanted. This is because these vectors are within the dataframe `anolis`. To be able to access `anoliś` goodies, we need to tell R where to look (more on this later).

Notice that the argument to by, which groups we want the mean over, has to be a list, so we coerced the variable `anolis$ecomorph` into a list.

Calculate the mean and the sd by ecomorph group, and this time save them:

```
> anolis.mean <- aggregate(anolis$logSSD, by=list(anolis$ecomorph), mean)
> anolis.sd <- aggregate(anolis$logSSD, by=list(anolis$ecomorph), sd)
> anolis.sd
```

```
       Group.1          x
1  crown-giant 0.09909567
2   grass-bush 0.06924584
3        trunk 0.02136480
4  trunk-crown 0.09968872
5 trunk-ground 0.06966130
6         twig 0.07107131
```

Give the results of aggregate meaningful column names:

```
> names(anolis.mean)   # check that this is what we want to modify
```

```
[1] "Group.1" "x"
```

```
> names(anolis.mean) <- c("ecomorph", "mean")
> names(anolis.sd) <- c("ecomorph", "sd")
```

While we're at it, let's get the sample size so that we can calculate the standard error, which is the standard deviation divided by the square root of the sample size.

```
> anolis.N <- aggregate(anolis$logSSD, by=list(anolis$ecomorph), length)
> names(anolis.N) <- c("ecomorph", "N")
```

8.2.1 merge

It's not convenient to have so many data objects, what we'd really like is to have all summary statistics together in one data frame. So let's use the `merge` function.

Merge works two objects at a time, and merges by default on the common column names (here, ecomorph):

```
> merge(anolis.mean, anolis.sd)
```

```
      ecomorph      mean         sd
1  crown-giant 0.1391750 0.09909567
2   grass-bush 0.1437525 0.06924584
3        trunk 0.1467167 0.02136480
4  trunk-crown 0.2626575 0.09968872
5 trunk-ground 0.3339650 0.06966130
6         twig 0.0848450 0.07107131
```

Otherwise, you must specify `by=`. Or to be safe, you can specify it, it's good practice:

```
> out <- merge(anolis.mean, anolis.sd, by="ecomorph")
```

There is also options for `by.x=` and `by.y=` in case your columns have different names in the two objects – you can tell R which two columns to match.

Do it again to add the third object, N:

```
> out <- merge(out, anolis.N, by="ecomorph")
> out
```

```
      ecomorph      mean         sd N
1  crown-giant 0.1391750 0.09909567 4
2   grass-bush 0.1437525 0.06924584 4
3        trunk 0.1467167 0.02136480 3
4  trunk-crown 0.2626575 0.09968872 4
5 trunk-ground 0.3339650 0.06966130 4
6         twig 0.0848450 0.07107131 4
```

Now, it's easy to compute the standard error:

```
> out$se <- out$sd / sqrt(out$N)
> out
```

```
      ecomorph      mean         sd N         se
1  crown-giant 0.1391750 0.09909567 4 0.04954783
2   grass-bush 0.1437525 0.06924584 4 0.03462292
3        trunk 0.1467167 0.02136480 3 0.01233497
4  trunk-crown 0.2626575 0.09968872 4 0.04984436
5 trunk-ground 0.3339650 0.06966130 4 0.03483065
6         twig 0.0848450 0.07107131 4 0.03553565
```

8.3 write.csv

Writing out objects is even simpler. To write out a .csv file:

```
> write.csv(out, "anolis.summary.csv", row.names=FALSE)
```

The argument "row.names=" is optional, but I like to put it in or else you get row names added to your spreadsheet as an extra column. Leave it as TRUE (the default) only if the names are meaningful and useful.

8.4 save

You can also save the objects as R data files (.Rdat or .rda), which are R's binary format. The objects are saved directly, so you can just slurp up the .Rdata file and you will have your objects back. This is handy if you want to continue your analysis with your objects later.

```
> save( anolis, anolis.mean, anolis.sd, anolis.N, file="anolis.out.Rdata")
```

The command to load these back in is:

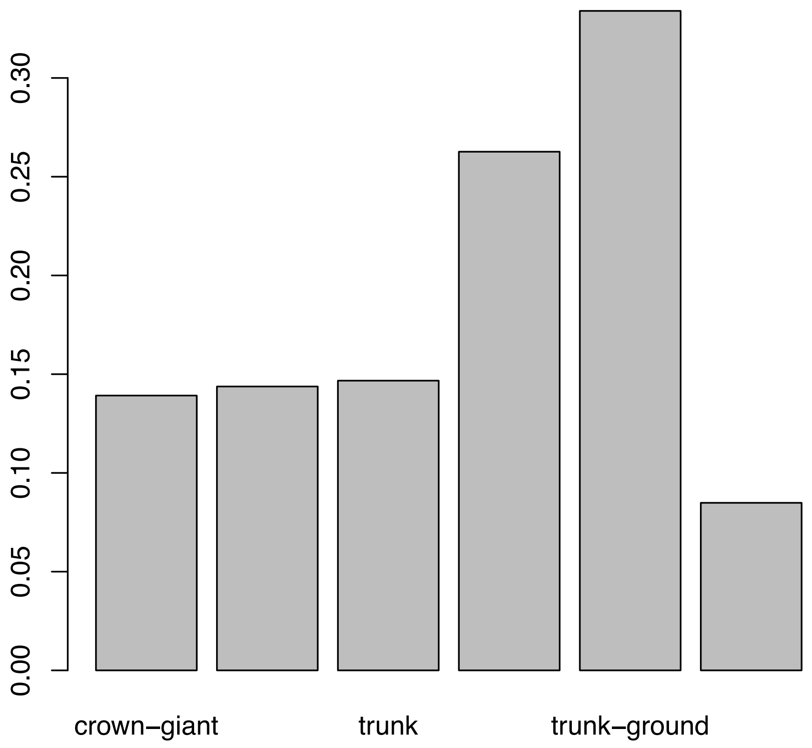
```
> load("anolis.out.Rdata")
```

Which will restore your objects.

8.5 Saving plots

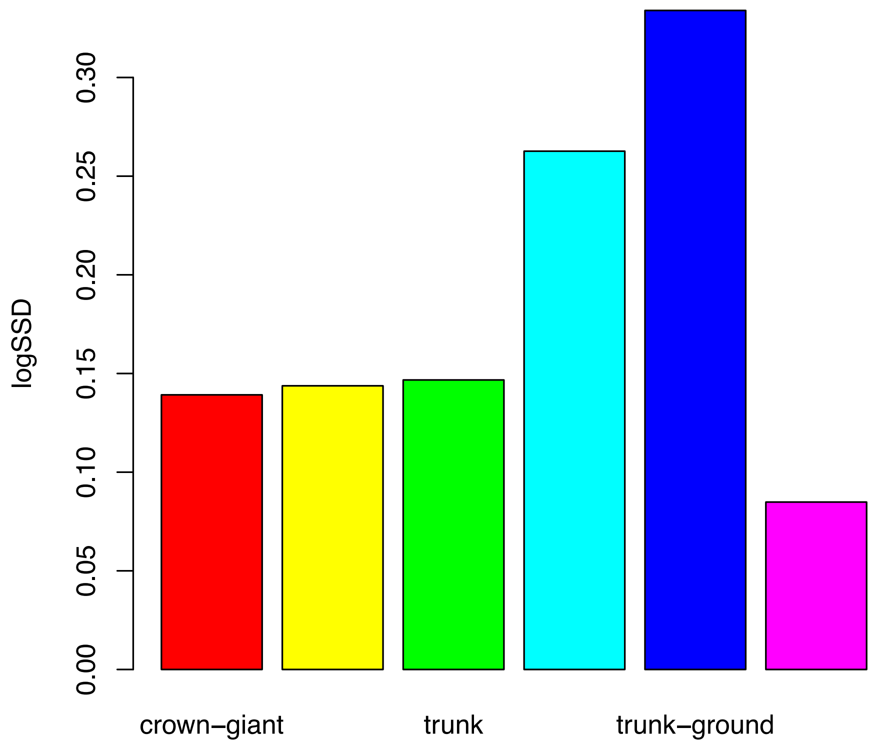
Let's make some plots to visualize SSD by ecomorph type. Recall that we can get box plots (median, quartiles, and range):

```
> barplot(out$mean, names.arg=out$ecomorph)
```



Let's add some color and a label for the y variable. Rainbow is a function which will generate a pallete of colors according to the number of colors you specify.

```
> barplot(out$mean, names.arg=out$ecomorph, col=rainbow(6), ylab="logSSD")
```

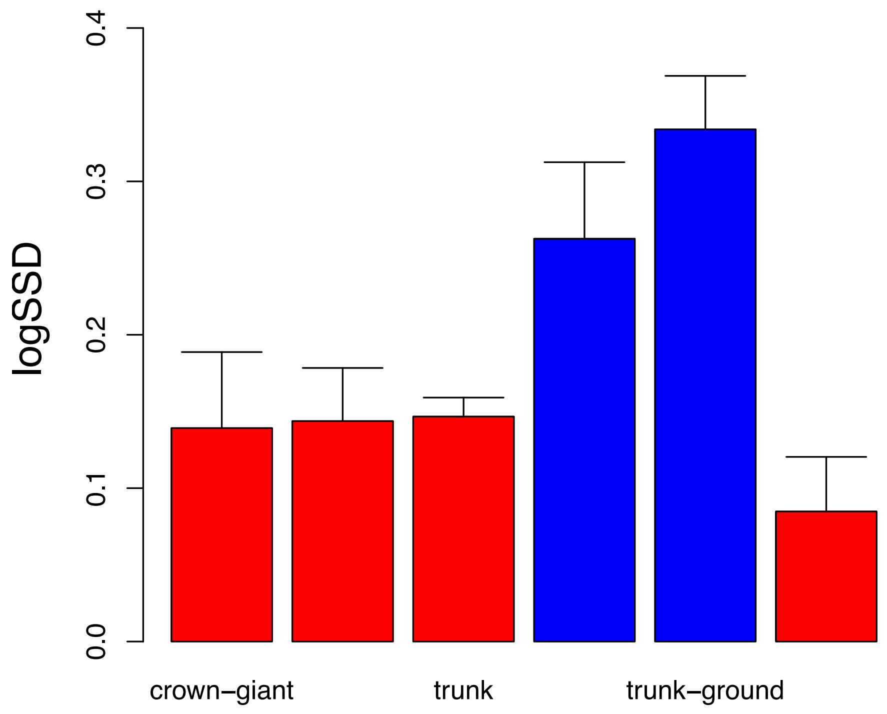


Alternatively, we may want to visually accentuate the "high" versus "low" dimorphism groups (for a talk for instance):

```
> bb <- barplot(out$mean, names.arg=out$ecomorph, col=c("red", "red", "red", "blue",
+  "blue", "red"), ylab="logSSD", cex.lab=1.5, ylim=c(0, max(out$mean)+.1))
> bb
```

```
     [,1]
[1,]  0.7
[2,]  1.9
[3,]  3.1
[4,]  4.3
[5,]  5.5
[6,]  6.7
```

```
> arrows(bb, out$mean, bb, out$mean+out$se, angle=90)
```



We've also made the y-axis label bigger using `cex.lab=1.5`, and finally added error bars by using the `arrows()` function. This function basically draws the error bars as line segments specified by the first four arguments. The `angle=90` tells the function to make the arrow heads flat, as in error bars. Read `?arrows` for more info. Finally, because the graph was not big enough to plot the highest error bar, I had to increase the y-limit using the `ylim` argument, which sizes the y-axis according to the lower and upper bounds given.

8.5.1 pdf

Now, if we are quite happy with our plot, we can save it as a pdf file. First we have to set the graphical devide to a pdf printer. Then plot the file, then turn the pdf device off (or it will keep writing to the same file every time you plot).

```
> pdf(file="anolisMeanSSD.pdf")  # turns on the pdf device for plotting
> barplot(out$mean, names.arg=out$ecomorph, col=c("red", "red", "red", "blue",
+ "blue", "red"), ylab="logSSD", cex.lab=1.5)
> dev.off()  # turns off pdf device for output

quartz
     2
```

8.6 Messier input files

The first example of a csv file was very easy to bring in to R. If it was hand-entered, you may have several issues including:

- extra delimiters in some rows (extra commas, etc.) so that some rows have extra columns
- extra header lines
- lots of missing values
- mixed character and numeric input

Any of these issues will cause problems because what you are reading in is a data frame. R expects columns to be of the same type, and the object is square, and etc.

Extra header lines are really easy to fix using the `skip=` option. However, the other issues will have to be fixed by editing your .csv file, or by writing code that reads in the lines one by one, makes the appropriate changes, and then writing out a "clean" .csv file. Which way to go should be determined by how much work it will be to hand-edit vs. program, which will depend a lot on how many problems the file contains, and whether they are unique or not. (Probably 80% or more of your R programming efforts are aimed at getting your input data into shape for analysis – which is why we cover these in the next section).

8.6.1 Input files generated by data loggers

An easier case to handle: files that are generated by computer. Take, for example, the file format generated from our hand-held Ocean Optics specroradiometer. It is very regular in structure, and we have tons of data files, so it is well worth the programming effort to code a script for automatic file input.

First, you can open the file below in a text editor. If you'd rather open it in R, you can use:

```
> readLines("Data/20070725_01forirr.txt")
```

Notice that there is a very large header, in fact the first 17 lines. Notice also that the last line will cause a problem. Also, the delimiter in this file is tab (backslash t).

```
> temp <- readLines("Data/20070725_01forirr.txt")
> head(temp)
```

```
[1] "SpectraSuite Data File"
[2] "++++++++++++++++++++++++++++++++++++"
[3] "Date: Wed Jul 25 10:39:54 HST 2007"
[4] "User: guest"
[5] "Dark Spectrum Present: Yes"
[6] "Reference Spectrum Present: No"
```

```
> tail(temp)
```

```
[1] "888.38\t3.1306E-01"
[2] "888.54\t2.8153E-01"
[3] "888.71\t2.8245E-01"
[4] "888.87\t1.8988E-01"
[5] "889.04\t1.8988E-01"
[6] ">>>>>End Processed Spectral Data<<<<<"
```

We can solve these issues using the "skip" and the "comment.char" arguments of read.table to ignore both types of lines, reading in only the "good stuff". Also, the default delimiter in this function is the tab:

```
> dat <- read.table(file="Data/20070725_01forirr.txt", skip=17, comment.char=">")
> names(dat) <- c("lambda", "intensity")
> head(dat)
```

```
  lambda intensity
1 177.33         0
2 177.55         0
3 177.77         0
4 177.99         0
5 178.21         0
6 178.43         0
```

```
> tail(dat)
```

```
     lambda intensity
3643 888.21   0.29491
3644 888.38   0.31306
3645 888.54   0.28153
3646 888.71   0.28245
3647 888.87   0.18988
3648 889.04   0.18988
```

The file produces (useless) rows of data outside of the range of accuracy of the spectraradiometer. We can get rid of these by subsetting the data, selecting only the range 300-750nm:

```
> dat <- dat[dat$lambda >= 300, ]  # cut off rows below 300nm
> dat <- dat[dat$lambda <= 750, ]  #cut off rows above 750nm
```

Or do both at once:

```
> dat <- dat[dat$lambda >= 300 & dat$lambda <= 750,]
```

If we are going to be doing this subsetting over and over, we might want to save this as an index vector which tells us the position of the rows of data we want to keep in the dataframe (don't worry, we'll cover this again in the workhorse functions chapter).

```
> oo <- dat$lambda >= 300 & dat$lambda <= 750
> dat <- dat[oo, ]   # same as longer version above
```

We can now save the cleaned up version of the irradiance data:

```
> write.csv(dat, "20070725_01forirr.csv")
```

Chapter 9

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

9.1 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).

9.1.1 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

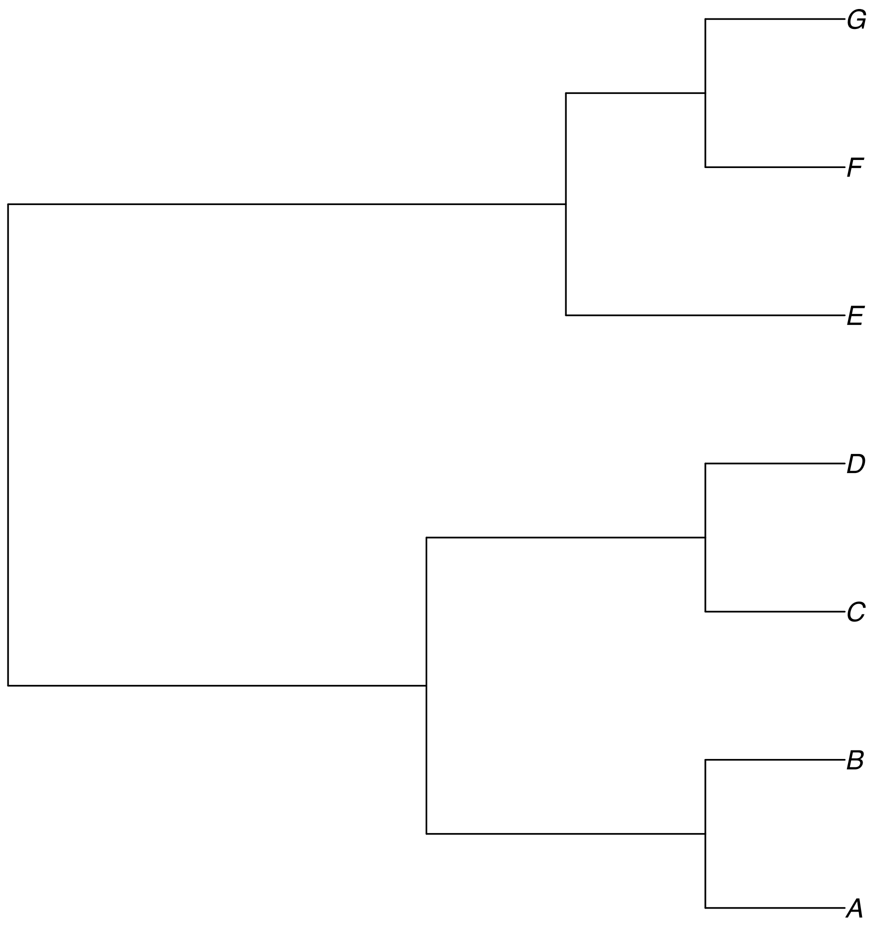


Figure 9.1: A simple tree

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.

9.1.2 phylo (ape 1.9 or above)

The `ape` package (?) 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.

9.1.3 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)
```

9.1.4 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.frame`s 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,]        0          8
 [9,]        8         12
[10,]       12          5
[11,]       12         13
[12,]       13          6
[13,]       13          7

$edge.length
  8-9  9-10  10-1  10-2  9-11  11-3  11-4   0-8  8-12  12-5 12-13  13-6  13-7
  1.5   1.0   0.5   0.5   1.0   0.5   0.5    NA   2.0   1.0   0.5   0.5   0.5

$label
  1   2   3   4   5   6   7
"A" "B" "C" "D" "E" "F" "G"

$edge.label
```

```
named character(0)

$order
[1] "unknown"

$annote
list()

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

9.2 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.

9.2.1 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.

9.2.2 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`.

9.3 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 9.1: 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.

9.4 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()
```

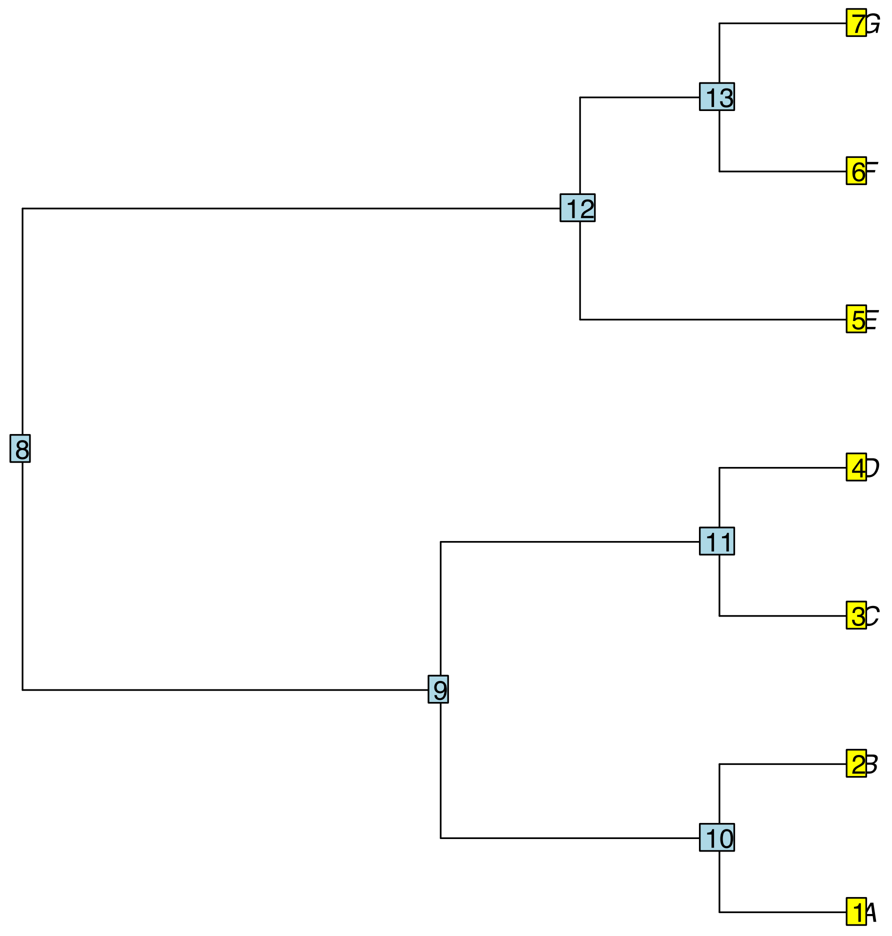


Figure 9.2: A simple tree with ape's numbering of nodes included

Chapter 10

Working with Trees by Michael Alfaro

10.1 Introduction

In this class you will be introduced to a number of comparative methods in R. Nearly all of them will require you to manipulate a phylogenetic tree in some way. This section provides a brief introduction to tree structures in R. We'll learn how to read newick and nexus formatted trees, plot them, and work with branch lengths.

10.2 Getting Started

First lets load APE, GEIGER, and some example data sets

```
> require(geiger) ##load packages
> data(geospiza)          ## data from geiger
> geospiza


$geospiza.tree

Phylogenetic tree with 14 tips and 13 internal nodes.

Tip labels:
        fuliginosa, fortis, magnirostris, conirostris, scandens, difficilis, ...

Rooted; includes branch lengths.
```

```
$geospiza.data
                wingL  tarsusL  culmenL    beakD   gonysW
magnirostris 4.404200 3.038950 2.724667 2.823767 2.675983
conirostris  4.349867 2.984200 2.654400 2.513800 2.360167
difficilis   4.224067 2.898917 2.277183 2.011100 1.929983
scandens     4.261222 2.929033 2.621789 2.144700 2.036944
fortis       4.244008 2.894717 2.407025 2.362658 2.221867
fuliginosa   4.132957 2.806514 2.094971 1.941157 1.845379
pallida      4.265425 3.089450 2.430250 2.016350 1.949125
fusca        3.975393 2.936536 2.051843 1.191264 1.401186
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
```

The geospiza object is a list that contains a tree and a data set of measurements on the tip species. We can use the `str()` command to look more closely at geospiza. We will separate the tree and data objects so that we can more easily work with them.

```
> str(geospiza)        ## structure gives you info on what is in the file
```

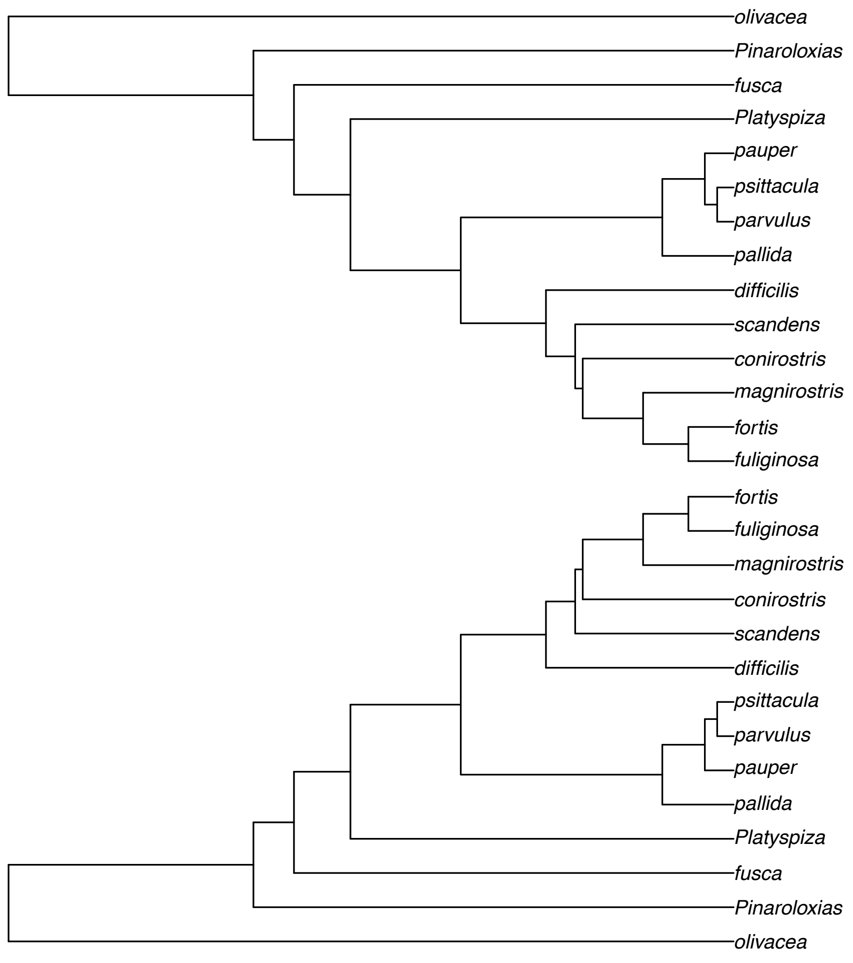
```
List of 2
 $ geospiza.tree:List of 4
  ..$ edge       : num [1:26, 1:2] 15 16 17 18 19 20 21 22 23 24 ...
  ..$ edge.length: num [1:26] 0.2974 0.0492 0.0686 0.134 0.1035 ...
  ..$ Nnode      : int 13
  ..$ tip.label  : chr [1:14] "fuliginosa" "fortis" "magnirostris" "conirostris" ...
  ..- attr(*, "class")= chr "phylo"
 $ geospiza.data:'data.frame':       13 obs. of  5 variables:
  ..$ wingL  : num [1:13] 4.4 4.35 4.22 4.26 4.24 ...
  ..$ tarsusL: num [1:13] 3.04 2.98 2.9 2.93 2.89 ...
  ..$ culmenL: num [1:13] 2.72 2.65 2.28 2.62 2.41 ...
  ..$ beakD  : num [1:13] 2.82 2.51 2.01 2.14 2.36 ...
  ..$ gonysW : num [1:13] 2.68 2.36 1.93 2.04 2.22 ...
```

```
> tree<-geospiza$geospiza.tree ## separate the tree
> data<-geospiza$geospiza.data ## separate the dataframe
```

10.3 Basic Tree Plotting

If we want to visualize the tree we can use the plot command. Often we will want to draw the tree in a consistent way so that it is obvious when topologies are the same or different. The ladderize command can be used to always place the smallest clades on the right or left hand side of a figure.

```
> layout(matrix(1:2, 2, 1))
> plot(tree, main = 'unladderized', cex = 0.75, no.margin = 'T')
> plot(ladderize(tree, right = F), main = 'ladderized left',  cex = 0.75, no.margin = 'T
```



We can save trees as pdf files with the `pdf()` command. This command redirects all graphical output to the pdf file, so make sure to include the `dev.off()` command after your have made your plot.

```
> pdf("geoTree.pdf")
> plot(tree)
> dev.off()
```

```
quartz
     2
```

10.4 Tree Structure

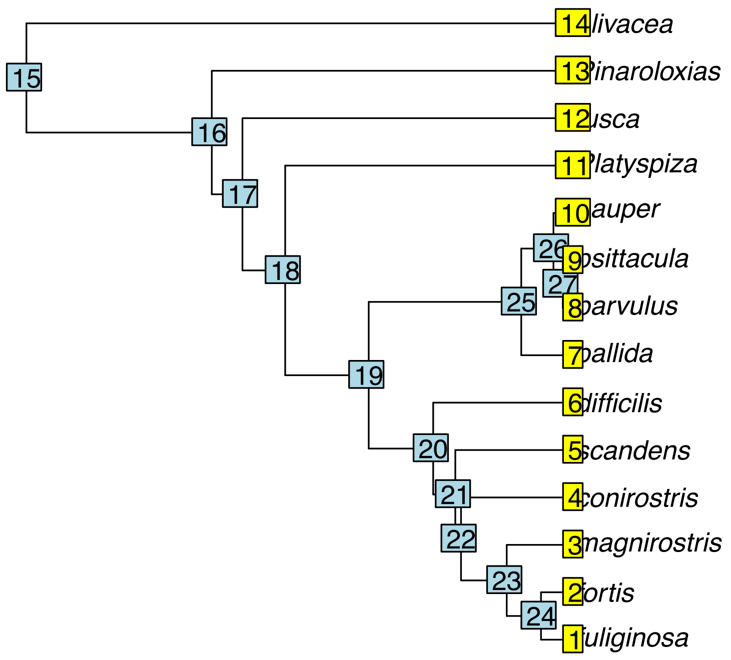
The basic tree structure in R is called an edge matrix. Let's take a look at the edge matrix for the geospiza tree.

```
> tree$edge
```

```
      [,1] [,2]
 [1,]   15   16
 [2,]   16   17
 [3,]   17   18
 [4,]   18   19
 [5,]   19   20
 [6,]   20   21
 [7,]   21   22
 [8,]   22   23
 [9,]   23   24
[10,]   24    1
[11,]   24    2
[12,]   23    3
[13,]   22    4
[14,]   21    5
[15,]   20    6
[16,]   19   25
[17,]   25    7
[18,]   25   26
[19,]   26   27
[20,]   27    8
[21,]   27    9
[22,]   26   10
[23,]   18   11
[24,]   17   12
[25,]   16   13
[26,]   15   14
```

You can see that the edge matrix has two columns. The number of rows equals the number of all of the tips of the tree plus all of the internal nodes. The entries in the first column are parent nodes and the entries in the second column show a daughter node from the corresponding parent. In a bifucating tree, each parent node will appear in column 1 twice and the entries in column two of those rows will be the node numbers of the daughters. We can use the `nodelabels()` and `tiplabels()` commands to see this.

```
> plot(tree)
> nodelabels()
> tiplabels()
```



Check the tree figure and verify that the parent nodes (in blue) appear in column 1 of the edge matrix twice and that the column 2 entries for those rows point to their respective daughter nodes (in yellow boxes). To see the tip names of a tree we ask for the tip.label attribute of the tree.

```
> str(tree)
```

```
List of 4
 $ edge       : num [1:26, 1:2] 15 16 17 18 19 20 21 22 23 24 ...
 $ edge.length: num [1:26] 0.2974 0.0492 0.0686 0.134 0.1035 ...
 $ Nnode      : int 13
 $ tip.label  : chr [1:14] "fuliginosa" "fortis" "magnirostris" "conirostris" ...
 - attr(*, "class")= chr "phylo"
```

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

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

We can also get the branch lengths for the tree.

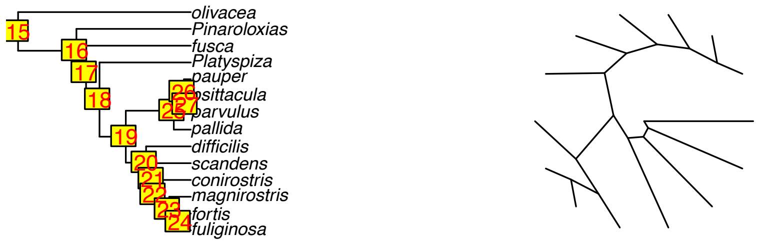
```
> tree$edge.length
```

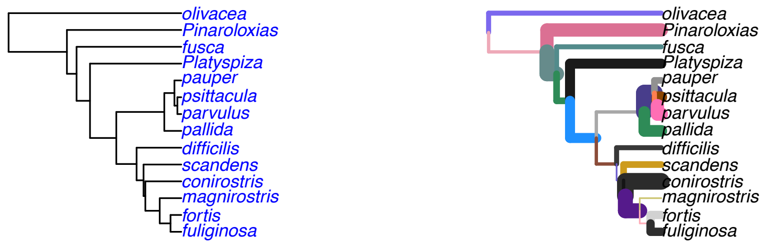
```
 [1] 0.29744 0.04924 0.06859 0.13404 0.10346 0.03550 0.00917 0.07333 0.05500
[10] 0.05500 0.05500 0.11000 0.18333 0.19250 0.22800 0.24479 0.08667 0.05167
[19] 0.01500 0.02000 0.02000 0.03500 0.46550 0.53409 0.58333 0.88077
```

10.5 More Tree Plotting Tricks

Now that you are familiar with the basic structure of trees in R, we can explore some of the tree plotting options. It is easy to specify colors and sizes of branches and tip labels using plot options. You can use the `help(phylo)` to see a list of more options.

```
> layout(matrix(1:4, 2, 2))
> plot(tree)
> nodelabels(col="red",bg="yellow")
> plot(tree,tip.color="blue")
> plot(tree, type = 'radial', show.tip.label = 'F')
> plot(tree,
+      edge.color = sample(colors(), length(tree$edge)/2),
+      edge.width = sample(1:10, length(tree$edge)/2, replace = TRUE))
```





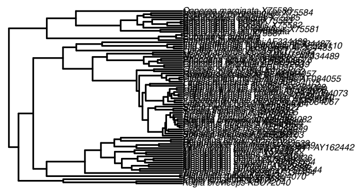
10.6 Tree Input and Output

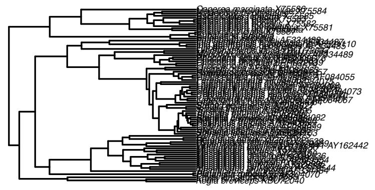
R can read both newick and nexus-formatted tree trees using the APE package.

10.6.1 Reading Trees

To read in a trees use the `read.tree()` command for Newick formatted tree and `read.nexus()` for NEXUS tree files.

```
> whale.phy<-read.tree('Data/whale.phy')
> whale.nex<-read.nexus('Data/whale.nex')
> layout(matrix(1:2, 2, 1))
> plot(whale.phy, cex = 0.33)
> plot(whale.nex, cex = 0.33)
```





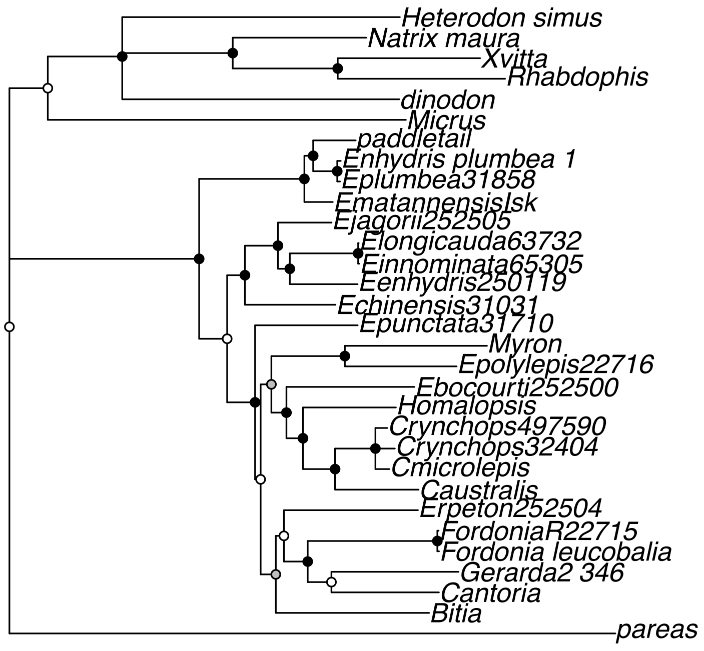
R can also read in multiple trees at the same time. This is useful when you are working with output from Bayesian analysis programs like BEAST or MrBayes

```
> lots.of.trees<-read.tree('Data/multiple_trees.phy')
```

10.6.2 Plotting Support Symbols on Trees

This example shows how you can take the consensus file from a MrBayes analysis and visualize clade posterior probabilities in R (Thanks to Rich Glor at dechronization for code).

```
> snakecon<-read.nexus('Data/consensus.tre')
> snakecon[[1]]->snakecon
> p <- character(length(snakecon$node.label))
> #The following three lines define your labeling scheme.
> p[snakecon$node.label >= 0.95] <- "black"
> p[snakecon$node.label < 0.95 & snakecon$node.label >= 0.75] <- "gray"
> p[snakecon$node.label < 0.75] <- "white"
> #Almost done, you're ready to plot your tree
> layout(matrix(1))
> plot(snakecon)
> #Now label your tree:'pch' tells R to use filled circles, 'cex' defines the size of th
> nodelabels(pch=21, cex = .75, bg = p)
```



10.6.3 Writing Trees

Similarly, the `write.tree()` command will write a Newick representation of your tree and `write.nexus()` produces a nexus version of it.

```
> write.tree( whale.phy, file ='new.phylip.tre')
> write.nexus( whale.phy, file ='new.nexus.tre')
```

10.6.4 Manipulating Tree Labels and Branch Lengths

The edge.length attribute of the tree structure in R defines the branch lengths described by the rows of the edge matrix we saw earlier. It is easy to modify the edge.length values. Let's look again at the geospiza example.

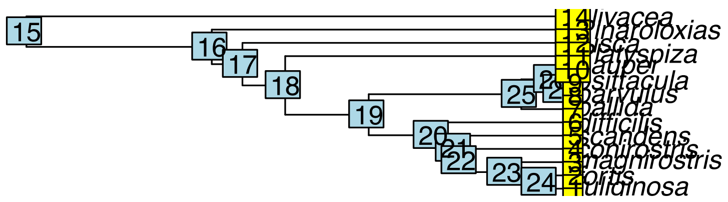
```
> layout(matrix(1:2, 2, 1))
> data(geospiza)
> tree<-geospiza$geospiza.tree
> tree$edge
```

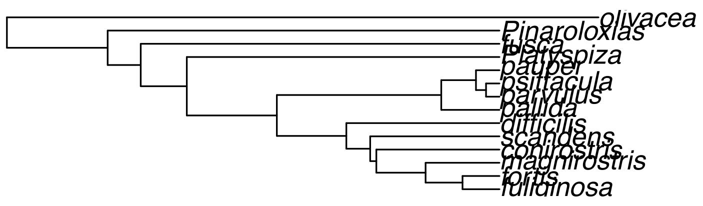
```
      [,1] [,2]
 [1,]   15   16
 [2,]   16   17
 [3,]   17   18
 [4,]   18   19
 [5,]   19   20
 [6,]   20   21
 [7,]   21   22
 [8,]   22   23
 [9,]   23   24
[10,]   24    1
[11,]   24    2
[12,]   23    3
[13,]   22    4
[14,]   21    5
[15,]   20    6
[16,]   19   25
[17,]   25    7
[18,]   25   26
[19,]   26   27
[20,]   27    8
[21,]   27    9
[22,]   26   10
[23,]   18   11
[24,]   17   12
[25,]   16   13
[26,]   15   14
```

```
> tree$edge.length
```

```
 [1] 0.29744 0.04924 0.06859 0.13404 0.10346 0.03550 0.00917 0.07333 0.05500
[10] 0.05500 0.05500 0.11000 0.18333 0.19250 0.22800 0.24479 0.08667 0.05167
[19] 0.01500 0.02000 0.02000 0.03500 0.46550 0.53409 0.58333 0.88077
```

```
> plot(tree)
> nodelabels()
> tiplabels()
> tree$edge.length[1]<-0.15
> tree$edge.width[1]<-2
> plot(tree)
```





We can set the length of a particular branch in the edge matrix (as we did here with edge.length[1]<-0.15. We can also change all of the branch lengths at once.

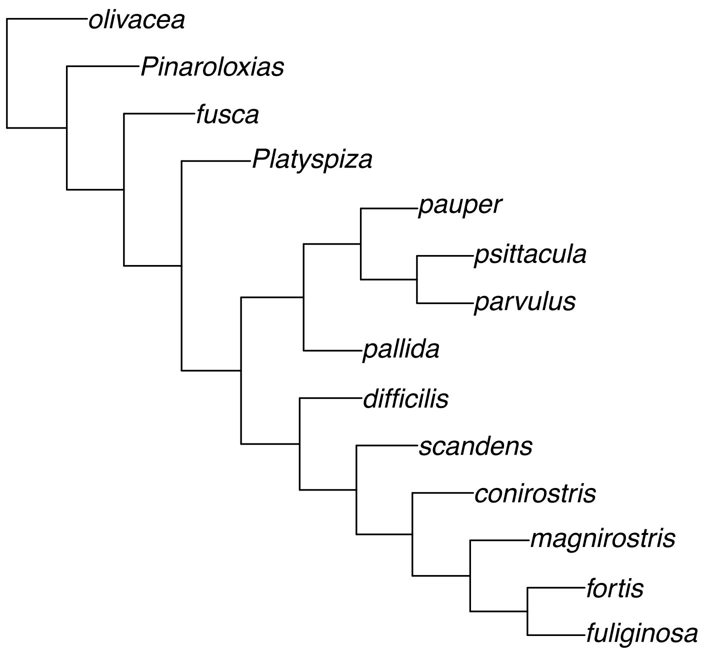
```
> layout(1)
> tree$edge.length
```

```
 [1] 0.15000 0.04924 0.06859 0.13404 0.10346 0.03550 0.00917 0.07333 0.05500
[10] 0.05500 0.05500 0.11000 0.18333 0.19250 0.22800 0.24479 0.08667 0.05167
[19] 0.01500 0.02000 0.02000 0.03500 0.46550 0.53409 0.58333 0.88077
```

```
> tree$edge.length<-tree$edge.length + 1
> tree$edge.length

 [1] 1.15000 1.04924 1.06859 1.13404 1.10346 1.03550 1.00917 1.07333 1.05500
[10] 1.05500 1.05500 1.11000 1.18333 1.19250 1.22800 1.24479 1.08667 1.05167
[19] 1.01500 1.02000 1.02000 1.03500 1.46550 1.53409 1.58333 1.88077

> plot(tree)
```



10.6.5 Miscellaneous Tree Commands

Once you have a tree in R, it is easy to get information about it. These commands show some of what is possible–you can query whether a tree is ultrametric, find out how many taxa are in it, or look at the distribution of branch lengths.

```
> data(geospiza)
> tree<-geospiza$geospiza.tree
> class(tree)
```

```
[1] "phylo"
```

```
> mean(tree$edge.length)
```

```
[1] 0.1764008
```

```
> hist(tree$edge.length)
> is.ultrametric(tree)
```

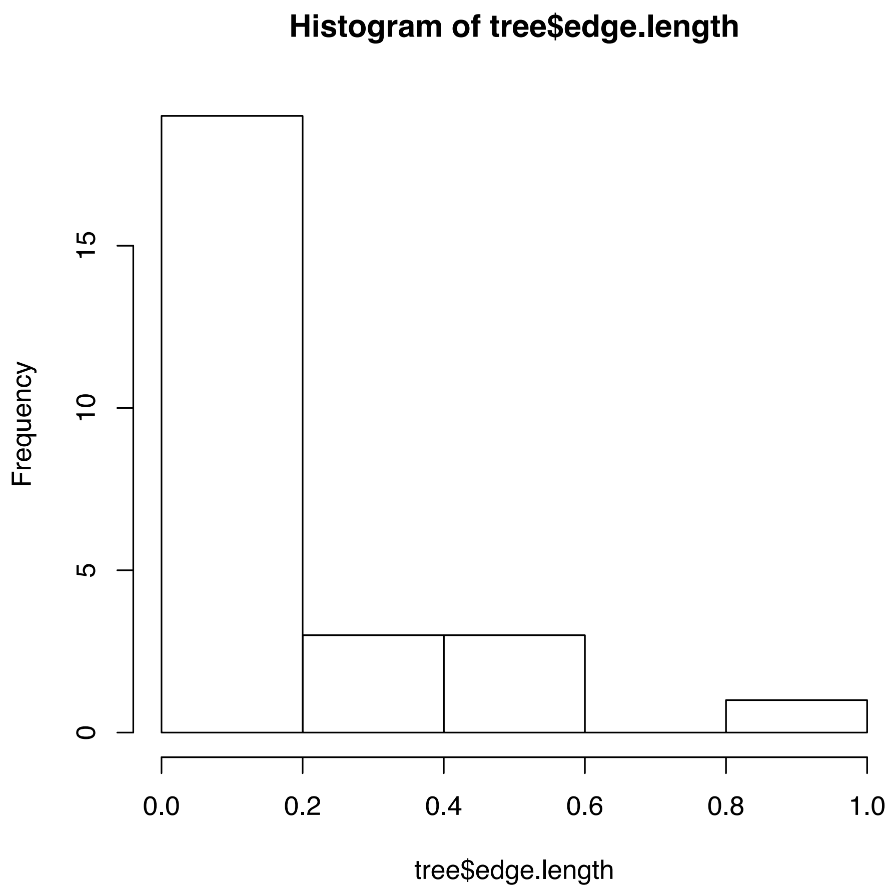
```
[1] TRUE
```

```
> tree$Nnode
```

```
[1] 13
```

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

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



Practice

1. Read one of your trees into R and save it as a pdf.
2. Set all the branch lengths in the geospiza tree to 1.
3. Make the tip labels of a clade on your tree red. Hint: use the `tip.col` option within plot. This command will label 5 taxa red and 18 taxa blue: `plot(your.tree, tip.col = c(rep("red", 5), rep("blue", 18)))`.

Chapter 11

Ancestral State Reconstructions by Graham Slater

Ancestral state reconstructions allow you to reconstruct the history of a trait along a phylogeny. This is incredibly useful when trying to predict the ancestral state for a trait or inferring polarity in character evolution. Plus, the make for some really cool images when preparing manuscripts. Older methods based on parsimony resulted in some quite ambiguous reconstructions. Fortunately we now have some neat likelihood methods that can take additional phylogenetic information, such as branch lengths, into account. These methods allow us to additionally infer rates of character evolution and test different models of state changes. Let's take a look at how this works in R. We'll be using the ace function in the ape package.

First we need a sample data set to work with. We'll use the Geospiza data from the geiger package:

```
> require(geiger)
> data(geospiza)
> tree<-geospiza$geospiza.tree
> data<-geospiza$geospiza.data
> nc<-name.check(tree,data)
> tree<-drop.tip(tree,nc[[1]])
```

the last two steps there checked that the names in the tree and data matched and dropped any unrepresented taxa from the tree now we've matched up the names in the tree and data, we're ready to go. the ace function requires the data to be in a vector format, with entries either in the same order as they are in the tree, or else with names assigned to them (just like with PICs). We'll create a vector for beak depth for this example

```
> beakD<-data[,4]
> names(beakD)<-rownames(data)
```

Now we're ready to try some ancestral state reconstructions. We'll begin by trying the default options in the ace function. type...

```
> ace(beakD, tree)


    Ancestral Character Estimation

Call: ace(x = beakD, phy = tree)

    Log-likelihood: -1.180158

$ace
      14       15       16       17       18       19       20       21
1.826695 1.850263 1.967723 2.087918 2.197687 2.264403 2.287339 2.380169
      22       23       24       25
2.227995 2.047707 2.057914 2.054239

$sigma2
[1] 0.40589799 0.08289514

$CI95
       [,1]     [,2]
14 1.442009 2.211382
15 1.500153 2.200372
16 1.642478 2.292967
17 1.809334 2.366502
18 1.972721 2.422653
19 2.063233 2.465574
20 2.087948 2.486731
21 2.196500 2.563837
22 2.081668 2.374321
23 1.866770 2.228644
24 1.945915 2.169912
25 1.965992 2.142487
```

You'll see that you got a list back with a series of elements:

loglik this is the maximum log-likelohood for the reconstruction - you'll only get this if you use the ML method

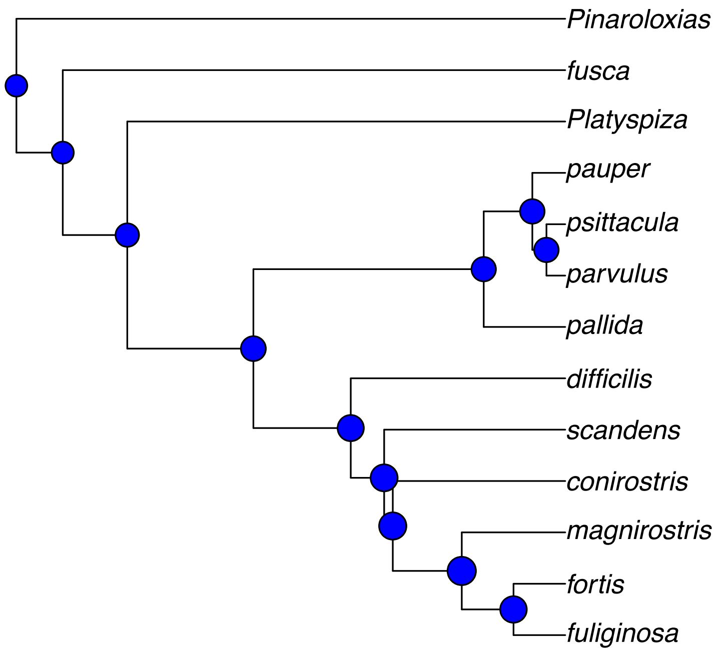
ace this is a vector of estimated ancestral states for the nodes in your tree

sigma2 this is the maximum likelihood estimate of sigma2 - the brownian rate

CI95 95% confidence limits for the node values. Notice that the CIs are quite wide, especially for the basal nodes. This is expected - we have low power to estimate ancestral states towards the base of the tree because, under BM, a wide range of values are possible.

We can also save this output and visualize it, like this:

```
> asr<-ace(beakD,tree)
> plot(tree)
> nodelabels(pch=21,cex=asr$ace,bg="blue")
```



This plots the ancestral state reconstructions on the nodes using circles of corresponding size. If the circles are too small or the range is too big, you can adjust them by scaling the size.

You can also add circles at the tips to show the range of observed values - you might want to replot the tree and shift the tip labels a bit first using offset.

```
> plot(tree,label.offset=0.025)
> tiplabels(pch=21,cex=beakD,bg="blue")
> nodelabels(pch=21,cex=asr$ace,bg="blue")
```

Let's take a look at the options in ace

```
> ?ace
```

Let's look at the arguments that ace takes.

We see that we have two options for "type" - continuous or discrete. We just used the default option, continuous traits. We'll look at discrete next. We also have options for method: "ML" (maximum likelihood, and the default option), "pic" or "GLS". the last two do exactly what they say they do: independent contrasts or generalized least squares. We actually did ancsestral state reconstructons using these methods earlier when doing contrasts and GLS, although we didn't save or do anything with the reconstructions. We'll focus on ML here. ML is the default option for continuous traits, and the only option for discrete ones. I should note here that the algorithm that ace uses optimizes all node states simulataneously. This works well for small trees like the one we're using here, but can make errors with larger datasets. You might like to confirm that your inferred states agree with those from another ancestral state function, getAncStates, in geiger. The final thing I want you to note here is the model argument. For continuous traits, we use BM. For discrete traits, we default to something called "ER", which stands for equal rates in any direction of change (an MK1 model if you've tried this in MESQUITE). There are other options and we'll see those now.

Let's get a discrete dataset together for the Geospiza tree. Here is diet coded from Schluter et als 1997 paper in Evolution. Diet is coded as 0 = granivores, 1 = insectivores and 2=folivores. The taxa in this dataset are slightly different from those in the continuous dataset so we'll need to make a new tree from the original.

```
> diet<-c(0,0,0,0,0,0,1,1,1,2,1)
> names(diet)<-c("fuliginosa","fortis","magnirostris","difficilis","conirostris","scan
> tree2<-geospiza$geospiza.tree
> nc<-name.check(tree,diet)
> tree2<-drop.tip(tree2,nc[[1]])
```

Now we'll try some discrete state reconstructions. Remember we need to specify that we're doing discrete characters. Let's try the default options first

```
> asrD<-ace(diet,tree2,type="discrete")
> asrD
```

```
    Ancestral Character Estimation

Call: ace(x = diet, phy = tree2, type = "discrete")

    Log-likelihood: -5.794289

Rate index matrix:
  0 1 2
0 . 1 1
1 1 . 1
2 1 1 .

Parameter estimates:
 rate index estimate std-err
          1   0.6073  0.3108

Scaled likelihoods at the root (type '...$lik.anc' to get them for all nodes):
        0         1         2
0.3300753 0.4159415 0.2539831
```

Note that we saved the ace output here as asrD, so we have to type that in to get the results displayed in the console. you'll notice that the output of the discrete state reconstruction is different to that for continuous traits. You still have a log-likelihood but the other output looks a little different. Instead of sigma2, you have something called "rates". We used the default, equal rates (ER) model so we get one maximum likelihood estimate of the rate. We also get a standard error (se) associated with that rate esimate. index.matrix is a matrix showing which rate in rates corresponds to which transitions. This will make more sense in a minute. We also get a list of ancestral state likelihoods (lik.anc). Because we're dealing with a discrete trait, we get likelihoods associated with each state, rather than one esimated value. Note that each row (a node) sums to one. This makes it convenient to represent these values visually.

```
> plot(tree2,label.offset=0.02)
> co <- c("blue", "yellow","red")
> tiplabels(pch = 21, bg = co[as.numeric(diet+1)], cex = 2) ## note we had to add one
> nodelabels(pie =asrD$lik.anc, piecol=co , cex =1)
```

You'll note here again that we can estimate the states of higher nodes quite well but uncertainty increases as we move toward the root We made an assumption above that the rate of change from one state to another is the same, regardless of direction. We can test that assumption. Remember how we used the default ER setting? Let's try changing that and using an asymmetric model, where the state of change in one direction differs from that going the other way (i.e. all rates are different).

```
> asrD2<-ace(diet,tree2,type="discrete",model="ARD")
> asrD2


    Ancestral Character Estimation

Call: ace(x = diet, phy = tree2, type = "discrete", model = "ARD")

    Log-likelihood: -5.505021

Rate index matrix:
  0 1 2
0 . 3 5
1 1 . 6
2 2 4 .

Parameter estimates:
 rate index estimate std-err
          1   0.5550  0.9221
          2   0.7390  1.9748
          3   0.6737  0.8782
          4   1.8559  3.0988
          5   0.3124  0.7419
          6   0.7447  1.2270

Scaled likelihoods at the root (type '...$lik.anc' to get them for all nodes):
        0         1         2
0.2645695 0.3666062 0.3688243
```

Checking the results from this, you'll see there are 6 different rates in asrD2$rates. These correspond to the entries in index.matrix. rate 1 is the transition rate from state 2 to state 1,2 is the rate from 3 to 1 and so on. You can try plotting the results as pie charts using the code above. You will see that the differences are marginal and mostly affect nodes towards the base of the tree. Lets also try another situation where transitions between the same pair of states have the same rate but transitions between other states are different.

```
> asrD3<-ace(diet,tree2,type="discrete",model="SYM")
> asrD3


    Ancestral Character Estimation

Call: ace(x = diet, phy = tree2, type = "discrete", model = "SYM")
```

```
    Log-likelihood: -5.723915

Rate index matrix:
  0 1 2
0 . 1 2
1 1 . 3
2 2 3 .

Parameter estimates:
 rate index estimate std-err
          1   0.7169  0.6123
          2   0.3299  0.5590
          3   0.9428  1.2283

Scaled likelihoods at the root (type '...$lik.anc' to get them for all nodes):
        0         1         2
0.3132024 0.4089024 0.2778952
```

Looking at the rates and index matrix, you'll see that we now have 3 rates - 1 for a transition between each pair of states. We've just used the 3 built-in transition matrices, but you can invent your own - for example if you want to allow rates to all be the same for transitions in one direction but different transitions in the other. You do this by building a model matrix. For example, model=matrix(c(0,1,1,0),2). In this case we build a 2x2 matrix where there is just one rate between the two states. Make sure that if you use a custom transition matrix, that the number of rows and columns in your matrix corresponds to the number of states your character can take.

Now we've tried the three built in matrices, how do we chose among the possible models of trait evolution? Which one do your data fit best? We could use the log-likelihood values - Bigger values imply better fits. You could get a P-value by computing likelhood ratio tests with degrees of freedom = difference in number of parameters. Alternatively you could directly extract AIC scores for comparison. Let's look at the log likelihood scores first.

```
> list("ER"=asrD$loglik,"ARD"=asrD2$loglik,"SYM"=asrD3$loglik)


$ER
[1] -5.794289

$ARD
[1] -5.505021
```

```
$SYM
[1] -5.723915
```

Not that different are they? Based on these, it might be tempting to suggest that the "All Rates Different" model is best because its log-likelihood is the highest. But remember, there are different numbers of parameters in these models and you want to penalize for this. Likelihood ratio tests do this using degrees of freedom. Using a model-inference approach, you can compute the AIC score like this:

```
> ER.AIC<-AIC(ace(diet,tree2,type="discrete"))
> ARD.AIC<-AIC(ace(diet,tree2,type="discrete",model="ARD"))
> SYM.AIC<-AIC(ace(diet,tree2,type="discrete",model="SYM"))
> list("ER"=ER.AIC,"ARD"=ARD.AIC,"SYM"=SYM.AIC)
```

```
$ER
[1] 13.58858

$ARD
[1] 23.01004

$SYM
[1] 17.44783
```

When using AIC, the model with the lowest AIC score is the one that the data fit best. Here the "Equal Rates" model is the best fit. The difference between it and the symmetric model is 4, which is the normal threshold taken as strong support for one model over another. The "All Rates Different" model is a poor fit to the data, probably because of the large number of parameters. Because we have only 11 tips in the tree here, we should probably have used a small-sample corrected AIC score. For bigger trees, AIC will be fine.

Chapter 12

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

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

```
> tree$edge
```

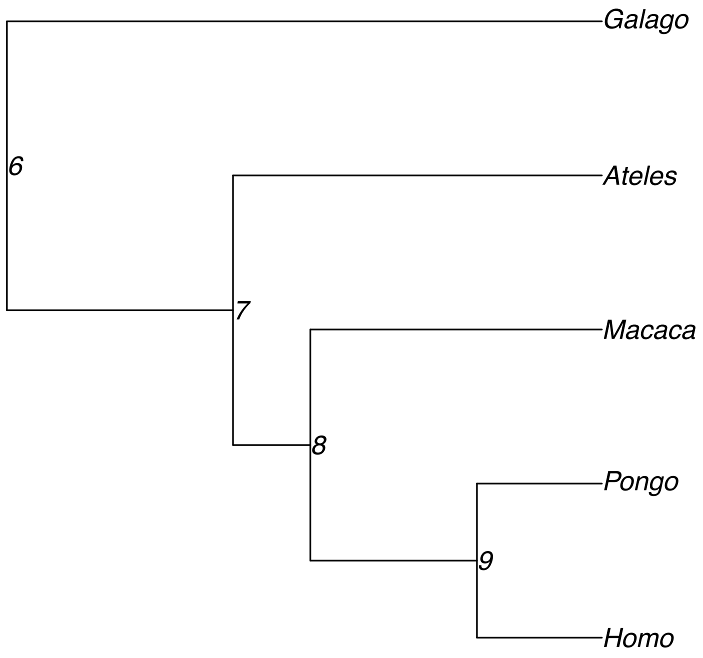
```
     [,1] [,2]
[1,]    6    7
[2,]    7    8
[3,]    8    9
[4,]    9    1
[5,]    9    2
[6,]    8    3
[7,]    7    4
[8,]    6    5
```

```
> 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"
```

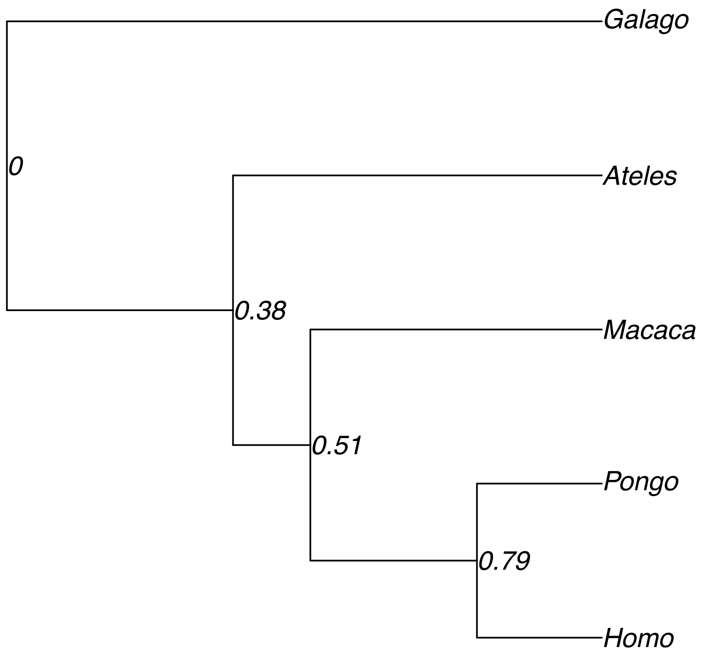
Since we need to calculate a similarity matrix based on time in shared evolutionary history, it would actually be much more convenient to have for each node, the time from the root to the node. Let's call this "times". For a small and simple phylogeny, we can do this by "hand". Looking at the dataframe 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:

```
> 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


> XY <- data.frame(Y, X)
> 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
```

and with the regression from PIC:

```
> pic.X <- pic(X, tree.primates)
> pic.Y <- pic(Y, tree.primates)
> 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 13

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.

13.1 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.

13.2 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.

13.3 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`".

13.4 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
```

13.5 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.

13.6 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
```

13.7 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>>=
@
```

13.8 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 14

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.

14.1 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 14.1: Attributes of Hero characters.

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

Table 14.2: 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.

14.2 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.

14.3 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              [[.data.frame              [[<-.data.frame
 [4] [<-.data.frame            $<-.data.frame             aggregate.data.frame
 [7] anyDuplicated.data.frame as.data.frame.data.frame as.list.data.frame
[10] as.matrix.data.frame      by.data.frame              cbind.data.frame
[13] dim.data.frame            dimnames.data.frame        dimnames<-.data.frame
[16] duplicated.data.frame     edit.data.frame*           format.data.frame
[19] formula.data.frame*       head.data.frame*           is.na.data.frame
[22] Math.data.frame           mean.data.frame            merge.data.frame
[25] na.exclude.data.frame*    na.omit.data.frame*        Ops.data.frame
[28] plot.data.frame*          print.data.frame           prompt.data.frame*
[31] rbind.data.frame          row.names.data.frame       row.names<-.data.frame
[34] rowsum.data.frame         split.data.frame           split<-.data.frame
[37] stack.data.frame*         str.data.frame*            subset.data.frame
[40] summary.data.frame        Summary.data.frame         t.data.frame
[43] tail.data.frame*          transform.data.frame       unique.data.frame
[46] unstack.data.frame*       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] +.phylo                    all.equal.phylo            as.hclust.phylo
 [4] as.matching.phylo          coalescent.intervals.phylo cophenetic.phylo
 [7] identify.phylo             makeLabel.phylo            plot.phylo
[10] print.phylo                reorder.phylo              skyline.phylo
[13] summary.phylo              vcv.phylo
```

14.3.1 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"
```

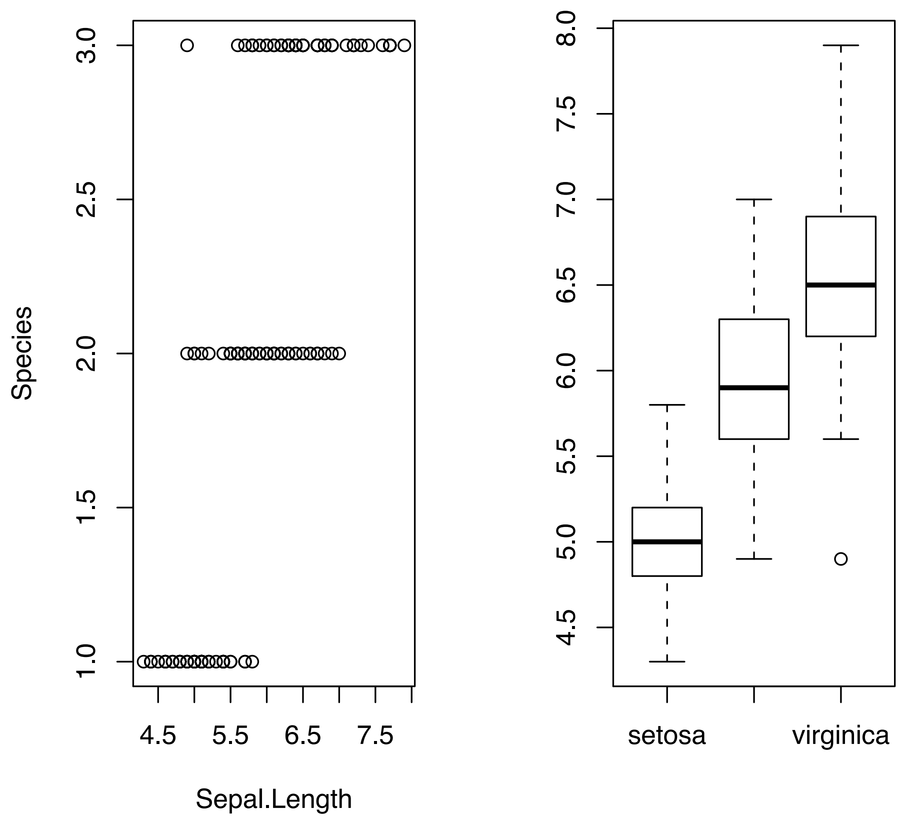
There is no validation or checking that the objects have appropriate contents. R just tries to do what it can with the crazy object. Worse things will happen if you try to plot this fake tree.

14.3.2 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.

14.4 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.

14.4.1 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 15

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.

15.1 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.

15.2 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.

15.3 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.

15.4 Tree and Data Formats

15.4.1 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
```

15.4.2 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:

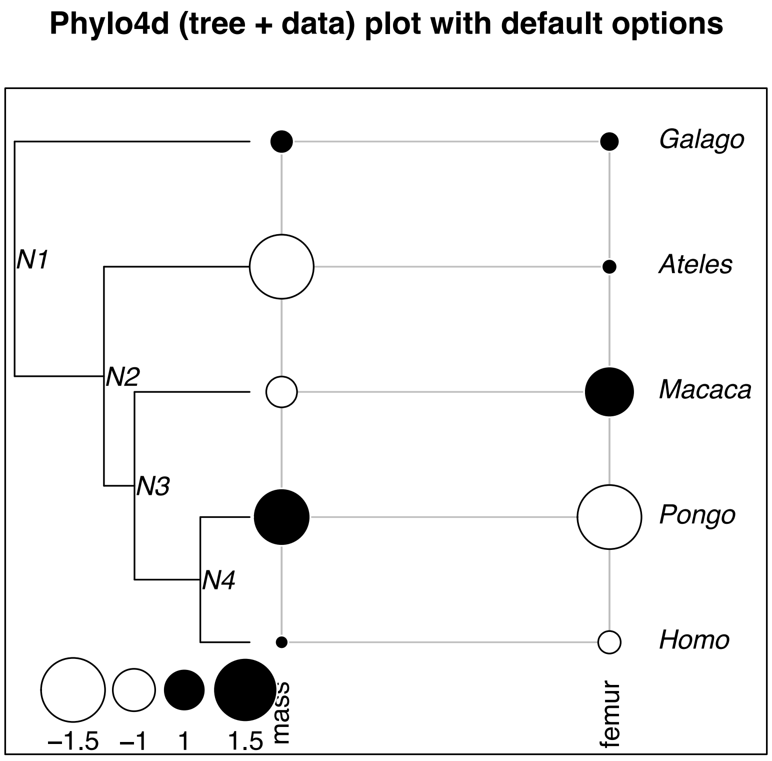
1. Make a dataframe of morphological data to match the primate tree
2. Name the rows with the tip.label (species names)
3. Use the phylo4d constructor to bind the tree and data together

```
> 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
```

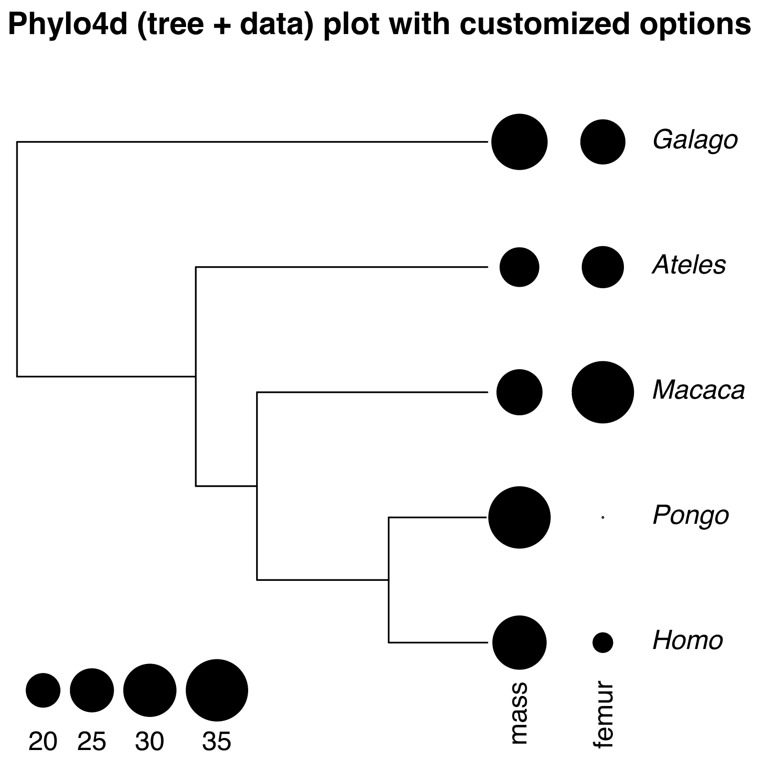
`phylo4d` objects are plotted in the same way as `phylo4`, but we can also add a bubble plot to indicate quantitative data.

```
> 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
```

15.5 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
```

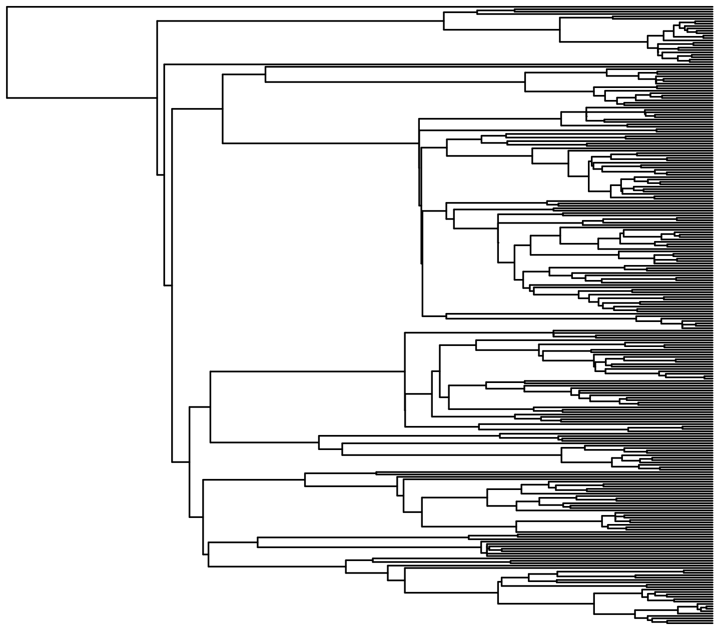
However, if you really want to access an internal element directly, the `$` operator which works for data frames and lists doesn't generally work for S4 objects (it actually works for `phylo4` objects because the developers wrote translation functions for them). Instead, internal elements are accessed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

15.6 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.

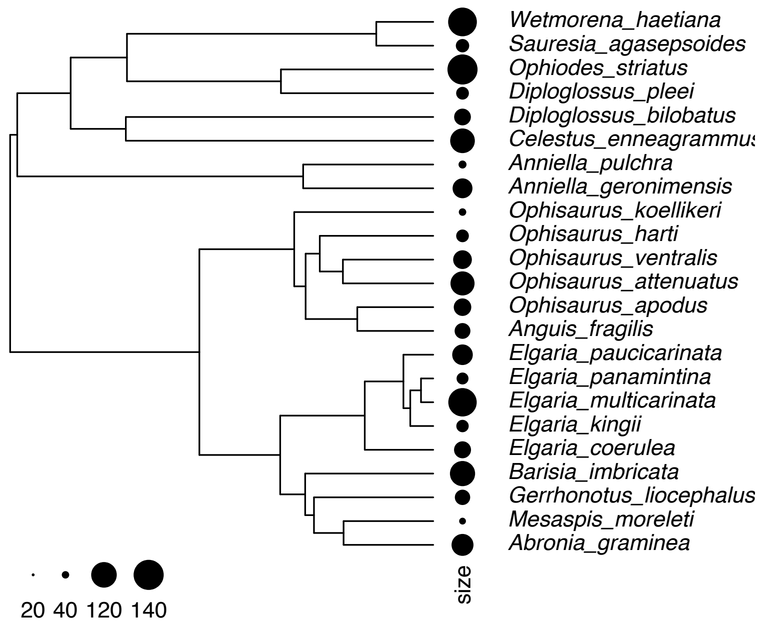
For this example, let's use a larger phylogeny of squamates (lizards). We plot without tip labels because there are just too many taxa.

```
> 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).

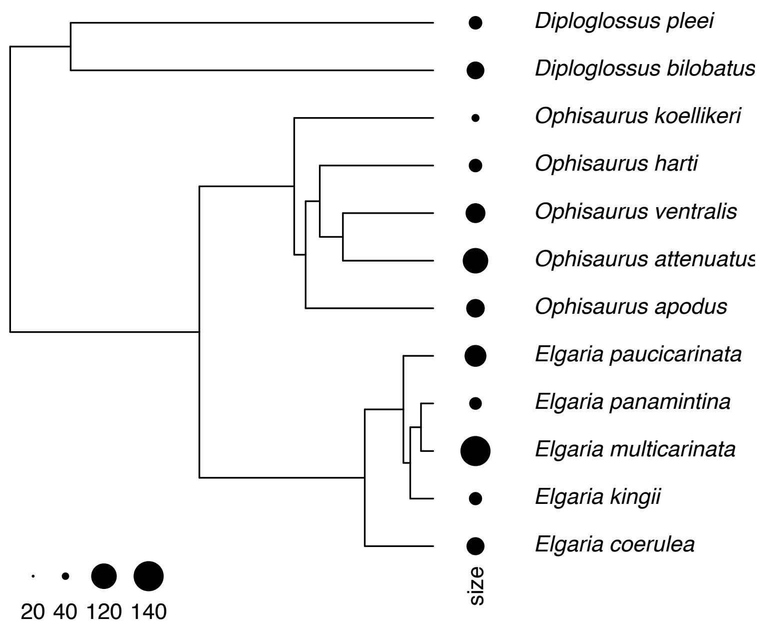
```
> include <- c( grep("Ophisaurus", labels(smalltree4), value=T),
+                 grep("Diploglossus", labels(smalltree4), value=T),
+                 grep("Elgaria", labels(smalltree4), value=T))
> include
```

```
 [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.

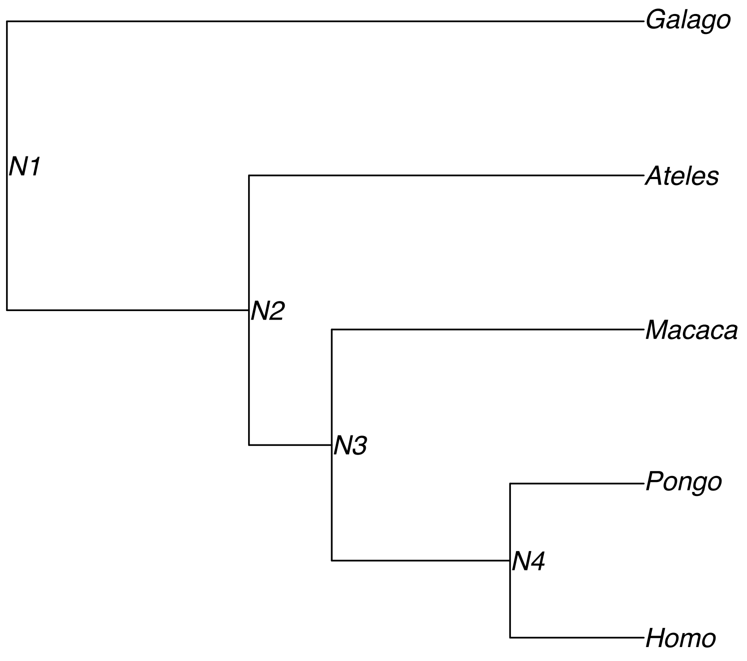
```
> tree.ape <- as(smalltree4d, "phylo")
> tree.ape.dat <- tdata(smalltree4d)
```

15.7 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)
```

15.8 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)

15.8.1 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
```

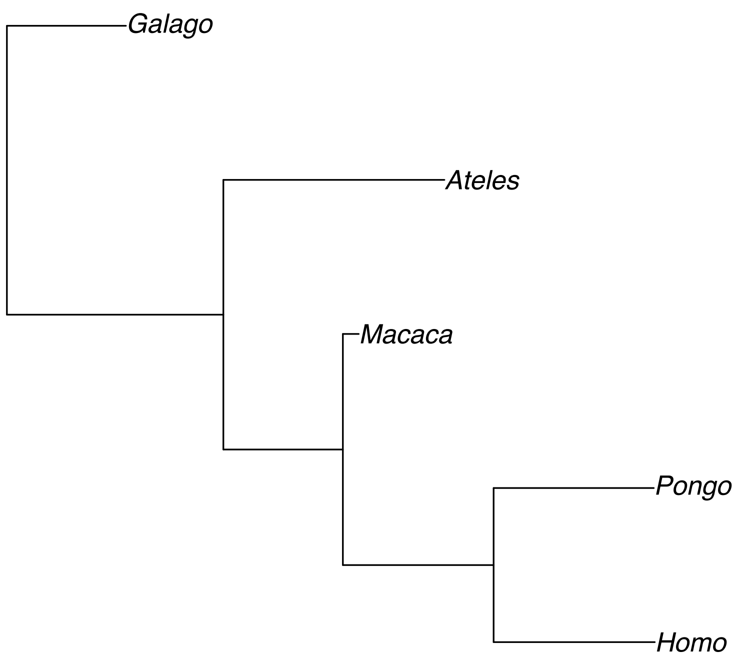
```
[1] 0.6463380 0.3569145 0.4500820 0.4814310 0.4783944 0.0474164
[7] 0.6601589 0.3555261
```

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)))
```

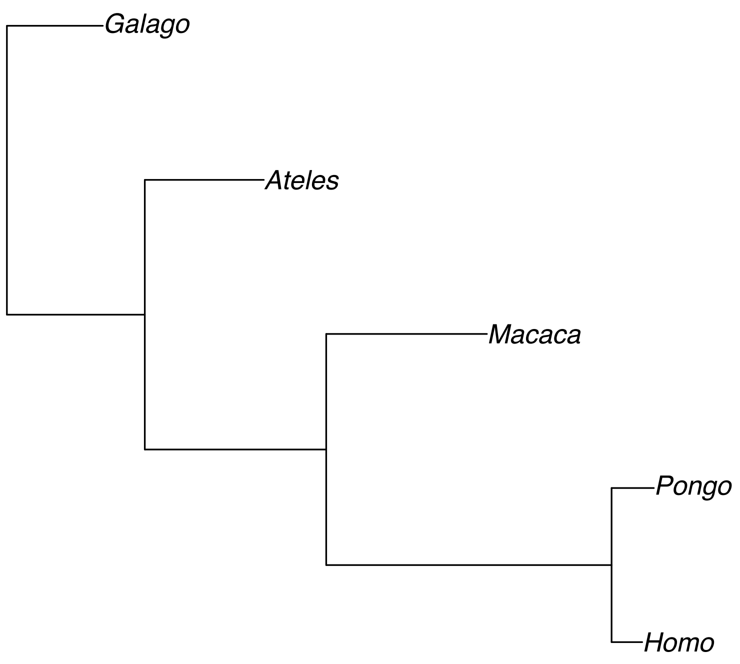
Now we need to make a new tree with the phylo4 constructor. For clarity in this example, lets first save the information from `tree4` into separate variables:

```
> 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):

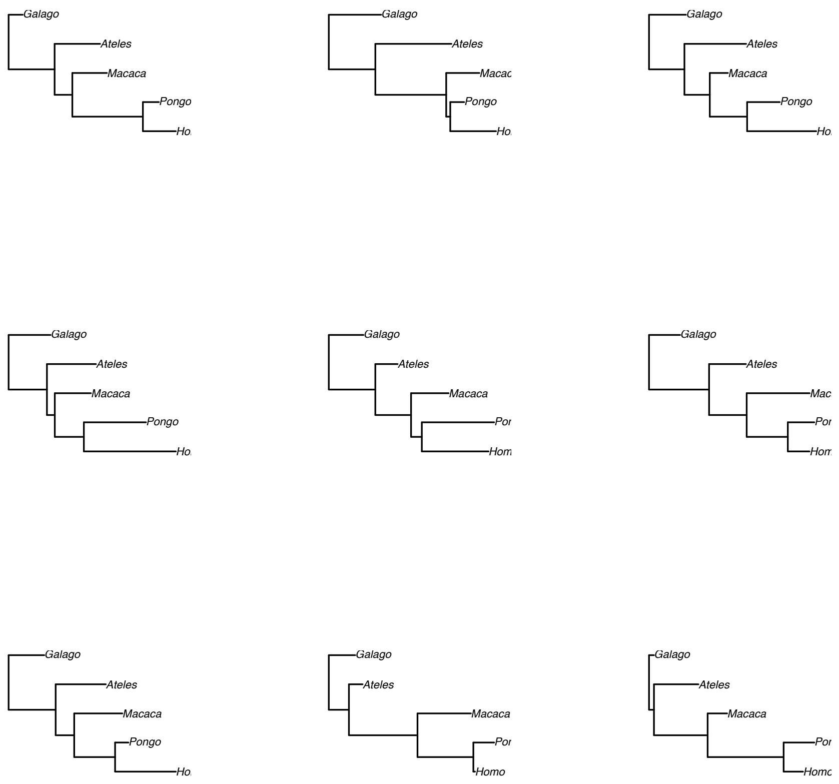
```
> simtrees <- vector(mode='list')
> for (i in 1:9) {
+   bl <- gen.bl(tree4)
+   simtrees[[i]] <- change.bl(tree4, bl)
+ }
```

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)
```



15.8.2 Branch lengths drawn from normal distributions with separate means

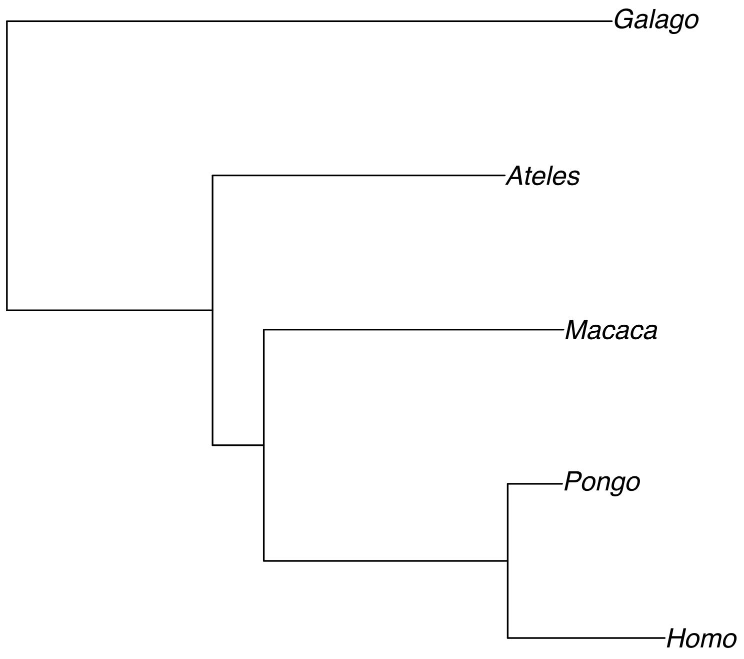
The key difference between these two options is the generation of the branch lengths. So all we need to do is change the `gen.bl` function. Now instead of eight draws from the same distribution, for each tree, we want a single draw from eight distributions (one for each branch). We will assume that they 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)
+ }
```

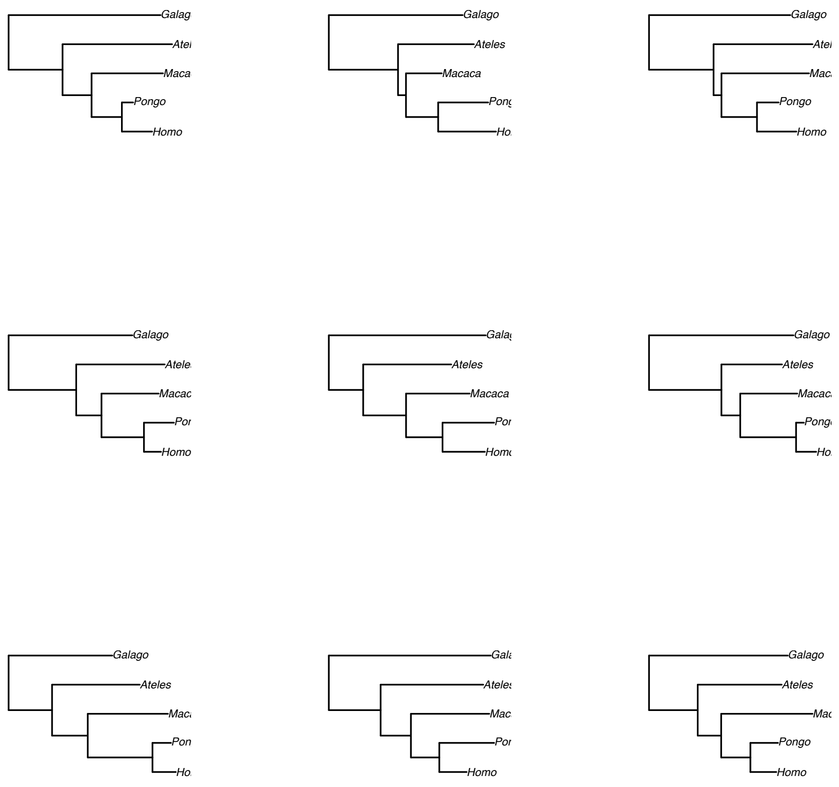
We can reuse the function `change.bl` that we wrote previously, substitute our new branch length generating function and plot our new tree:

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



Let's redo our nine trees:

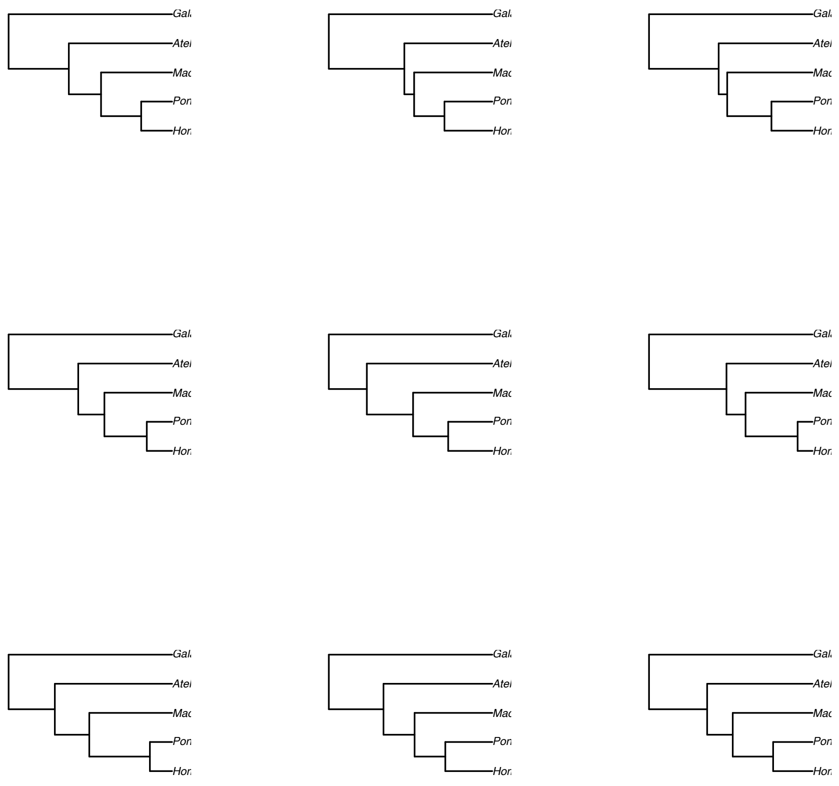
```
>                  # generate the list of branch lengths
> bls <- lapply(1:9, function(x) gen.bl(edgeLength(tree4), bl.stan.dev))
>                  # reuse our lapply code
> simtrees3 <- lapply(bls, function(x) {
+       return( phylo4(edge=edges(tree4), edge.length=x, tip.label=labels(tree4)))
+       })
> 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 16

Stochastic Simulations

Let's make some graphical animations to illustrate the BM and OU model.

16.1 Brownian motion model

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

$$dX(t) = \sigma\, dB(t). \tag{16.1}$$

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.

16.2 Exercises

1. Go back to `bm.sim.slow` and modify it to an OU. Recall the OU equation:

$$dX(t) = \alpha\,(\theta - X(t))\,dt + \sigma\,dB(t). \tag{16.2}$$

   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.

16.3 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.

16.4 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 17

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

17.1 The OU Model for Comparative Analysis

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

$$dX(t) = \sigma \, dB(t). \tag{17.1}$$

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:

$$dX(t) = \alpha \left(\theta - X(t)\right) dt + \sigma \, dB(t). \tag{17.2}$$

Eq. 17.2 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 (17.2) collapses to the familiar BM model of pure drift,

17.2 Introduction to Likelihood

17.3 `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.

17.3.1 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
```

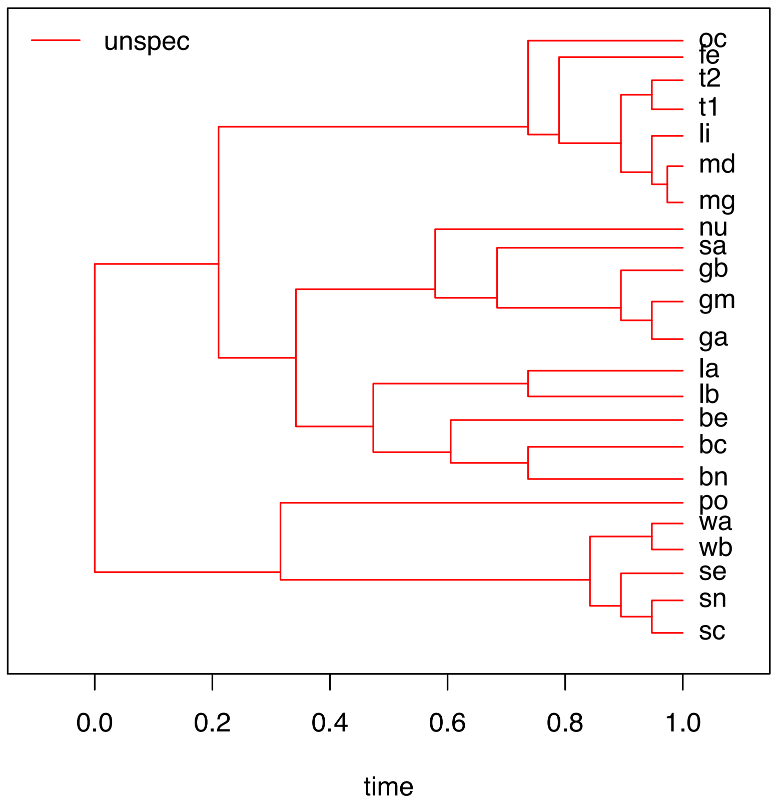
NOTE: a very important detail about `ouch` is that it matches trees with data and regimes using the node labels stored in the rownames of the objects you pass to the `ouch` functions. So it is important to make sure that your dataframes and vectors are appropriately named. The dataframe bimac already 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)
```



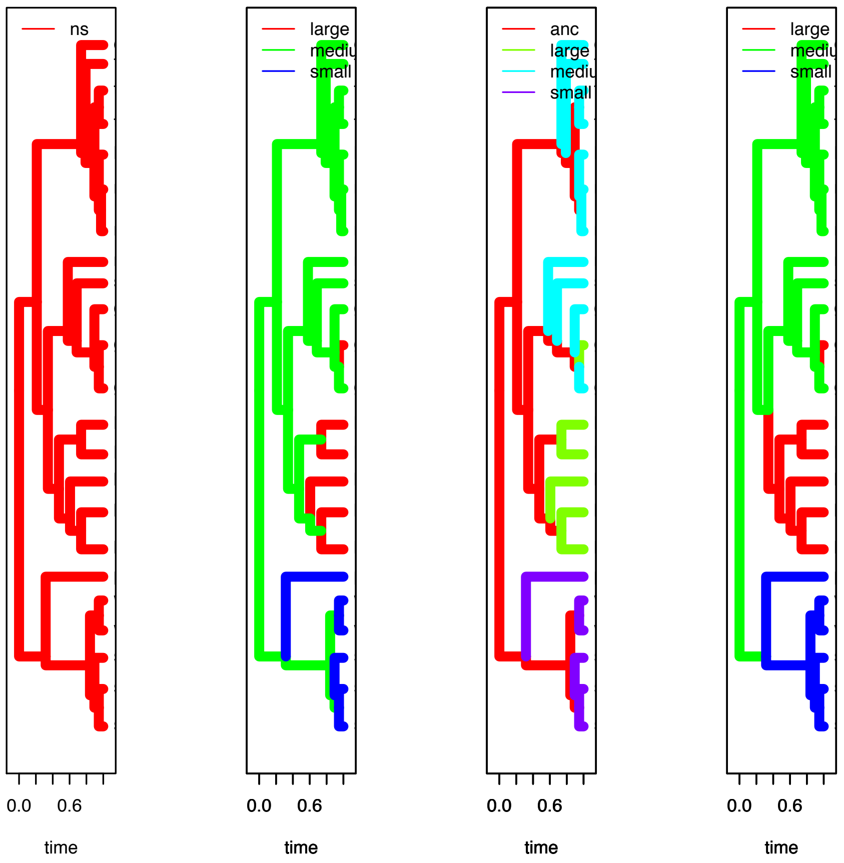
`ouch` fits the OU model Eq. 17.2 along each branch of the phylogeny. While α and σ are held constant across the entire tree, the optima along each branch θ are allowed to vary. Users can then "paint" various combinations of optima on the tree to reflect various biological scenarios.

For example, the dataset `bimac` was used to test the hypothesis of character displacement using an interspecific daaset of body sizes and sympatry/allopatry **?**. 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).

17.3.2 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.

17.3.3 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'],sqrt.alpha=1,sigma=1)
> h3 <- hansen(log(bimac['size']),tree,bimac['OU.3'], sqrt.alpha =1,sigma=1)
> h4 <- hansen(log(bimac['size']),tree,bimac['OU.4'], sqrt.alpha =1,sigma=1)
> h5 <- hansen(log(bimac['size']),tree,bimac['OU.LP'], sqrt.alpha =1,sigma=1,reltol=1e-5
```

17.3.4 `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


$sqrt.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, sqrt.alpha = sqrt.alpha,
    sigma = sigma)

$conv.code
[1] 0

$optimizer.message
NULL

$alpha
         [,1]
[1,] 1.537793

$sigma.squared
           [,1]
[1,] 0.03969809

$optima
$optima$size
   large   medium    small
3.396533 3.019229 2.408103


$loglik
[1] 24.2015

$deviance
[1] -48.403

$aic
[1] -38.403

$aic.c
[1] -34.87359

$sic
[1] -32.72553

$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   5.655594    0.07516647          3.357344           3.050044
2   6.554647    0.08362697          3.307289           3.039102
3   4.394283    0.05017280          3.388255           3.029544
4   5.219845    0.08214195          3.309318           3.083165
5   4.410114    0.04231034          3.489863           3.144652
6  16.798593    0.18899210          3.297658           3.052313
7   6.441586    0.07012845          3.331735           3.092722
8   5.792469    0.08273575          3.243500           3.047785
9   1.683254    0.04143253          3.463243           3.030148
10  2.184674    0.04651457          3.379991           3.063189
   optima.size.small   loglik       aic     aic.c       sic dof
1           2.546907 26.62548 -43.25095 -39.72154 -37.57348   5
2           2.705521 26.77364 -43.54728 -40.01787 -37.86981   5
3           2.608936 29.03324 -48.06648 -44.53707 -42.38901   5
4           2.589340 24.87756 -39.75512 -36.22571 -34.07765   5
5           2.582032 31.02622 -52.05244 -48.52303 -46.37497   5
6           2.574254 27.11561 -44.23122 -40.70180 -38.55375   5
7           2.613275 28.63411 -47.26822 -43.73881 -41.59075   5
8           2.652373 25.74096 -41.48193 -37.95252 -35.80446   5
9           2.402024 24.22658 -38.45315 -34.92374 -32.77568   5
10          2.527909 24.53509 -39.07018 -35.54077 -33.39271   5
```

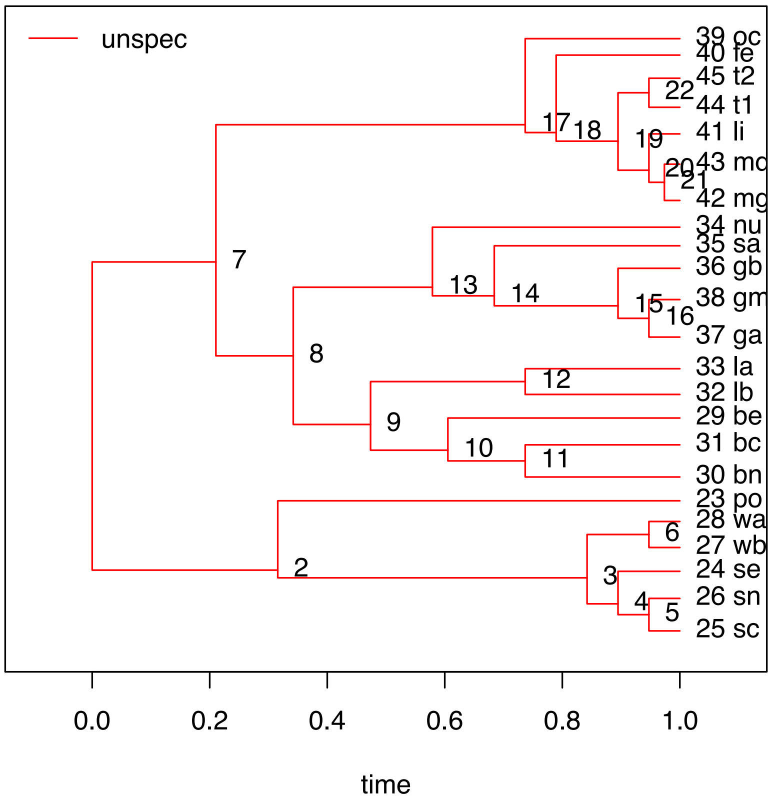
17.3.5 painting regimes on trees

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.

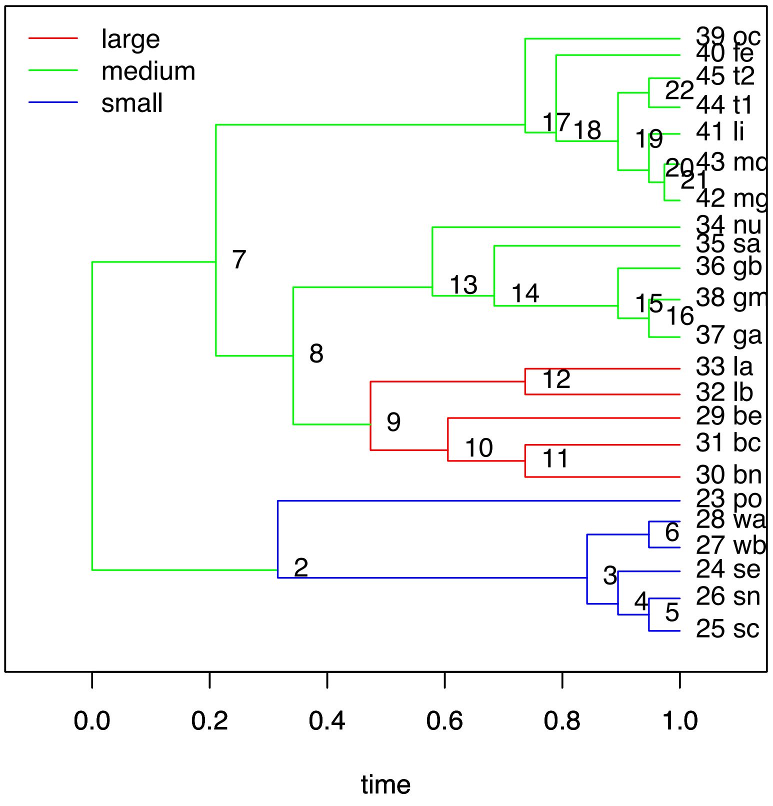
Let's try it on the `bimac` tree and try to recreate the OU.LP regime:

```
> 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)
```

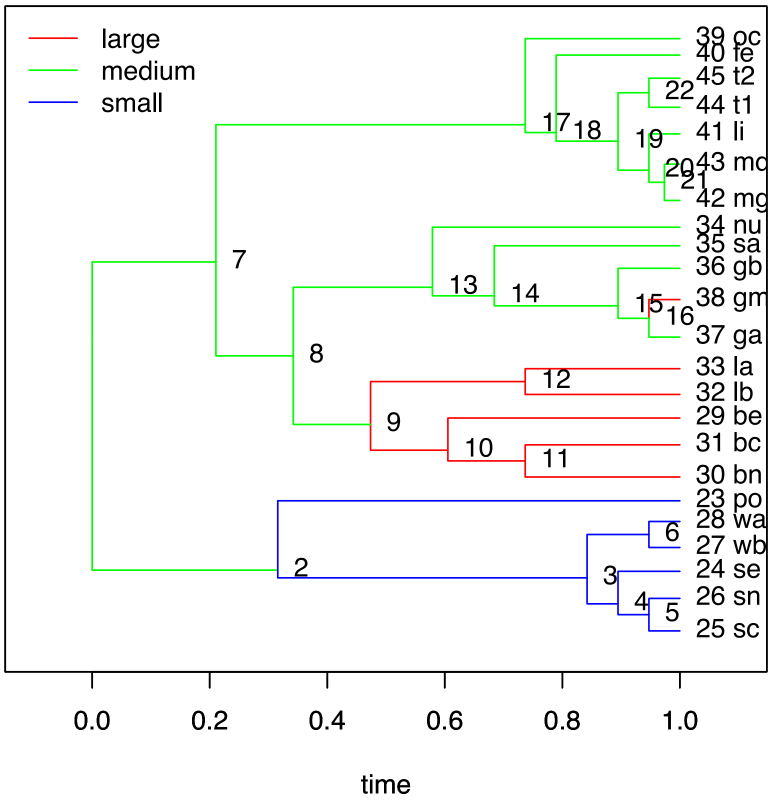


But there was an independent switch from medium to large at species ”gm”, or node ”38”, and the node connecting ”9” to its ancestor:

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

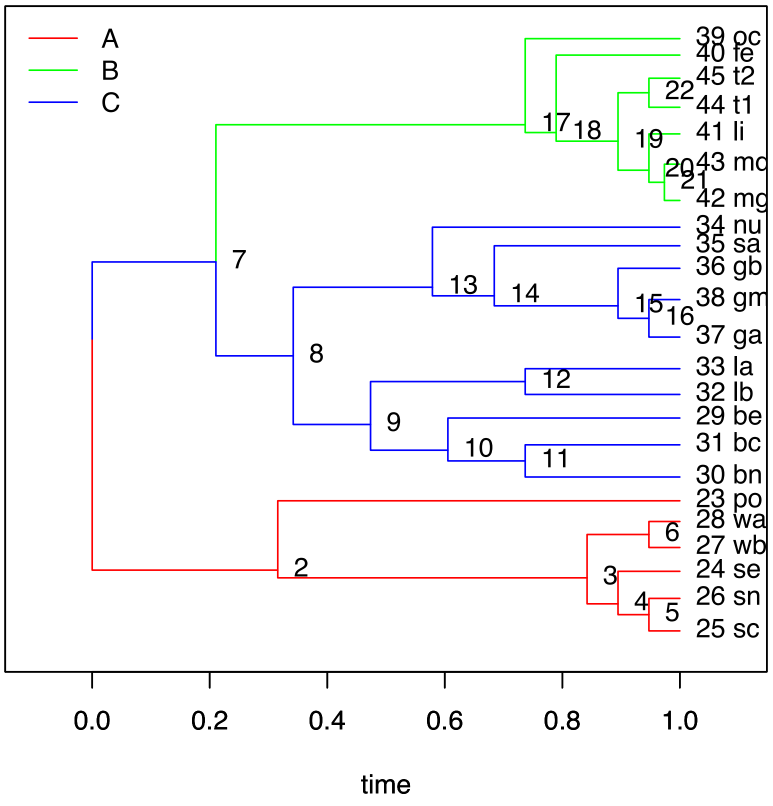
Compare it to the original ”OU.LP” from above.

```
> 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, sqrt.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"     "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 18

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!).

18.0.6 The Bivariate model

The Hansen model describes the evolutionary processes operative on a single quantitative character **??**. In the case of two characters, we will accordingly have two equations.

$$dX_1(t) = \alpha_1 \left(\theta_1 - X_1(t)\right) dt + \sigma_1 \, dB_1(t). \tag{18.1}$$

$$dX_2(t) = \alpha_2 \left(\theta_2 - X_2(t)\right) dt + \sigma_2 \, dB_2(t). \tag{18.2}$$

The above system of eqs. 18.3 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.

18.0.7 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).

18.1 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:

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

18.2 Implementation in ouch

To illustrate, we reanalyzed the evolution of sexual size dimorphism in association with habitat specialization in *Anolis* lizards (**?**), 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], sqrt.alpha = alpha.guess,
+     sigma = sigma.guess, method = "Nelder-Mead", maxit = 3000, reltol = 1e-12)
```

```
call:
hansen(data = xdata, tree = otree, regimes = regimes[5], sqrt.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 log.mSVL
1       1      <NA>  0.00                    twig         twig       NA       NA
2       2         3  0.62             trunk-crown  trunk-crown       NA       NA
3       3         4  0.37             trunk-crown  trunk-crown       NA       NA
4       4         5  0.14             crown-giant  crown-giant       NA       NA
5       5         1  0.08                    twig         twig       NA       NA
6       6         8  0.65            trunk-ground trunk-ground       NA       NA
7       7         8  0.61            trunk-ground trunk-ground       NA       NA
8       8        10  0.54            trunk-ground trunk-ground       NA       NA
9       9        10  0.65              grass-bush   grass-bush       NA       NA
10     10        12  0.50            trunk-ground trunk-ground       NA       NA
11     11        12  0.79             trunk-crown  trunk-crown       NA       NA
12     12        14  0.43            trunk-ground trunk-ground       NA       NA
13     13        14  0.63                   trunk        trunk       NA       NA
14     14        23  0.29            trunk-ground trunk-ground       NA       NA
15     15        16  0.78            trunk-ground trunk-ground       NA       NA
16     16        17  0.57            trunk-ground trunk-ground       NA       NA
17     17        22  0.36            trunk-ground trunk-ground       NA       NA
18     18        19  0.77             trunk-crown  trunk-crown       NA       NA
19     19        20  0.72             trunk-crown  trunk-crown       NA       NA
20     20        21  0.51             trunk-crown  trunk-crown       NA       NA
21     21        22  0.46            trunk-ground trunk-ground       NA       NA
22     22        23  0.27            trunk-ground trunk-ground       NA       NA
23     23        28  0.14            trunk-ground trunk-ground       NA       NA
24     24       124  0.60             trunk-crown  trunk-crown       NA       NA
124   124        25  0.47             trunk-crown  trunk-crown       NA       NA
25     25        26  0.41             trunk-crown  trunk-crown       NA       NA
```

```
26     26       27  0.33                   twig         twig        NA        NA
27     27       28  0.23                   twig         twig        NA        NA
28     28       34  0.11                   twig         twig        NA        NA
29     29      145  0.40            crown-giant  crown-giant        NA        NA
145   145       34  0.28            crown-giant  crown-giant        NA        NA
30     30       31  0.38             grass-bush   grass-bush        NA        NA
31     31       34  0.22                   twig         twig        NA        NA
32     32       33  0.77           trunk-ground trunk-ground        NA        NA
33     33      161  0.72           trunk-ground trunk-ground        NA        NA
161   161       34  0.44           trunk-ground trunk-ground        NA        NA
34     34        1  0.08                   twig         twig        NA        NA
36     36        2  1.00 A_aliniger  trunk-crown  trunk-crown 3.815512 4.063885
62     62       24  1.00 A_allisoni  trunk-crown  trunk-crown 4.100989 4.382027
54     54       17  1.00  A_allogus trunk-ground trunk-ground 3.749504 4.030695
64     64      134  1.00 A_alutaceu   grass-bush   grass-bush 3.487375 3.569533
134   134       27  0.54             grass-bush   grass-bush        NA        NA
60     60      110  1.00 A_angustic         twig         twig 3.653252 3.788725
110   110       26  0.40                   twig         twig        NA        NA
49     49       13  1.00 A_breviros        trunk        trunk 3.693867 3.864931
37     37        2  1.00 A_chlorocy  trunk-crown  trunk-crown 3.992681 4.276666
38     38        3  1.00 A_coelesti  trunk-crown  trunk-crown 4.000034 4.247066
44     44        7  1.00    A_cooki trunk-ground trunk-ground 3.728100 4.085976
43     43        7  1.00 A_cristate trunk-ground trunk-ground 3.797734 4.195697
66     66       29  1.00  A_cuvieri  crown-giant  crown-giant 4.782479 4.877485
70     70       32  1.00  A_cybotes trunk-ground trunk-ground 3.927896 4.188138
50     50       13  1.00 A_distichu        trunk        trunk 3.797734 3.935740
39     39      413  1.00 A_equestri  crown-giant  crown-giant 5.045359 5.129899
413   413        4  0.90            crown-giant  crown-giant        NA        NA
48     48       11  1.00 A_evermann  trunk-crown  trunk-crown 3.958907 4.258446
56     56       18  1.00  A_garmani  crown-giant  crown-giant 4.412798 4.700480
57     57       19  1.00  A_grahami  trunk-crown  trunk-crown 3.784190 4.182050
42     42        6  1.00 A_gundlach trunk-ground trunk-ground 3.811097 4.171306
35     35      403  1.00 A_henderso   grass-bush   grass-bush 3.693867 3.869116
403   403        4  0.58             grass-bush   grass-bush        NA        NA
53     53       16  1.00 A_homolech trunk-ground trunk-ground 3.706228 3.958907
69     69      150  1.00 A_insolitu         twig         twig 3.676301 3.737670
150   150       31  0.65                   twig         twig        NA        NA
46     46        9  1.00    A_krugi   grass-bush   grass-bush 3.671225 3.906005
59     59      101  1.00 A_lineatop trunk-ground trunk-ground 3.788725 4.177459
101   101       21  0.62           trunk-ground trunk-ground        NA        NA
61     61      113  1.00 A_loysiana        trunk        trunk 3.575151 3.706228
113   113       25  0.59                  trunk        trunk        NA        NA
40     40        5  1.00 A_occultus         twig         twig 3.668677 3.663562
```

```
68      68          30  1.00  A_olssoni   grass-bush   grass-bush 3.703768 3.802208
55      55          18  1.00 A_opalinus  trunk-crown  trunk-crown 3.701302 3.901973
52      52          15  1.00 A_ophiolep   grass-bush   grass-bush 3.440418 3.600048
41      41           6  1.00 A_poncensi   grass-bush   grass-bush 3.678829 3.819908
63      63          24  1.00 A_porcatus  trunk-crown  trunk-crown 3.998201 4.261270
45      45           9  1.00 A_pulchell   grass-bush   grass-bush 3.610918 3.848018
65      65          29  1.00 A_ricordii  crown-giant  crown-giant 4.933754 5.023222
51      51          15  1.00   A_sagrei trunk-ground trunk-ground 3.688879 3.977811
67      67          30  1.00 A_semiline   grass-bush   grass-bush 3.616309 3.728100
71      71          32  1.00  A_shrevei trunk-ground trunk-ground 3.832980 4.001864
47      47          11  1.00 A_stratulu  trunk-crown  trunk-crown 3.686376 3.843744
58      58          20  1.00 A_valencie         twig         twig 4.226834 4.374498
72      72          33  1.00 A_whiteman trunk-ground trunk-ground 3.873282 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 11.59748 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],
+     sqrt.alpha = alpha.guess, sigma = sigma.guess, method = "subplex",
+     maxit = 20000, reltol = 1e-12)
> toc <- Sys.time()
> print(toc - tic)

Time difference of 1.144624 mins

> summary(h.subplex)

$call
hansen(data = xdata, tree = otree, regimes = regimes[1], sqrt.alpha = alpha.guess,
    sigma = sigma.guess, method = "subplex", maxit = 20000, reltol = 1e-12)

$conv.code
[1] 0

$optimizer.message
NULL

$alpha
          [,1]      [,2]
[1,] 3.0345706 0.7042764
[2,] 0.7042764 4.3493269

$sigma.squared
          [,1]      [,2]
[1,] 0.1232641 0.1585029
[2,] 0.1585029 0.2201611

$optima
$optima$log.fSVL
   ancestral  crown-giant   grass-bush        trunk  trunk-crown trunk-ground         tw
    4.688810     4.957811     3.588247     3.657690     3.888971     3.758807     3.7770
```

```
$optima$log.mSVL
   ancestral  crown-giant   grass-bush        trunk  trunk-crown trunk-ground          tw
    6.001249     4.959877     3.710174     3.775684     4.111575     4.050079     3.8078


$loglik
[1] 82.4903

$deviance
[1] -164.9806

$aic
[1] -124.9806

$aic.c
[1] -109.7079

$sic
[1] -78.36594

$dof
[1] 20
```

The Brownian motion model is fit:

```
> brown.fit <- brown(data = xdata, tree = otree)
> summary(brown.fit)

$call
brown(data = xdata, tree = otree)

$sigma.squared
          [,1]      [,2]
[1,] 0.1523957 0.1564986
[2,] 0.1564986 0.1746020

$theta
$theta$log.fSVL
[1] 3.954663

$theta$log.mSVL
[1] 4.118798
```

```
$loglik
[1] 22.56325

$deviance
[1] -45.12651

$aic
[1] -35.12651

$aic.c
[1] -34.26937

$sic
[1] -23.47284

$dof
[1] 5
```

18.3 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.

18.4 Variations of the OU Model — Brian?

Instead of assuming a constant σ across the entire tree, **?** 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 19

Phylogenetic Community Analysis by Todd Oakley

Make sure your working directory points to the class directory, which in turn has inside a directory called Data

```
> require(picante)
> #setwd("/RClass")
```

Load in the distribution data set. This is just a text file in phylocom format that I created based on the Mayr and Diamond data. Type:

```
> birds<-readsample("Data/bird_dist.phylocom")
```

Load in geographical data on the islands

```
> geo<-read.csv("Data/IslandGeoData.csv", na.strings = "-9999")
```

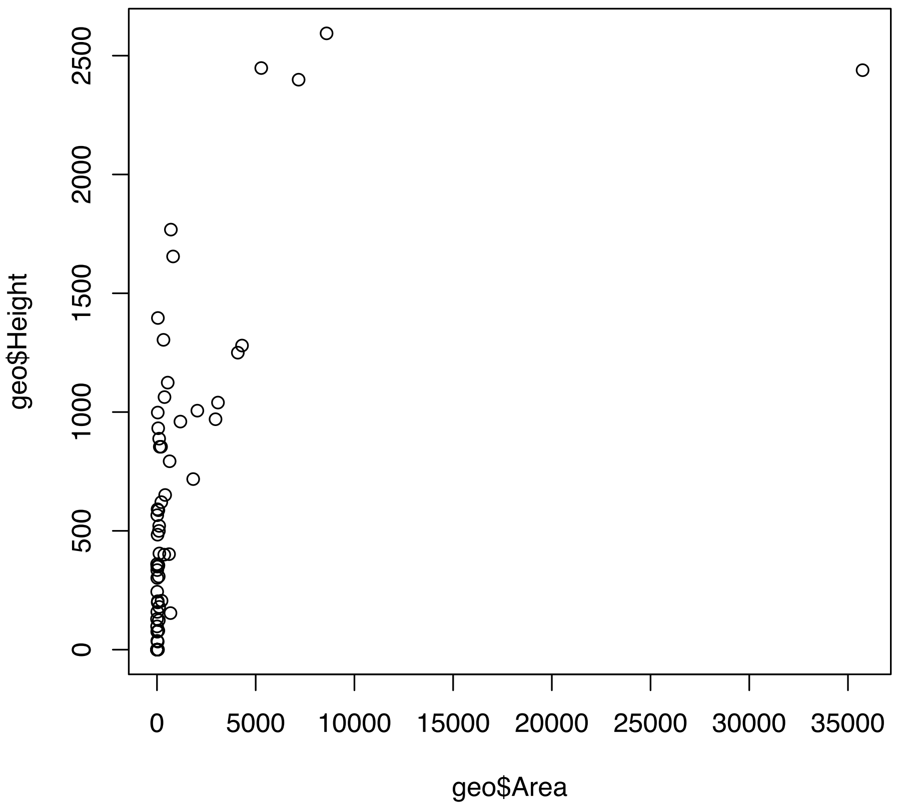
You can see the type of data

```
> head(geo)
```

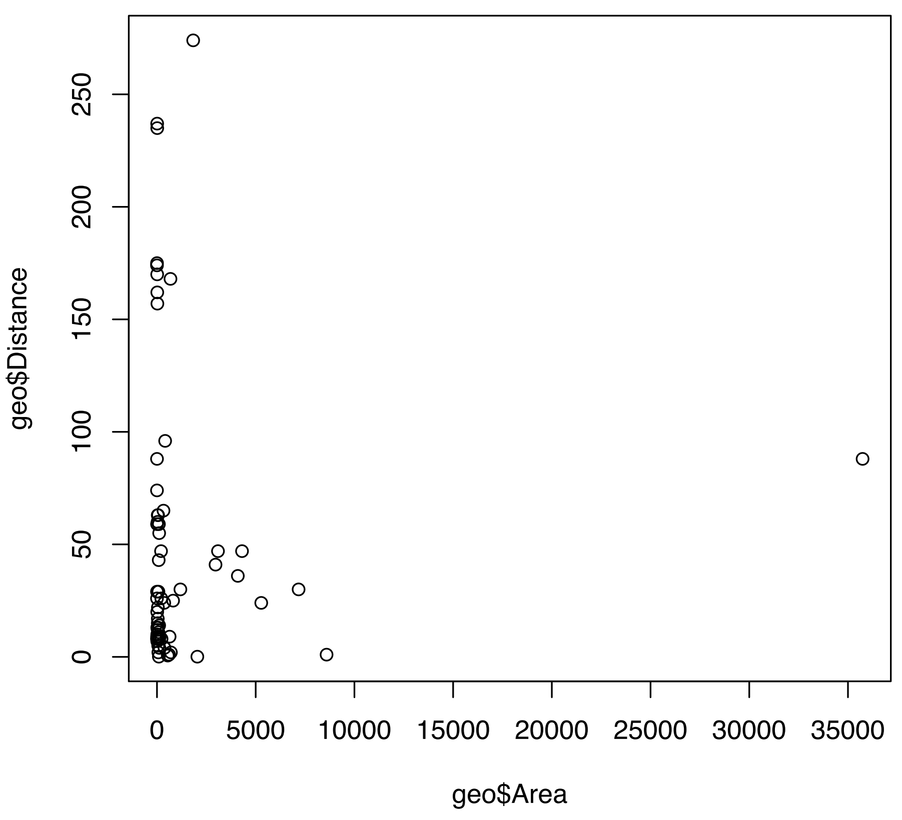
```
           Island Area..km2. Height..m. Distance.to.the.nearest.Island..km.
1   1_Anchorites       0.52          0                                 174
2      1_Credner       1.00          0                                   8
3        1_Crown      14.00        566                                  10
4 1_Duke_of_York      52.00          0                                  13
5        1_Dyaul     110.00        180                                  14
6       1_Emirau      41.00          0                                  17
```

And plot the data

```
> plot(geo$Area, geo$Height)
```



```
> plot(geo$Area, geo$Distance)
```

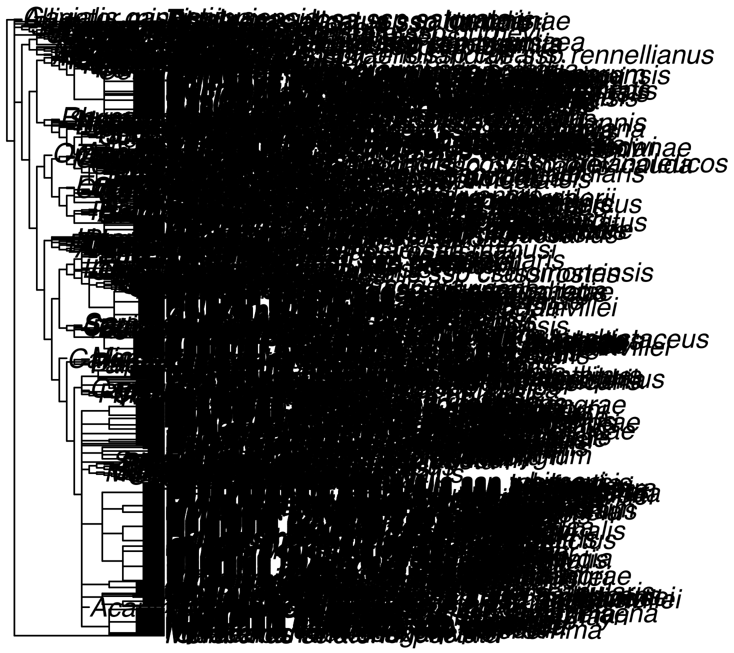


Load in the bird phylogeny. This is just a text file in the newick tree format. The tree is based on the published molecular phylogeny of Hackett et al 2008 in Science. That published tree had only about 100 species, whereas the island dataset has 500 species. The island birds that are not in the published phylogeny are 'grafted' on to the molecular phylogeny based on taxonomy using phylomatic, a program of the phylocom package.

```
> fulltree <- read.tree("Data/BirdTree.tre")
```

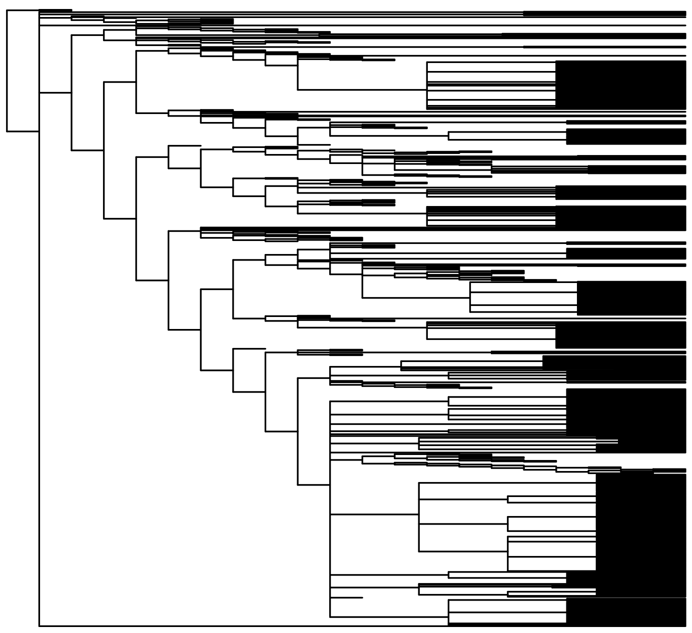
The community phylogenetic data are now in memory and you can conduct many of the analyses described in the picante walkthrough. Note that the 'object' containing the phylogeny is now called "fulltree" whereas in the tutorial it was called `phy`. Also note that the community data are in the object called `birds`, but in the tutorial, it was called `samp`. As one example, you can visualize the phylogeny. This tree is very large, containing over 670 species/subspecies (OTU's), since it's so big its a little ugly to view, but it can be done.

```
> plot(fulltree)
```



To beautify this, we could remove the species names, and still get a sense of what the tree looks like. To do this type:

```
> plot(fulltree, show.tip.label = FALSE)
```



Notice the polytomies, which would be correspond to families or genera for which there is little molecular data in the reference/published phylogeny. Think about how this might impact results.

We can also visualize the distributions of species from a particular island on the full phylogeny (Compare this to section 3 of the picante walkthrough). A similar thing was done in the picante walkthrough. Lets' try it for one island here. First, we need to prune out species not found in at least one community. The tree does contain species in the molecular tree that are not in the Islands studied.

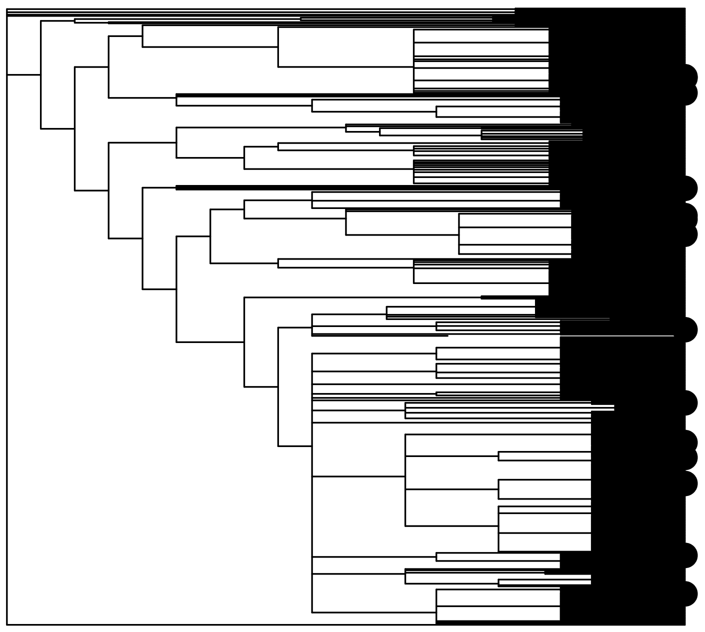
```
> prunedtree <- prune.sample(birds, fulltree)
```

We also need to make sure the species are arranged in the some order in the community data and the phylogeny. This is an important step - several functions in picante assume that the community or trait data and phylogeny data have species arranged in the same order, so it's good to always make sure we've done so before running any analysis. The following command sorts the columns of samp to be in the same order as the tip labels of the phylogeny

```
> birds <- birds[, prunedtree$tip.label]
```

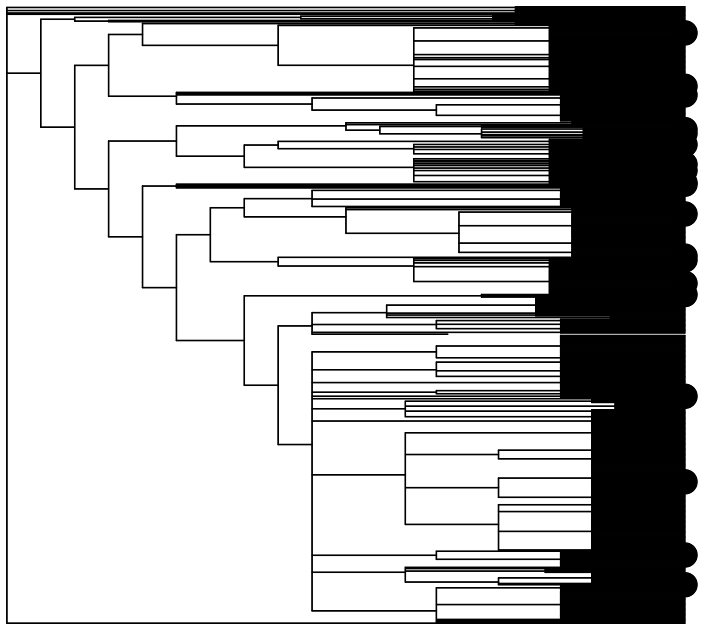
Now, we can identify the birds found on a particular island on the phylogeny to visually inspect the phylogenetic distribution of those birds. To see the distribution for the 2nd island in the list, type:

```
> plot(prunedtree, show.tip.label=FALSE)
> tiplabels(tip=which(birds[2, ] >0), pch=19, cex=2)
```



In the command above, The number after birds[dictates which island is being plotted. You could try plotting different islands on the tree, to get a sense of the data (change the number after birds[ )

```
> plot(prunedtree, show.tip.label=FALSE)
> tiplabels(tip=which(birds[80, ] >0), pch=19, cex=2)
```



The number after cex determines the size of the dots depicting the species, and the number after pch determines which graphic shape will be used. Is this distribution clustered or over-dispersed? How does the 1 passerine (at the bottom of the tree) influence the statistic?

Now imagine we want to know which island contains the most phylogenetic diversity. One of the earliest metrics of eco communities is PD (Faith, 1992) PD is defined as the total branch length spanned by a tree that connects all the species in a community.

```
> pd.result <- pd(birds, prunedtree, include.root=FALSE)
> pd.result
```

```
                           PD  SR
1_Anchorites        114.00000   8
1_Credner           147.16667  14
1_Crown             312.91667  31
1_Duke_of_York      525.90000  55
1_Dyaul             399.01667  46
```

```
1_Emirau          258.33333  24
1_Feni            371.16667  38
1_Hermits         159.33333  16
1_Lihir           533.26667  58
1_Lolobau         451.18333  52
1_Long            552.66667  54
1_Manus           481.30000  51
1_Nauna           198.66667  20
1_New_Britain    1063.02917 126
1_New_Hanover     670.48333  74
1_New_Ireland     848.75833 102
1_Ninigos         177.33333  15
1_Rambutyo        313.71667  29
1_Sakar           335.58333  36
1_San_Miguel      154.33333  13
1_St._Matthias    379.91667  40
1_Tabar           535.51667  59
1_Tanga           384.16667  38
1_Tench           146.66667  13
1_Tingwon         151.91667  13
1_Tolokiwa        405.25000  42
1_Umboi           700.68334  81
1_Unea            208.80000  19
1_Vuatom          517.96666  55
1_Witu            316.43333  30
1_Wuvulu          178.66667  17
2_Bagga           111.25000  10
2_Banika          431.10000  42
2_Bellona         220.66667  19
2_Borokua         173.00000  13
2_Bougainville    811.58333  99
2_Buena_Vista     409.96667  40
2_Buka            611.51667  70
2_Choiseul        671.85000  74
2_Fauro           482.80000  49
2_Fead            131.00000  10
2_Florida         531.76667  58
2_Ganonga         513.46667  53
2_Gatukai         547.88333  57
2_Gizo            572.35000  59
2_Gower           272.66667  26
2_Guadalcanal     885.33334 102
2_Kilinailau       37.00000   4
```

```
2_Kohinggo         546.68333  59
2_Kulambangra      708.91667  81
2_Malaita          644.43333  71
2_Mono             449.25000  44
2_New_Georgia      622.18334  68
2_Nissan           311.66667  32
2_Nukumanu          64.00000   4
2_Ontong_Java      116.00000   8
2_Pavuvu           428.60000  43
2_Ramos            192.00000  17
2_Rendova          583.35000  63
2_Rennell          418.88334  39
2_San_Cristobal    664.85000  77
2_Santa_Anna       451.25000  44
2_Santa_Catalina   401.08333  38
2_Savo             342.46667  34
2_Shortland        502.63333  54
2_Sikaiana          81.00000   5
2_Simbo            396.55000  39
2_Tau               33.33333   3
2_Tetipari         524.35000  55
2_Three_Sisters    388.66667  39
2_Ugi              479.71667  49
2_Ulawa            321.46667  32
2_Vangunu          580.51667  64
2_Vella_Lavella    585.35000  64
2_Wana_Wana        529.35000  57
2_Ysabel           705.43333  79
3_Australia        175.00000  14
3_Celebes           88.00000   6
3_Lesser_Sundas     84.00000   5
3_Moluccas         233.66666  19
3_New_Guinea       396.16666  35
3_New_Hebrides     112.00000   8
3_New_Zealand       59.00000   5
3_North_Caledonia  104.00000   7
3_Philippines      116.00000   7
```

SR is "species richness" the number of species on each island Let's see how PD relats to SR

```
> plot(pd.result$SR, pd.result$PD)
```

PD has been proposed as a metric for conservation prioritization. let's find the row with the highest PD:

```
> pd.result[ pd.result$PD==max(pd.result$PD), ]
```

```
                    PD  SR
1_New_Britain 1063.029 126
```

So New Britain might be a target for conservation based on PD (and SR) But what if the cost of conservation is proportional to island size? We have island size data. Let's find the ratio of PD to Island size to see which island has the most PD for its size

The goal is to merge the pd.results with the data on island size in geo We can do this with the merge function because each object has the island names in it. However, the results from the PD function have islands as row names and the merge command needs to match 2 columns.

Let's take the row names (Islands) from pd.result and put them in a temporary object.

```
> row.names(pd.result)->Island
> Island
```

```
 [1] "1_Anchorites"      "1_Credner"         "1_Crown"
 [4] "1_Duke_of_York"    "1_Dyaul"           "1_Emirau"
 [7] "1_Feni"            "1_Hermits"         "1_Lihir"
[10] "1_Lolobau"         "1_Long"            "1_Manus"
[13] "1_Nauna"           "1_New_Britain"     "1_New_Hanover"
[16] "1_New_Ireland"     "1_Ninigos"         "1_Rambutyo"
[19] "1_Sakar"           "1_San_Miguel"      "1_St._Matthias"
[22] "1_Tabar"           "1_Tanga"           "1_Tench"
[25] "1_Tingwon"         "1_Tolokiwa"        "1_Umboi"
[28] "1_Unea"            "1_Vuatom"          "1_Witu"
[31] "1_Wuvulu"          "2_Bagga"           "2_Banika"
[34] "2_Bellona"         "2_Borokua"         "2_Bougainville"
[37] "2_Buena_Vista"     "2_Buka"            "2_Choiseul"
[40] "2_Fauro"           "2_Fead"            "2_Florida"
[43] "2_Ganonga"         "2_Gatukai"         "2_Gizo"
[46] "2_Gower"           "2_Guadalcanal"     "2_Kilinailau"
[49] "2_Kohinggo"        "2_Kulambangra"     "2_Malaita"
[52] "2_Mono"            "2_New_Georgia"     "2_Nissan"
[55] "2_Nukumanu"        "2_Ontong_Java"     "2_Pavuvu"
[58] "2_Ramos"           "2_Rendova"         "2_Rennell"
[61] "2_San_Cristobal"   "2_Santa_Anna"      "2_Santa_Catalina"
```

```
[64] "2_Savo"           "2_Shortland"      "2_Sikaiana"
[67] "2_Simbo"          "2_Tau"            "2_Tetipari"
[70] "2_Three_Sisters"  "2_Ugi"            "2_Ulawa"
[73] "2_Vangunu"        "2_Vella_Lavella"  "2_Wana_Wana"
[76] "2_Ysabel"         "3_Australia"      "3_Celebes"
[79] "3_Lesser_Sundas"  "3_Moluccas"       "3_New_Guinea"
[82] "3_New_Hebrides"   "3_New_Zealand"    "3_North_Caledonia"
[85] "3_Philippines"
```

Next add those names as a column to the pd.result matrix

```
> new.pd.result <- cbind(Island, pd.result)
```

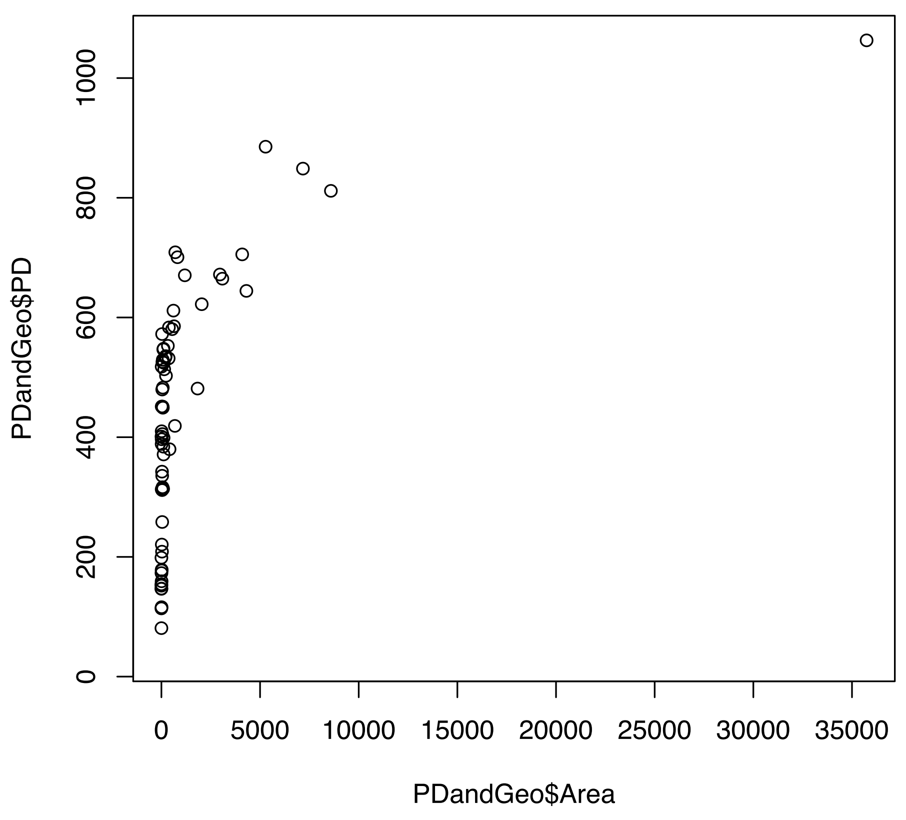
Now we can merge the PD result with the geographic data

```
> PDandGeo <- merge(new.pd.result, geo)
> head(PDandGeo)
```

```
          Island       PD SR Area..km2. Height..m.
1   1_Anchorites 114.0000  8       0.52          0
2      1_Credner 147.1667 14       1.00          0
3        1_Crown 312.9167 31      14.00        566
4 1_Duke_of_York 525.9000 55      52.00          0
5        1_Dyaul 399.0167 46     110.00        180
6       1_Emirau 258.3333 24      41.00          0
  Distance.to.the.nearest.Island..km.
1                                 174
2                                   8
3                                  10
4                                  13
5                                  14
6                                  17
```

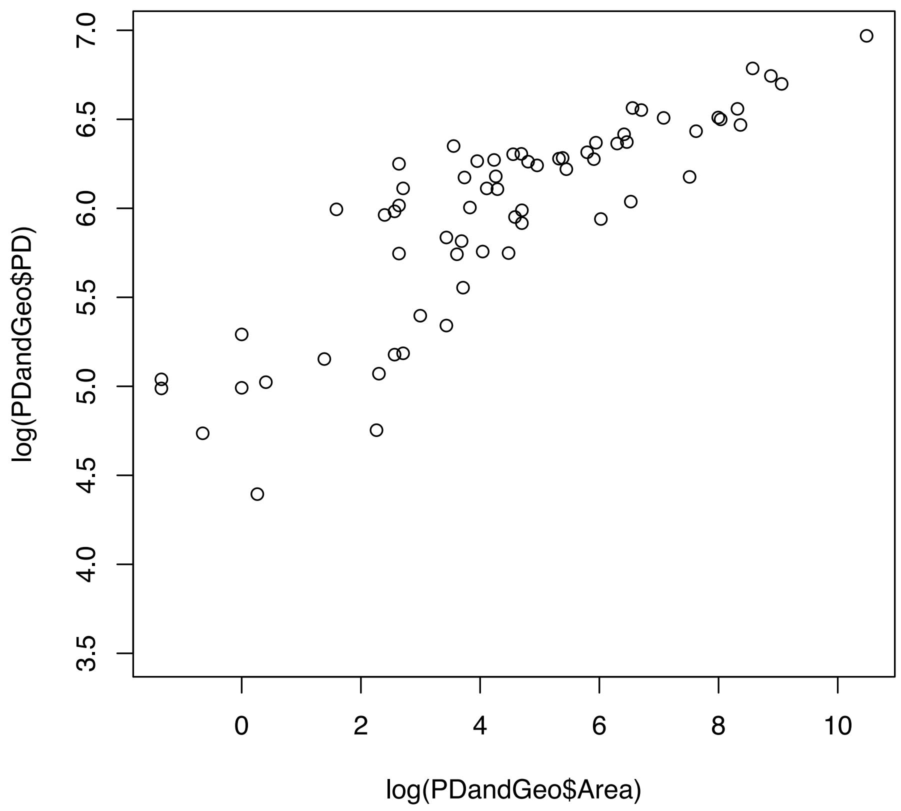
First, out of interest, let's look at the relationship between PD and Island Area

```
> plot(PDandGeo$Area, PDandGeo$PD)
```



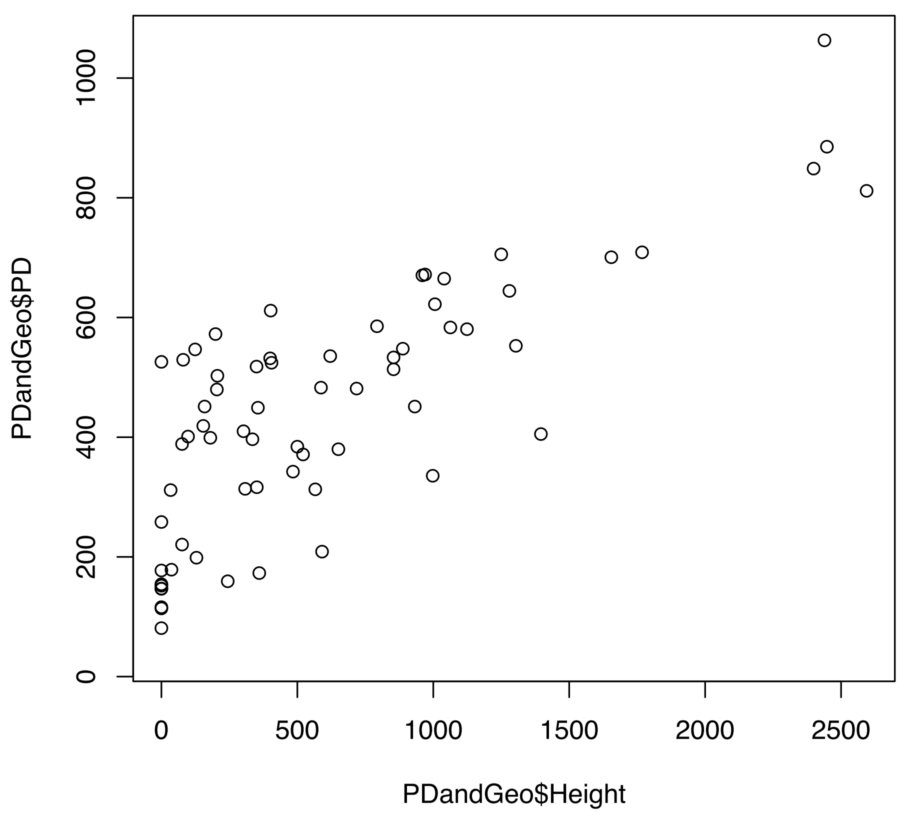
This might be easier to see on a log log plot

```
> plot(log(PDandGeo$Area), log(PDandGeo$PD))
```



Out of interest, let's try the same thing for height

```
> plot(PDandGeo$Height, PDandGeo$PD)
```



Back to the question of the ratio of PD to Island_size:

Let's calculate that ratio and put the result into a new column:

```
> PnG <- cbind(PDandGeo, PDandGeo$PD/PDandGeo$Area)
```

Now we'll find the maximum value of the ratio, and pull out the row the max is in to get the Island name Let's change the names to make it easier to find

```
> names(PnG) <- c("Island", "PD", "SR", "Area", "Height", "Distance", "ratio")
> max(PnG$ratio)
```

```
[1] NA
```

There are null values in the column, and it takes its as max

```
> max(PnG$ratio, na.rm=TRUE)
```

```
[1] 593.5897
```

```
> PnG[ PnG$ratio==max(PnG$ratio, na.rm=TRUE), ]
```

```
             Island       PD SR Area Height Distance    ratio
20    1_San_Miguel 154.3333 13 0.26      0       26 593.5897
NA            <NA>       NA NA   NA     NA       NA       NA
NA.1          <NA>       NA NA   NA     NA       NA       NA
NA.2          <NA>       NA NA   NA     NA       NA       NA
NA.3          <NA>       NA NA   NA     NA       NA       NA
NA.4          <NA>       NA NA   NA     NA       NA       NA
NA.5          <NA>       NA NA   NA     NA       NA       NA
NA.6          <NA>       NA NA   NA     NA       NA       NA
NA.7          <NA>       NA NA   NA     NA       NA       NA
NA.8          <NA>       NA NA   NA     NA       NA       NA
NA.9          <NA>       NA NA   NA     NA       NA       NA
NA.10         <NA>       NA NA   NA     NA       NA       NA
NA.11         <NA>       NA NA   NA     NA       NA       NA
NA.12         <NA>       NA NA   NA     NA       NA       NA
NA.13         <NA>       NA NA   NA     NA       NA       NA
NA.14         <NA>       NA NA   NA     NA       NA       NA
NA.15         <NA>       NA NA   NA     NA       NA       NA
NA.16         <NA>       NA NA   NA     NA       NA       NA
NA.17         <NA>       NA NA   NA     NA       NA       NA
NA.18         <NA>       NA NA   NA     NA       NA       NA
```

So, Manus island has the highest PD for its size

Let's try another community metric besides PD. MPD is the "Mean Phylogenetic Distance" of the species in the community. This metric uses a distance matrix unlike PD, which used the phylogeny directly. We first can get a pairwise distance matrix based on the phylogeny

```
> phydist <- cophenetic(prunedtree)
```

Then we can calculate MPD, along with a statistical analysis of whether the taxa in the community (island) are more clustered or overdispersed than random this is a large tree, and the randomization takes some time. for now, let's just do 10 replications. In practice, ~1000 is better for a published analysis

```
> MPD <- ses.mpd(birds, phydist, null.model="taxa.labels", abundance.weighted=FALSE, r
```

Let's take the row names (Islands) from MPD and put them in a temporary object.

```
> row.names(MPD)->Island
> Island
```

```
 [1] "1_Anchorites"      "1_Credner"         "1_Crown"
 [4] "1_Duke_of_York"    "1_Dyaul"           "1_Emirau"
 [7] "1_Feni"            "1_Hermits"         "1_Lihir"
[10] "1_Lolobau"         "1_Long"            "1_Manus"
[13] "1_Nauna"           "1_New_Britain"     "1_New_Hanover"
[16] "1_New_Ireland"     "1_Ninigos"         "1_Rambutyo"
[19] "1_Sakar"           "1_San_Miguel"      "1_St._Matthias"
[22] "1_Tabar"           "1_Tanga"           "1_Tench"
[25] "1_Tingwon"         "1_Tolokiwa"        "1_Umboi"
[28] "1_Unea"            "1_Vuatom"          "1_Witu"
[31] "1_Wuvulu"          "2_Bagga"           "2_Banika"
[34] "2_Bellona"         "2_Borokua"         "2_Bougainville"
[37] "2_Buena_Vista"     "2_Buka"            "2_Choiseul"
[40] "2_Fauro"           "2_Fead"            "2_Florida"
[43] "2_Ganonga"         "2_Gatukai"         "2_Gizo"
[46] "2_Gower"           "2_Guadalcanal"     "2_Kilinailau"
[49] "2_Kohinggo"        "2_Kulambangra"     "2_Malaita"
[52] "2_Mono"            "2_New_Georgia"     "2_Nissan"
[55] "2_Nukumanu"        "2_Ontong_Java"     "2_Pavuvu"
[58] "2_Ramos"           "2_Rendova"         "2_Rennell"
[61] "2_San_Cristobal"   "2_Santa_Anna"      "2_Santa_Catalina"
[64] "2_Savo"            "2_Shortland"       "2_Sikaiana"
[67] "2_Simbo"           "2_Tau"             "2_Tetipari"
[70] "2_Three_Sisters"   "2_Ugi"             "2_Ulawa"
[73] "2_Vangunu"         "2_Vella_Lavella"   "2_Wana_Wana"
[76] "2_Ysabel"          "3_Australia"       "3_Celebes"
[79] "3_Lesser_Sundas"   "3_Moluccas"        "3_New_Guinea"
[82] "3_New_Hebrides"    "3_New_Zealand"     "3_North_Caledonia"
[85] "3_Philippines"
```

Next, lets add those names as a column back into MPD matrix

```
> MPD <- cbind(Island, MPD)
> PnG <- merge(PnG, MPD)
> head(PnG)
```

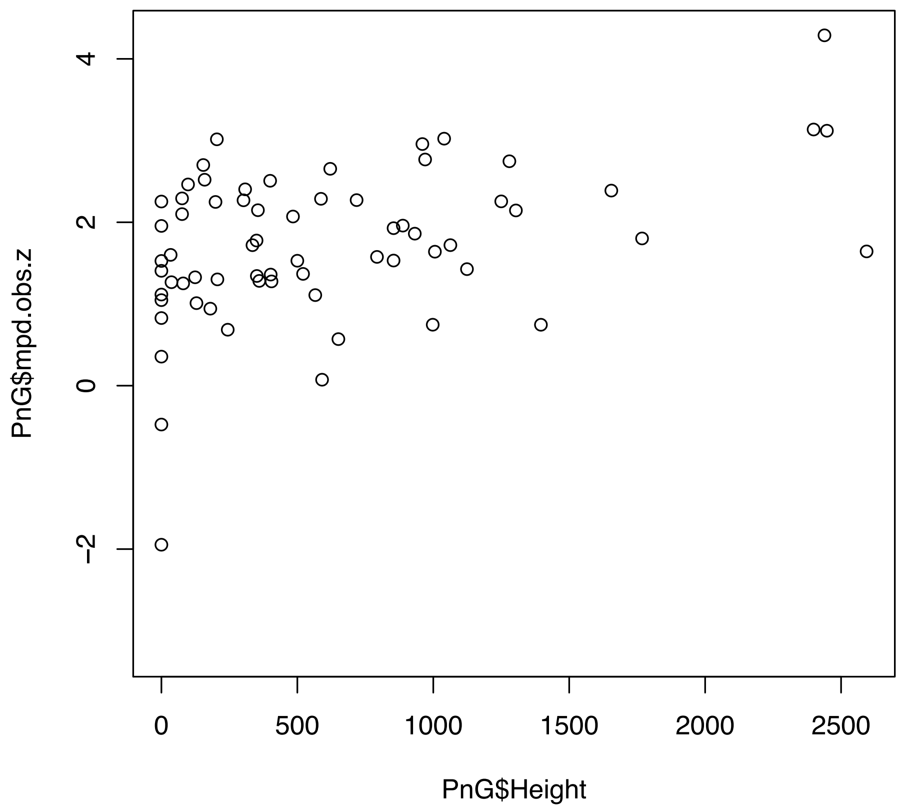
```
         Island       PD SR   Area Height Distance      ratio ntaxa  mpd.obs
1  1_Anchorites 114.0000  8   0.52      0      174 219.230771     8 32.50000
2     1_Credner 147.1667 14   1.00      0        8 147.166666    14 29.18132
3       1_Crown 312.9167 31  14.00    566       10  22.351190    31 30.95591
```

```
4 1_Duke_of_York 525.9000 55  52.00      0       13  10.113461    55 31.61140
5        1_Dyaul 399.0167 46 110.00    180       14   3.627424    46 30.26454
6       1_Emirau 258.3333 24  41.00      0       17   6.300813    24 31.73913
  mpd.rand.mean mpd.rand.sd mpd.obs.rank  mpd.obs.z  mpd.obs.p runs
1      28.95929   1.5706963           11  2.2542323 1.00000000   10
2      30.50513   0.6800313            1 -1.9466897 0.09090909   10
3      29.59191   1.2300569           11  1.1088975 1.00000000   10
4      29.91005   0.8700440           11  1.9554770 1.00000000   10
5      29.46159   0.8516380            9  0.9428263 0.81818182   10
6      29.89727   1.3100318           10  1.4059658 0.90909091   10
```

the columns are the MPD stat and the deviations from the null (in this case random shuffles)

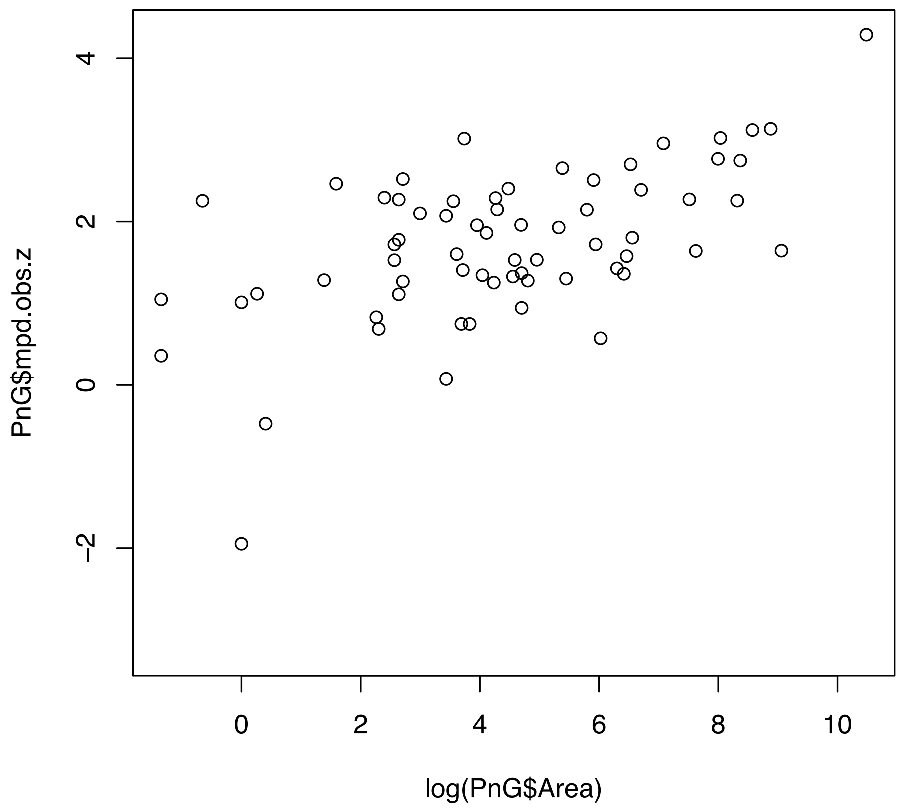
Now, let's compare the level of phylogenetic dispersion of each community to the height of the island where the community resides.

```
> plot(PnG$Height, PnG$mpd.obs.z)
```



How about island area?

```
> plot(log(PnG$Area), PnG$mpd.obs.z)
```



Chapter 20

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

20.1 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 `(){}'[]$`

20.2 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`'.

20.3 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.

20.3.1 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.

20.3.2 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.

20.4 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
```

20.5 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
```

20.6 Distributing the package

20.6.1 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.

20.6.2 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.

20.6.3 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 21

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.

21.1 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 22

Other Packages Available For Comparative Analysis

22.1 `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@

22.2 `geiger`

22.3 `picante`

http://picante.r-forge.r-project.org/

`caic` is coming soon!

Bibliography

Felsenstein, J. 2004. Inferring Phylogenies. Sinauer, Sunderland, Mass.

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.

Paradis, E. 2006. Analysis of Phylogenetics and Evolution with R, volume XII of *Use R*. Springer-Verlag.